MÉMOIRE

présenté en vue d'obtenir

LE DIPLÔME D'INGÉNIEUR I.I.E.

Rapport final

Yoann Guillot

# Etude de faisabilité de services de sécurité sur systèmes embarqués

*Mars - Août 2005*

Directeur de stage : Fabrice Stevens, *Ingénieur*

France Télécom, division R&D,
équipe MAPS/NSS/SII
38-40, rue du Général Leclerc
92794 Issy les Moulineaux, Cedex 9

Soutenu devant le jury
M. Yvan Augé, Maître de conférences
M. Olivier Pons, Maître de conférences

# Table des matières

# Chapitre 1

# Objet de la recherche

## 1.1 Encadrement et environnement

Dans le cadre de la validation de ma troisième année d'études à l'IIE, j'ai été accueilli par le département Recherche et Développement de France Télécom pour y effectuer un stage de recherche. J'ai intégré le l'unité R&D Sécurité Internet/Intranet sous la direction de Fabrice Stevens.

### 1.1.1 Coordonnées

Responsable du stage :
**Fabrice Stevens**
France Telecom, division R&D
38-40, rue du Général Leclerc
92794 Issy les Moulineaux
Tel : 01.45.29.81.64
Email : `fabrice.stevens@francetelecom.com`

Stagiaire :
**Yoann Guillot**
19, rue des Mazières
91000 EVRY
Tel : 01.60.79.16.47
Mail : `guillot@ofjj.net`

### 1.1.2 Présentation de l'établissement et du département

France Télécom division Recherche et Développement[1], leader européen de la recherche et du développement en télécommunications, regroupe l'ensemble des activités de Recherche et Développement de France Télécom. Son objectif est de créer le maximum de valeur pour l'opérateur.

Avec 3000 ingénieurs et chercheurs répartis sur 13 sites, dont Boston, Londres, San Francisco, Tokyo ; France Télécom R&D présente un véritable atout pour l'opérateur. De plus,

[1] anciennement CNET, Centre National d'Étude des Télécommunications

son organisation transverse et décentralisée permet une ouverture internationale du groupe et ainsi de s'impliquer auprès des grands groupes industriels et de la communauté scientifique mondiale.

Le Centre de Recherche et Développement MAPS (Middleware et Plateformes Avancées) a la responsabilité de :

- développer les composantes middleware (primitives) et les plateformes de services de l'opérateur intégré,
- définir un urbanisme de plateformes de services interopérables, permettant une ouverture à des tiers contrôlée par l'opérateur, mutualisant les fonctions et données communes, et en cohérence avec le SI et le cœur de réseau,
- définir une architecture technique modulable et les outils de création de services associés pour diminuer le time to market des services et les coûts,
- déployer les services de communication de production et de consommation de contenus sur tous les terminaux.

Dans ce contexte, l'enjeu pour France Télécom est de maîtriser les plateformes de services, les passerelles et les terminaux, et d'en gérer la complexité pour améliorer la réactivité des développements de nouveaux services.

Le développement des synergies et la convergence de services, la préservation du positionnement de l'opérateur dans la chaine de valeur, ainsi que la simplification de l'offre aux clients, sont avec la garantie de la qualité de service et la sécurité des enjeux essentiels.

Le laboratoire NSS (Sécurité des Services et des Réseaux) a pour mission de définir et développer les solutions de sécurité permettant de protéger contre les malveillances les systèmes de l'opérateur et de ses clients (réseaux, plates-formes, SI, passerelles, terminaux). Il est également chargé de développer une offre de services de confiance, directement perçue par le client, ainsi que de sécuriser l'accès aux services et leur exécution.

Le laboratoire développe le patrimoine intellectuel du groupe en matière de sécurité (cryptographie notamment), produit des briques de confiance (composants logiciels, algorithmes, plates-formes, ...) et développe une expertise technique lui permettant de contribuer à l'aspect sécurité des projets de croissance du groupe.

L'Unité de Recherche et Développement SII (Sécurité Internet Intranet) est le groupe qui est plus particulièrement concerné par la sécurité des terminaux et des réseaux.

### 1.1.3 Environnement de travail

Un ordinateur de type PC est à ma disposition, il est équippé de 30Go de disque dur et de 512Mo de mémoire vive, et est équipé d'un processeur cadencé à 2.5GHz, sur lequel j'ai pu installer le système d'exploitation GNU/Linux, distribution Debian.

Je dispose également de plusieurs switches/routeurs Wrt54g.

Ce sont des équipements réseaux destinés à un usage par des particulier. Ils possèdent des capacités de routage, et sont capable de communiquer sur support ethernet et/ou WiFi.

Leur système d'exploitation est basé sur GNU/Linux, et est open-source, donc librement modifiable ; c'est la raison principale du choix de cet équipement par rapport à d'autres. Le système d'exploitation choisi est OpenWRT [3], un système développé par une communauté d'utilisateurs sur internet.

## 1.2 Présentation du sujet

### 1.2.1 Description du sujet du stage

Le stage se situe dans un domaine en plein essor, celui de la sécurité informatique.

Il s'agit en l'occurence de l'étude de services de sécurité sur les systèmes embarqués. Cela va donc consister dans un premier temps à essayer de faire une liste de tous les services de sécurité qu'il serait opportun de placer sur un système embarqué. L'ensemble des systèmes embarqués existant étant très large, l'étude se limitera à des appareils de caractéristiques se rapprochant du Wrt54g, que l'on détaillera dans la suite. Ensuite, parmi la liste de logiciels sélectionnés, il conviendra de déterminer lesquels sont adaptables aux systèmes embarqués, et enfin de mettre en place une série de prototypes.

Ce type d'équipement est de plus en plus présent, aussi bien chez les particuliers que chez les entreprises. Ceci est dû principalement à leur excellent rapport qualité/prix, pour un certain nombre de tâches spécifiques, où ils peuvent remplacer avantageusement un équipement de type ordinateur de bureau. Leur "exotisme" les rend cependant sujet à certaines erreurs, aussi bien de la part de l'administrateur qui n'y est pas habitué, que de la part du programmeur, qui ignore souvent les spécificités liés à ce type de matériel. En particulier ces systèmes sont beaucoup plus souvent dans une situation de manque de mémoire vive qu'un équipement de type PC (du fait de leur faible réserve de mémoire physique et de l'absence de mémoire *swap*)

Il devient donc nécessaire de déterminer les dispositifs de sécurité qu'il est possible d'installer sur ces équipements, en tenant compte de leurs contraintes particulières :

- puissance de calcul limitée,
- espace mémoire limité (espace disque et mémoire vive),
- limitation des entrées/sorties (pas d'écran, ni de clavier),
- contraintes particulières (batterie limitée).

Le terme "services de sécurité" est volontairement vague, afin d'englober le maximum de services possibles. Une partie de mon stage a été consacrée à lister ces services, qui seront détaillés plus loin.

Le stage se base exclusivement sur l'utilisation de logiciels libres.

### 1.2.2 Déroulement du stage

Les premières semaines du stage ont été consacrées à l'étude du matériel en question, ainsi qu'à son système d'exploitation. Celui-ci est de type unix, basé sur busybox, qui est une distribution réimplémentant un système unix classique, avec pour principal objectif de minimiser l'espace mémoire utilisé (parfois au détriment de certaines fonctionnalités marginales).

Il m'a également fallu apprendre l'art obscur de la "*cross-compilation*", c'est à dire la manière de compiler un programme sur une certaine architecture pour qu'il s'exécute sur une autre (par exemple compiler un programme destiné à s'exécuter sur un processeur MIPS [12] en le développant sur une plateforme Intel x86).

Les différences ne s'arrêtent pas au changement de type d'instructions assembleur à générer par le compilateur : en particulier le système d'exploitation est différent. La *libc* elle-même est en fait une version recodée et optimisée pour utiliser peu d'espace, et dans quelques cas son comportement est différent de celui de la librairie standard classique. Cette librairie de substitution est nommée *microlibc*. De même, la plupart des autres librairies courantes sur

PC sont également absentes, faute de place.

Force a été de constater qu'il n'existe que peu de librairies cryptographiques haut niveau. Les seules que j'ai pu trouver sont *OpenSSL* [2] et *GNUTLS*, et aucune des deux n'est concue dans l'optique d'une utilisation en environnement contraint. En cherchant une librairie à capacités de plus bas niveau, je suis tombé sur la *libtomcrypt*. C'est une librairie fournissant des primitives cryptographiques (comme "calculer un hash", "chiffrer un bloc de données en AES", ...), que l'on peut donc utiliser pour construire un protocole sécurisé personnalisé de plus haut niveau. Elle est distribuée sous license *public domain*, c'est à dire que le code est librement utilisable pour n'importe quel usage. Elle est développée et maintenue par un canadien, Tom $S^t$ Denis, pendant son temps libre. Il n'existe à ma connaissance aucune autre librairie libre utilisable en environnement limité fournissant ce type de fonctionnalités, bien qu'il existe des alternatives propriétaires.

Je me suis donc documenté sur l'API de cette librairie, ainsi que sur celle de la microlibc, afin de pouvoir réaliser un prototype de librairie d'authentification et de communication sécurisée entre deux parties, la sécurité reposant sur le partage préalable d'un secret.

J'ai en parallèle étudié les différents protocoles de communication existant (en particulier TLS [8] et SSLv3 [10]), afin de voir les différents écueils dans ce domaine, en tenant compte des spécificités du stage. J'ai alors conçu différents protocoles de communication, offrant différents niveaux de fonctionnalités, et je les ai implémentés. Ils sont destinés à être utilisés par-dessus *TCP-IP* [19], et seront présentés en détails dans la suite.

Un service de sécurité en particulier m'avait été demandé par mon maître de stage : un système de téléadministration de règles de pare-feu, permettant potentiellement de centraliser le contrôle des pare-feux d'un ou plusieurs systèmes distants.

Les systèmes cibles ayant un noyau *linux v2.4/2.6* [20], je me suis documenté sur l'interface *netfilter*, qui est l'interface standard de communication avec le sous-système pare-feu réseau du noyau. Cette interface étant assez mal documentée, ma principale source d'informations a été l'étude des sources du logiciel libre *iptables* [5], qui est le seul à utiliser cette interface (au travers de la librairie *libipt*).

J'ai ensuite mis au point un protocole permettant d'échanger des règles de filtrage, en réutilisant la librairie de communication sus-citée pour une transmission sécurisée, et j'ai réalisé une maquette de logiciel permettant l'administration distante de pare-feu.

Un des services de sécurité qui est apparu assez rapidement est l'utilisation d'un système embarqué en tant que honeypot [18]. Un Honeypot est une machine dédiée à la détection d'anomalies réseau, qu'elles soient dues à des erreurs ou qu'elles soient volontaires, en particulier des attaques. Pour cela il existe un logiciel, honeyd, qui permet d'utiliser un équipement réseau en le dotant de ces fonctionnalités. J'ai donc étudié et porté ce logiciel, et j'ai également travaillé avec les chercheurs de France Télécom R&D, de manière à pouvoir créer une maquette qui pourrait s'intégrer dans une architecture honeypot modulaire et potentiellement de grande échelle.

J'ai ensuite étudié différents services de sécurité qui permettraient d'utiliser le système embarqué comme passerelle de protection, pour un ordinateur par exemple (ou un petit réseau local).

Une première piste a été l'utilisation du système embarqué comme proxy d'accès à un compte mail (protocoles POP3 / IMAP [14]), sur lequel serait greffé un système antivirus,

qui permettrait donc de scanner les mails à la volée lors de leur rapatriement chez l'utilisateur, et d'éliminer ceux contenant une charge virale.

Malheureusement le seul logiciel libre permettant de scanner un fichier à la recherche de virus n'est pas adapté pour une utilisation sur un système tel que le Wrt54g.

Ce service n'a donc pas pu être porté sur système embarqué.

Une autre application, semblable, est le filtrage d'URL et de contenu (dans un flux HTTP [13], aussi appelé "Web" ou "Surf sur Internet"). Il existe pour ce type d'application un protocole normalisé, l'Internet Content Adaptation Protocol (ICAP [1]), qui permet de déporter la charge vers un serveur central externe, qui réalise le filtrage lui-même. Cette solution est donc très intéressante, puisqu'on peut obtenir les fonctionnalités d'antivirus, et même plus, sans avoir à en supporter la charge sur le système embarqué.

Un module pour le proxy HTTP *Squid* qui implémente cette fonctionnalité est développé, cependant il est abandonné depuis fin 2003, et aucune autre implémentation de client ICAP n'est adaptable vers un système embarqué (elles sont réalisées en langage de script). De plus il n'existe pas de version finalisée d'un serveur ICAP libre, donc ce service n'a pas pu être implémenté non plus.

## 1.3 Intérêt du travail

### 1.3.1 État de l'art

La sécurité des systèmes informatiques est un domaine qui est en plein essor. Tous les systèmes font l'objet d'études rigoureuses, et d'audits, avant leur mise en production. Les entrées/sorties sont chiffrées pour les communications réseau, les programmes sont plus sûrs, et de très nombreux logiciels sont développés pour protéger l'utilisateur final contre toute sorte de menaces qu'il pourrait rencontrer.

Cependant ces services sont conçus pour une utilisation sur une plate-forme de type ordinateur de bureau, ou serveur, disposant d'énormément de ressources. Le domaine des systèmes embarqués ne bénéficie pas, et de loin, d'autant d'attention.

### 1.3.2 Apport du stage

Ce type de systèmes est très simple à déployer (le matériel est léger, ne nécessite pas beaucoup de place, ne consomme presque rien...), et est donc la cible de choix pour installer des services de sécurité de manière non intrusive sur n'importe quel poste informatique : pas besoin d'installer de logiciel sur le poste final, ainsi que la possibilité d'utiliser un seul système embarqué pour sécuriser éventuellement plusieurs postes finaux.

De plus ce type d'installation permet de séparer l'entité protectrice des postes à protéger, ce qui renforce beaucoup la robustesse de la protection. Comme de plus ces systèmes ne nécessitent quasiment pas de maintenance, l'utilisateur final ne s'y connecte jamais, ainsi il a beaucoup moins de possibilités de faire des erreurs de configuration.

Le but du stage est de voir quelles sont les possibilités de déploiement de services de sécurité sur systèmes embarqués, quels seront ceux qui auront une utilité réelle.

Comme la plupart des éléments de base des services de sécurités ne sont pas développés dans l'optique d'une utilisation en environnement limité, il faudra certainement développer des briques respectant ces contraintes, de manière à pouvoir porter les logiciels intéressant en diminuant leurs besoins en ressources systèmes, sans pour autant sacrifier les caractéristiques de sécurité qui leur sont nécessaires.

Ces briques permettront à France Télécom de rester à la pointe de l'innovation, sur des plateformes parfois méconnues et souvent peu considérées. Ceci contribuera à la réputation de fiabilité de l'opérateur, dans le domaine changeant des nouvelles technologies de l'information.

De plus les logiciels développés permettront de définir un certain nombre de caractéristiques minimales, par exemple en termes de puissance de calcul, ou d'espace mémoire, nécessaires pour faire tourner ce type d'applications. Ceci pourrait s'avérer utile pour la rédaction d'un cahier des charges décrivant les spécifications d'un tel système avant construction, ou plus simplement pour évaluer les différentes offres d'un catalogue afin de commercialiser ce type de services.

### 1.3.3 Intéret par rapport à la formation de l'IIE

Ce stage est une très bonne opportunité sur le plan personnel, car il me permet de découvrir le monde de la recherche de l'intérieur, et non plus de l'extérieur comme j'ai pu le faire tout au long de ma formation, ce qui est important pour mon futur exercice du métier d'ingénieur.

Le développement sur systèmes embarqués est aussi beaucoup plus contraignant sur les techniques de programmation que la programmation “standard”, ce qui m'oblige à encore plus de rigueur dans ma manière de concevoir les logiciels et de les implémenter (d'autant plus qu'ils doivent répondre à des exigences de sécurité). Le fait de devoir optimiser un programme afin qu'il utilise le moins de mémoire possible est un défi intéressant à relever.

La nature même du système embarqué est également intéressante à étudier : je peux tester des programmes sur des machines d'*endianness* différente, ainsi que découvrir de nouvelles architectures, avec leur langage d'assemblage spécifique (même si cela n'est pas directement utile pour mon stage, puisque je me limiterai à une programmation en langage C ).

Je découvre également une nouvelle facette du monde de l'entreprise, en intégrant une grande entreprise, d'envergure internationale, dans une équipe jeune et très dynamique.

# Chapitre 2

# Realisations

## 2.1 Concepts cryptographiques

Voici quelques rappels des principes fondamentaux de la cryptographie [17], qui seront utilisés pour la description des protocoles développés.

### 2.1.1 Confidentialité

La confidentialité d'un message caractérise le fait qu'un attaquant ne puisse pas obtenir d'informations sur ce message.

Cette caractéristique est fournie par le *chiffrement*.

### 2.1.2 Perfect forward secrecy

Cette caractéristique, dont le nom n'a pas d'équivalent français, traduit le fait qu'un attaquant qui enregistrerait tout le trafic chiffré entre deux interlocuteurs, et qui par la suite obtiendrait une ou l'autre des clés secrètes des interlocuteurs (ou les deux), ne serait quand même pas en mesure de décrypter le flux de données échangées.

Cette caractéristique est fournie par l'utilisation de *clefs publiques de session éphémères*.

### 2.1.3 Intégrité

L'intégrité d'un message caractérise le fait que le destinataire d'un message sache que le message qu'il reçoit n'a pas été modifié accidentellement depuis son émission par l'émetteur.

Cette caractéristique est fournie par un *hash* (condensé).

### 2.1.4 Authenticité

L'authenticité d'un message est obtenue quand le destinataire d'un message est sûr de l'identité de l'auteur dudit message. La preuve d'authenticité est aussi une preuve d'intégrité.

Cette caractéristique est fournie par la *signature* ou par un *MAC*.

### 2.1.5 Rejeu

Le rejeu est une attaque consistant à enregistrer une communication chiffrée entre un client et un serveur, et à réemettre tout ou partie du message, sans nécessairement comprendre ce qu'il signifie. Un protocole vulnérable considérerait cette communication comme légitime, et reproduirait les actions définies par le contenu du message.

On peut parer cette attaque en employant un *challenge aléatoire*.

## 2.2 Outils cryptographiques

### 2.2.1 Chiffrement symétrique par bloc en mode chainé

Il existe deux grandes catégories d'algorithmes de chiffrement, les chiffrements symétriques et les chiffrements asymétriques. Nous ne nous intéresserons ici qu'aux chiffrement symétriques (c'est à dire que le chiffrement et le déchiffrement est réalisé avec la même clé).

Outre la symétrie, on peut également caractériser les chiffrements par le fait qu'ils travaillent sur une séquence d'octets ou sur des blocs de taille fixe. Les chiffrements à flot (*stream cipher*) sont les chiffrements octet par octet, mais nous nous utiliserons ici uniquement aux chiffrements par blocs.

Un chiffrement par bloc est un algorithme transformant un bloc de taille fixe de texte en clair en un bloc de même taille. Le bloc obtenu (appelé bloc *chiffré*) semble aléatoire pour un observateur. L'utilisation de la clé avec l'algorithme de déchiffrement permet la transformation inverse.

Le bloc de texte chiffré doit posséder certaines caractéristiques :
- il doit être impossible de déchiffrer le bloc sans connaître la clé,
- il doit être impossible d'obtenir des informations sur le texte en clair à partir du bloc chiffré,
- il doit être impossible de modifier de manière prédictible le texte en clair en modifiant le bloc chiffré.

Comme le chiffrement par blocs travaille sur des blocs de taille fixe, si la taille des blocs ne divise pas la taille du texte, il convient d'ajouter autant qu'il le faut d'octets, de manière à obtenir un texte découpable en un nombre entier de blocs. Ces octets constituent le *padding*.

On pourrait donc envisager la technique suivante pour chiffrer un message :
- découpage du texte en blocs de la bonne taille,
- chiffrement de chaque bloc avec la clé.

Cependant ce mode de chiffrement (appelé *ECB*, Electronic Code Book) présente le défaut suivant : deux blocs identiques seront chiffrés de la même manière, ce qui est détectable par un attaquant, et lui livre alors de l'information sur le texte clair.

Ceci est contraire au principe du chiffrement, il faut donc modifier cet algorithme de telle sorte que la fuite d'information soit éliminée.

Une solution courante (celle qui sera appliquée ici) est de chiffrer le message en mode *CBC* (Chained Block Cipher). Dans ce mode, voici le déroulement des opérations :
- découpage du texte en blocs de la bonne taille,
- application du *ou exclusif* bit à bit du bloc courant avec le résultat du chiffrement du bloc précédent,
- chiffrement du résultat.

Le premier bloc à chiffrer reçoit un traitement spécial : le ou exclusif est appliqué avec d'un bloc spécial, appelé *IV* (Initialization Vector). Ce bloc est transmi de la même manière que la clé de chiffrement.

Attention toutefois : le chiffrement par bloc ne garantit pas que le message n'a pas été modifié.

Exemples d'algorithme pouvant être utilisés avec le mode opératoire CBC : DES, AES.

### 2.2.2 Hash

Un hash cryptographique (appelé parfois one-way hash) est un procédé prenant en entrée une séquence quelconque d'octets, qui produit un condensé de taille fixe. Ce condensé possède les propriétés suivantes :

- Il est impossible de générer deux textes intelligibles possédant le même hash.
- Il est impossible de générer un texte intelligible dont le hash ait une valeur donnée.
- Toute modification du texte original entraîne un changement complet du hash.
- Étant donné un hash, il est impossible de retrouver le texte l'ayant généré.

Ici "impossible" est à prendre au sens informatique, c'est à dire "impossible en un temps raisonnable".

Exemples d'algorithmes : MD5, SHA-1.

### 2.2.3 MAC

Un MAC (Message Authentication Code) est un procédé calculant un hash qui est fonction d'une clé secrète et du message. Ainsi seule une personne possédant aussi le secret est en mesure de vérifier ou de créer le MAC, ce qui assure son authenticité. De plus la non-inversibilité du hash assure le secret du secret. Différents algorithmes de MAC proposent différentes manières d'ajouter le secret au message. Par exemple HMAC [9] est défini comme $H(K \otimes 0x36 + H(K \otimes 0x5C + text))$ où $K$ est la clé secrète, $+$ l'opération de concaténation de texte, $H()$ la fonction de hashage et $\otimes$ l'opération de disjonction exclusive bit à bit.

Exemples d'algorithmes : HMAC-MD5, HMAC-SHA1

### 2.2.4 Signature

Une signature numérique utilise un algorithme de chiffrement asymétrique, c'est à dire un algorithme qui permet de chiffrer un bloc de données avec une clé, mais qui nécessite une deuxième clé (liée à la première) pour le déchiffrement. L'une est désignée comme la *clé privée* alors que l'autre est la *clé publique*. On peut choisir d'utiliser l'une ou l'autre pour le chiffrement.

Le principe de signature repose alors sur le fait qu'une personne, dont la clé publique est connue de tous d'une manière où d'une autre (grâce à un certificat par exemple, mais nous ne nous étendrons pas sur le sujet), soit la seule capable de chiffrer un message qui soit déchiffrable grâce à sa clé publique. Donc en pratique, un émetteur va calculer un *hash* du message qu'il envoie, et le chiffrer grâce à sa clé privée. Ainsi le destinataire, recevant le message, pourra calculer lui aussi le *hash* à l'aide de la clé publique de l'émetteur, et si le déchiffrement de la signature correspond, il sera sûr de l'identité de l'émetteur, car lui seul aura été capable de générer la signature. Ceci dépend également de la force du hashage : car si un attaquant est capable de générer un texte clair intelligible tel que son hash soit le même que celui d'un message déjà envoyé par l'expéditeur, il lui suffira de rajouter la signature dudit message au sien pour le contrefaire, cependant on considère que la génération d'un texte produisant un hash donné est impossible.

### 2.2.5 Challenge

Lors de la négociation d'un échange entre deux interlocuteurs, il est utile d'inclure une phase de challenge-response interactive, qui va permettre aux interlocuteurs d'être sûr de l'identité de leur interlocuteur, au moment où ils pratiquent le challenge-response (afin d'éviter le rejeu de la phase de négociation).

Pour celà, le demandeur va générer un challenge, qui doit avoir la propriété de n'avoir jamais été utilisé dans une communication chiffrée par le secret. En pratique il s'agit généralement d'un grand nombre aléatoire, de telle sorte que la probabilité que celui-ci ait déjà été utilisé soit négligeable. On appelle ce genre de nombres un *nonce*.

Une fois ce challenge généré, il est chiffré de telle sorte que seul l'interlocuteur attendu soit en position de le déchiffrer, et donc de répondre au challenge. Pour cette réponse, il peut par exemple renvoyer le nonce incrémenté (de manière chiffré également). De cette manière, il prouve au demandeur qu'il sait chiffrer et déchiffrer les messages (donc qu'il est l'interlocuteur légitime) sans pour autant livrer d'informations à un espion éventuel. Si l'autre interlocuteur veut aussi valider l'identité de celui avec qui il communique, il devra faire le même genre de procédure, dans l'autre sens.

### 2.2.6 Clefs publiques de session éphémères

Le chiffrement symétrique par blocs est une solution rapide et efficace pour le chiffrement des données. Cependant il repose entièrement sur le secret de la clé (et éventuellement de l'IV). Les deux entités communiquantes ne peuvent s'échanger directement la clé, puisque dans ce cas un espion qui écouterait les communications la verrait et pourrait donc déchiffrer tout le trafic.

Etant donné que les deux interlocuteurs disposent d'un secret commun, on peut alors penser à l'utiliser tout simplement comme clé de chiffrement. Cependant cette démarche présente deux défauts : tout d'abord la même clé est utilisée tout le temps, ce qui affaiblit celle-ci ; et ensuite on perd la *perfect forward secrecy* : si un attaquant obtient après coup le secret commun, il peut alors déchiffrer toutes les communications passées.

Pour pallier à cela, on va utiliser l'échange Diffie-Hellman [15]. C'est un algorithme qui permet à deux interlocuteurs de créer un secret partagé, sans s'échanger de donnée permettant de trouver directement ce secret. Il est basé sur les propriétés du logarithme discret : étant donné un élement $X$ d'un corps fini, et un générateur $g$ de ce corps, on peut calculer simplement $g^X$, mais à l'inverse il est difficile de trouver $X$ connaissant $g$ et $g^X$.

Étant donné ces propriétés, chaque interlocuteur génère un nombre aléatoire, appelons-les $X$ et $Y$. Chacun garde cette valeur chez lui, et calcule la puissance de $g$ correspondante, qu'il envoie à l'autre. Le premier possède alors $X$ et $g^Y$, dont il se sert pour calculer $(g^Y)^X$, ce qui est équivalent à $g^{XY}$, que le deuxième calcule de la même manière. Ainsi chacun connait cette valeur, mais un attaquant qui écouterait les communications, ne connaitrait que $g^X$ et $g^Y$, et ne pourrait donc calculer le secret : on possède bien alors une clé de session éphémère.

Notons que de base, cet algorithme ne garantit pas l'identité de la personne avec qui on communique.

## 2.3 Application - protocole cryptographique

Cette section est consacrée à la description des différents protocoles cyptographiques développés lors du stage, pour aboutir au protocole adopté.

### 2.3.1 Base commune

Tous les protocoles suivants se basent sur l'existence d'un secret partagé (appelé PSK, pour *Pre-Shared Key*), en tant le seul moyen d'authentification des interlocuteurs.

Ces protocoles sont destinés à être utilisés sur un moyen de communication fiable (sans pertes ou désordonnancement), comme *TCP/IP* [19].

### 2.3.2 Première version

La première version du protocole avait pour but simplement d'authentifier le client se connectant au service, au moyen de la PSK, sans offrir de facilité de chiffrement des données, et n'était destiné qu'à l'envoi de messages de la part du client vers le serveur (unidirectionnel).

Ce protocole se déroule en 3 phases : établissement de la session, transmission des données, et terminaison de la session.

Tous les messages échangés dans ce protocole ont le format suivant :

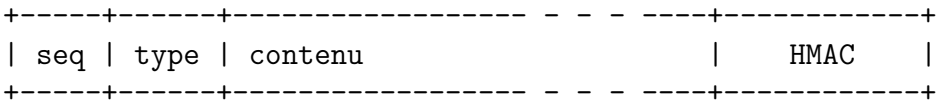



Fig. 2.1 – Format de message - v1

`seq` est le numéro de séquence du paquet, codé sur 1 octet. `type` est le type du paquet. `contenu` est le contenu du paquet, de taille fixe (255 octets). `HMAC` est le HMAC calculé sur tout le reste du paquet, avec le secret partagé. Il fait 16 octets avec le hash MD5, 20 avec le hash SHA-1.

Le client commence par annoncer sa volonté de communication par un message dont le champ *type* comporte la valeur 0 définie comme valeur pour l'établissement de session..

Le serveur lui répond en lui envoyant un message de type 1 (challenge), dont le contenu est un nonce chiffré par la PSK.

Le client répond alors un message de type 2 (réponse), dont le contenu est le nonce incrémenté, concaténé avec un secret de session généré par le client, le tout chiffré avec la PSK.

Les champs séquence et HMAC sont initialisés à 0 pour ces 3 types de messages.

La session est ainsi établie.

Le client envoie alors une série de messages dans des paquets de type 3 (message), dont le numéro de séquence est initialisé à 0 et incrémenté à chaque message, le contenu comporte ce que le client veut transmettre, et le message comporte un HMAC calculé sur tous les champs précédants du paquet, calculé à partir du secret de session défini lors de l'établissement de la

session, et le tout est envoyé.

Quand il a envoyé tous ses messages, le client envoie un paquet de type 4 (fin de transmission). Le numéro de séquence doit être le successeur du numéro de séquence du dernier message envoyé. Le champ contenu est ignoré, et le HMAC est calculé normalement.

Le serveur répond alors par un paquet d'acquitement (type 5), qui porte comme numéro de séquence le numéro du paquet de fin de connection, ainsi qu'un HMAC. Le champ contenu est ignoré.

La session est alors terminée.

En cas d'erreur (type incorrect, réponse incorrecte), la connection est fermée, et les messages précédents sont ignorés.

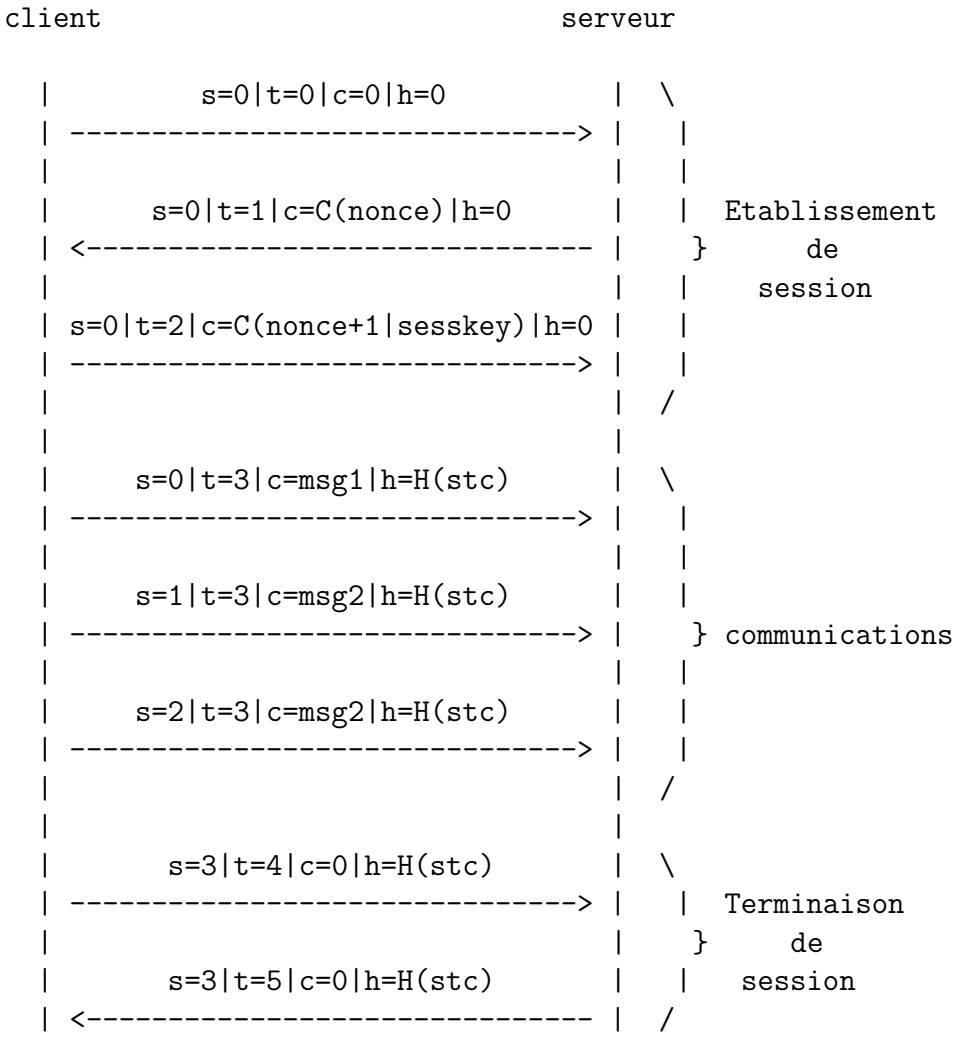



FIG. 2.2 – Protocole 1

Ce protocole permet donc d'authentifier complêtement la communication entre le client et le serveur. Il évite un certain nombre d'attaques :

– Le rejeu ou la suppression à l'intérieur d'une session, grâce aux numéros de séquence.
– La troncature (c'est une attaque consistant à supprimer tous les paquets de la connection entre le client et le serveur à partir d'un certain moment), grâce au protocole de terminaison de session.
– Le rejeu de l'établissement de session, grâce à la nonce.

Cependant ce protocole comporte deux défauts majeurs : tout d'abord il ne chiffre pas les données, mais ceci était un choix de conception, et surtout il n'est pas générique, en particulier il n'est pas adapté aux communications bidirectionnelles (bien qu'il puisse y être adapté). Il aurait donc pu servir à une application de type téléadministration de firewall, mais n'aurait pas été aisément extensible à d'autres type d'applications.

De plus, il oblige à conserver l'historique entier des messages, jusqu'à la réception du message de fin de terminaison, ce qui n'est pas forcément adapté à un protocole qui générerait beaucoup de trafic.

Le système de challenge-response n'est utilisé que pour valider le client, ce qui n'est généralement pas suffisant.

Cette version n'a pas été implémentée.

### 2.3.3 Deuxième version

Une deuxième version à donc été conçue, ses buts étaient cette fois d'être beaucoup plus modulaire, adaptable à d'autres types d'utilisation, et fournissant une fonction de chiffrement.

Ce nouveau protocole comporte une phase d'établissement de session et une phase de transfert de données bidirectionnel. La gestion de numéros de séquence et/ou de messages de fin de connection est laissée à la discrétion du programmeur.

Cette version à été implémentée, elle se présente sous la forme d'une librairie dont l'interface est aussi semblable que possible à celle des appels système unix read()/write().

Pendant la phase d'établissement de session, chaque interlocuteur génère des paramètres Diffie-Hellman [15], qu'il envoie concaténé à un HMAC réalisé avec la clé secrète partagée. À la réception du message par l'autre interlocuteur, l'identité de l'émetteur et l'intégrité du message sont vérifiés grâce au HMAC, et le secret partagé (au sens Diffie-Hellman) est calculé.

Le choix de l'algorithme Diffie-Hellman s'est imposé car il est implémenté de base dans la librairie libtomcrypt, que c'est le seul algorithme implémenté de type négociation de session, et que c'est un grand classique.

Les octets de ce secret de session sont alors partagés en un bloc d'octets pairs et un bloc d'octets impairs, dont le hash par MD5 est utilisé respectivement comme IV et comme clé de chiffrement pour l'algorithme AES [7], qui est utilisé pour les communications. Le hash est utilisé car il a une sortie de taille constante, qui est égale à la taille des clés pour AES, il est donc pratique pour transformer un secret dont on ne contrôle pas forcément la taille en un bloc de taille adaptée à l'utilisation en tant que clé ou qu'IV.

Les octets du secret sont divisés en 2 groupes afin de garantir l'indépendance de la clé et de l'IV.

La session est alors établie.

Les deux interlocuteurs s'échangent alors une série de messages, dans le format suivant :

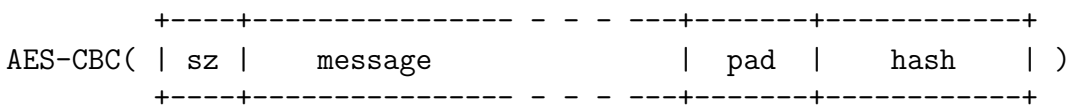



FIG. 2.3 – Format de message - v2

`sz` est la taille du message, encodée sur un octet. La valeur de cet octet est le nombre de bloc de 16 octets occupés par les trois premiers champs du message moins un. 16 octet est la taille du bloc de chiffrement d'AES. `message` est le message lui-même. `pad` est un bloc de padding, de taille variable, qui sert à aligner le message de sorte à ce que le paquet ait une longueur totale qui soit un multiple de 16 octets. Si le message est déjà aligné sur 16 octets, on ajoute un bloc complet de 16 octets de padding. La valeur du dernier octet de padding est le nombre total d'octets de padding utilisés, le lecteur en a besoin pour retrouver la fin réelle du message. `hash` est un hash d'intégrité de 16 octets (avec MD5, 20 avec SHA-1), qui sert de preuve d'authenticité du message, couplé avec le chiffrement AES-CBC. C'est le hash des trois premiers champs du message.

Le message est chiffré par AES en mode CBC avant d'être transmis à l'interlocuteur.

Le déchiffrement d'un message se déroule ainsi :

Le récepteur attend de recevoir les 16 premiers octets du message, qu'il déchiffre afin de lire l'octet spécifiant la taille du message.

Il lit ensuite autant de blocs de 16 octets que nécessaire, en les déchiffrant à la volée.

Une fois toutes les données et le hash reçus, il recalcule le hash et compare la valeur calculée à la valeur reçue, qui doivent être égales. En cas d'erreur, la session est terminée, et une erreur est renvoyée à l'utilisateur. En cas de succès, la valeur du dernier octet précédant le hash est utilisée pour déterminer la taille du padding ajoutée par l'expéditeur, et le message est prêt à être renvoyé à l'utilisateur.

Le hash de contrôle d'intégrité est nécessaire : en effet un attaquant n'est pas en mesure de comprendre le contenu des paquets, mais cela ne l'empêche pas de modifier ces paquets "en aveugle". L'attaquant peut d'ailleurs utiliser le fait que le chiffrement soit réalisé en mode CBC pour modifier d'une manière prédictible un bloc chiffré, au prix d'une modification incontrolée sur le bloc précédent.

AES se contente de déchiffrer les données, tout bloc étant valide en entrée, c'est pourquoi il faut ajouter un contrôle d'intégrité, qui est assuré par le hash. Des tests ont montrés qu'une modification d'un seul bit d'un paquet chiffré modifiait complètement le texte clair résultant du déchiffrement.

La taille maximale d'un message est obtenue avec sz=255 et pad=1, soit $256 * 16 - 1 = 4096o$, soit 4ko. Ceci garantit une faible utilisation de la mémoire de la part de la librairie, sans gêner la communication réseau, car la plupart des réseaux ont une MTU (taille maximale des paquets) de l'ordre de 4000 octets ou moins (1500 pour l'ATM).

Cependant, j'ai découvert lors d'une relecture du code deux faiblesses :

Tout d'abord, le protocole est complètement symétrique, ce qui permet à un attaquant de simplement se connecter, lire les paramètres envoyés par le serveur et les renvoyer tels quels : le HMAC sera alors valide, et la phase d'établissement sera couronnée de succès alors qu'elle

ne devrait pas l'être.

Notons qu'en faisant cela, l'attaquant passe simplement dans l'état "connecté" par rapport au serveur, mais il ne connait pas le secret de session généré ni la clé secrête, il ne peut donc pas communiquer. Pour pallier à ce problème, un octet, "asym" a été ajouté dans la phase de négociation, juste avant le $g^X$, et inclus dans le HMAC ; et une condition supplémentaire a été ajoutée : le champ asym reçu doit être différent de celui envoyé. Le champ asym est constant pour une application, il revient au programmeur de définir cette constante différemment dans le client et dans le serveur.

Notons que ceci ne règle que partiellement le problème, puisqu'il suffit à un attaquant de pouvoir enregistrer une communication légitime, et de rejouer l'établissement de session pour arriver au même résultat. Cependant la modification rend plus difficile techniquement la réalisation de cette pseudo-attaque, puisqu'il faut alors être en mesure d'écouter le réseau, ce qui n'était pas nécessaire précedemment.

La deuxième faille est par contre plus grave (théoriquement), car elle permet à un attaquant de se connecter de manière effective au serveur, sans connaître la clé secrête. Pour cela, il faut qu'il enregistre une connection légitime, qu'il extraie $g^X$ de la phase de négociation, et qu'il casse le Diffie-Hellman (en bruteforcant $X$). Il sera alors en mesure de rejouer cet établissement de session particulier, et connaissant $X$ il sera en mesure de connaitre le secret de session, et donc de communiquer effectivement, en se faisant passer pour un interlocuteur légitime, alors qu'il ne connait pas la PSK.

En pratique, casser un échange Diffie-Hellman prend de plusieurs mois (pour une clé très petite) à plusieurs siècles, voire beaucoup plus avec une clé de longueur correcte.

La correction de cette attaque donne la version finale du protocole.

### 2.3.4 Version finale

Dans cette version, le HMAC de la phase de négociation de session est remplacé par un simple hash, et le premier échange est chiffré en AES en utilisant la clé partagée pour initialiser l'IV et la clé (la PSK est passée dans MD5 pour obtenir la clé, et repassée dans MD5 pour donner l'IV).

Le HMAC est remplacé par un simple hash pour éviter d'utiliser la même clé de deux manières différentes sur un même bloc, ce qui est généralement déconseillé par les cryptologues.

Ceci permet en outre de réutiliser le code existant pour la communication, en supprimant les spécificités liées au format des messages de négociation.

La technique d'intialisation de l'IV et de la clé d'AES à été modifiée : au lieu de diviser en deux groupes les octets du secret et d'en calculer le hash, la clé est définie comme étant le hash du secret complet, et l'IV est le hash de la clé.

Cette modification à été faite compte tenu des conditions pratiques d'utilisation : la PSK consiste en général en un court mot de passe, dont l'entropie n'est pas très élevée, il convient donc de ne pas en perdre plus.

Pour résumer, voici le protocole obtenu :

À la connection, les interlocuteurs génèrent un nombre aléatoire, ainsi que l'exposant Diffie-Hellman correspondant. Ils initialisent également le moteur de chiffrement à l'aide de

la PSK, et envoient à leur interlocuteur un message formé de l'octet "asym", de $g^X$, du padding et du hash de contrôle, chiffré au moyen d'un AES en mode CBC.

Ils recoivent alors le message symétrique, le déchiffrent et vérifient son intégrité et son authenticité grâce au hash, et vérifient la validité du champ "asym", qui doit contenir une valeur différente du champ "asym" du message émis par eux.

Ils peuvent alors calculer le secret partagé de session par l'algorithme de Diffie-Hellman, et en déduire la clé et l'IV d'AES.

Les messages sont ensuite échangés en suivant le protocole version deux.

La preuve d'authenticité ici est le fait de pouvoir chiffrer ou déchiffrer correctement un bloc par AES, le "correctement" étant assuré par le contrôle d'intégrité.

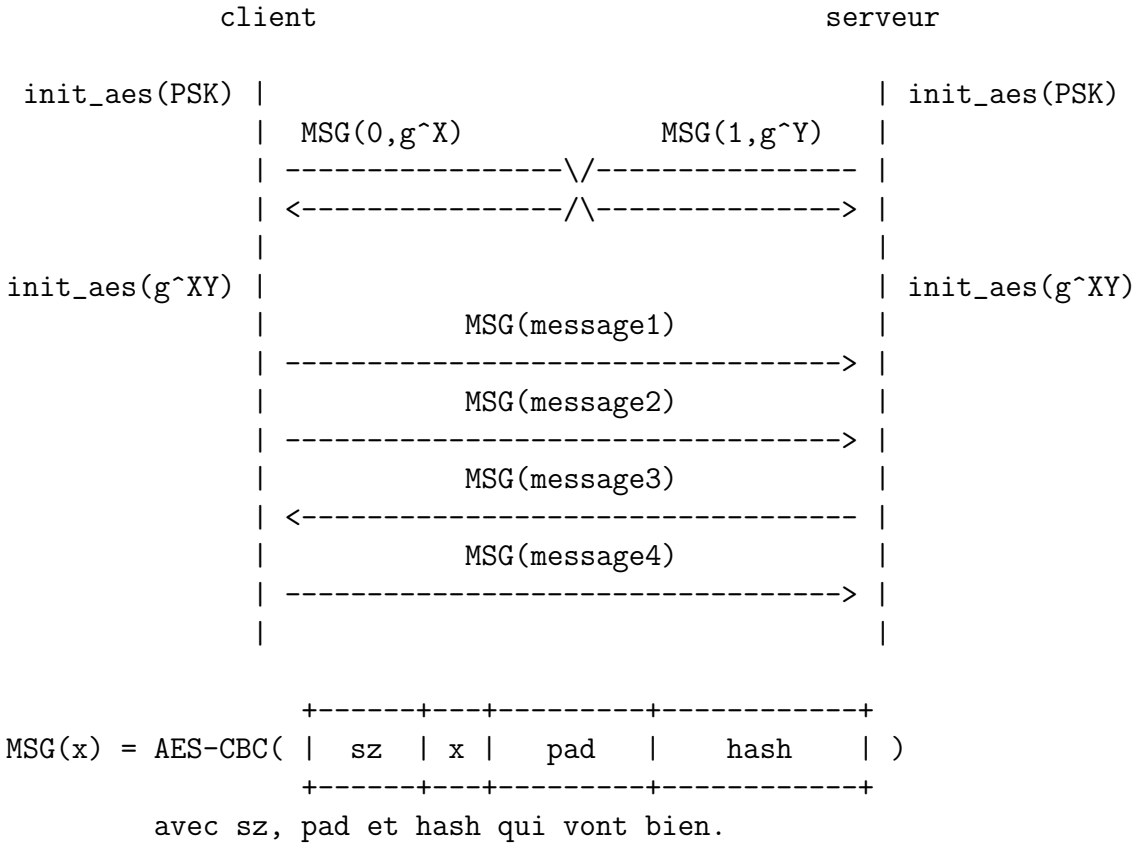



Fig. 2.4 – Communication chiffrée

### 2.3.5 Analyse finale

Le protocole ainsi obtenu est très semblable à celui défini par SSLv2, en lui enlevant les capacités de négociation du chiffrement (ici le chiffrement est immuable, sauf recompilation), et la notion de *canal* (qui permet avec SSL de multiplexer plusieurs flux distincts de messages dans une seule connection chiffrée, mais qui s'éloigne de l'interface de read()/write()).

Ainsi on peut être confiant dans la sécurité de ce protocole, car SSL est un protocole qui a été longuement audité et est considéré comme sûr. Les principales modifications apportées à SSLv2 pour le passage à la v3 sont liées à une amélioration de la phase de négociation des protocoles à utiliser, et ne sont dont pas répercutables sur le protocole développé ici.

Le protocole comporte encore certaines faiblesses, que l'on retrouve également dans SSL, à savoir :

- Il est possible en regardant le flot de messages chiffrés de pouvoir déterminer de manière assez précise la taille du flot de texte clair. Ceci est contrable en générant des messages de manière autonome quand le canal n'est pas utilisé, cependant ceci présente de gros désavantages en termes d'utilisation du réseau, qui sont disproportionnés par rapport à la fuite d'information. Si une application est particulièrement sensible à ce genre de problèmes, elle devra implémenter elle-même un système de messages dont le seul but est d'occuper le canal de communication, ainsi qu'un système de padding supplémentaire afin de cacher la taille réelle des messages transmis.
- Il est également possible d'obtenir des informations sur la durée de calcul nécessaire à la cryptographie pour les différents messages du protocole, en regardant les temps de réponse du serveur. Ceci peut être utilisé pour obtenir de l'information sur par exemple la PSK (cela consitue ce que l'on nomme une *timing attack* [16]), cette information pouvant être utilisée pour améliorer (voire rendre réalisable) une attaque de type bruteforce sur cette clé.
  Pour remédier à ceci il faudrait modifier directement la librairie sous-jacente, en changeant la manière de faire certains calculs, cependant ceci nécessite des connaissances mathématiques que je ne maîtrise pas.
  Notons que les versions récentes d'OpenSSL (9.7e ou supérieure) ne sont plus vulnérables aux timing attacks.

Une amélioration souhaitable aussi serait l'ajout d'un champ spécifiant la version du protocole utilisée lors de la phase de négociation, afin d'assurer une éventuelle compatibilité avec des évolutions futures.

Le code généré consiste en une librairie dynamique de 10ko, ce qui est extrêmement petit comparé aux 1.4Mo d'OpenSSL, même en tenant compte des différences de fonctionnalités. L'empreinte mémoire est aussi beaucoup plus petite.

Concernant les performances, un programme de tests montre que les débits obtenus par les deux librairies sont comparables, de l'ordre de 500ko/s sur un wrt54g.

## 2.4 Téléadministration de pare-feu

La première application conçue est un système d'administration à distance de pare-feu, par l'intermédiaire du réseau.

### 2.4.1 Netfilter

Les machines administrées sont les systèmes embarqués, qui utilisent un noyau Linux version 2.4 ou 2.6. Dans ces version du noyau, la partie pare-feu est nommée *netfilter* [5]. L'administration classique du pare-feu (localement) passe par l'utilisation des commandes iptables (qui sont les trois utilitaires de ligne de commande *iptables*, *iptables-save* et *iptables-restore*). Ces trois utilitaires ont été écrits par Rusty Russell, qui a également écrit la partie noyau de netfilter, ce qui explique probablement le manque de documentation sur l'interface de configuration du noyau.

Conceptuellement, les règles de filtrage peuvent être vues comme une hiérarchie, sur 3 niveaux. Les règles de filtrage pour IPv6 suivent la même hiérarchie que les règles IPv4. Au plus haut niveau, on trouve les *tables*. Elles sont au nombre de 3 pour les versions actuelles du noyau, et ce nombre est fixe. On les adresse par leur nom, que l'on peut trouver dans les pseudo-fichier */proc/net/ip_tables_names* et */proc/net/ip6_tables_names*. Chaque table contient un nombre quelconque de chaînes, qui rassemblent les règles proprement dites. Chaque table comporte un certain nombre de chaînes prédéfinies, que l'on ne peut supprimer (comme la chaîne *INPUT* de la table *filter*). Ces chaînes ont une propriété supplémentaire par rapport aux chaînes ajoutées par l'utilisateur : elles définissent ce que l'on nomme une *policy*, qui permet de déterminer le sort d'un paquet qui ne correspondrait à aucune règle. Ces chaînes sont les points d'entrée des paquets dans le système de filtrage par les règles : par exemple un paquet arrivant sur une interface réseau passe par la chaîne PREROUTING de la table mangle, puis par la chaîne PREROUTING de la table nat, par la chaîne INPUT de la table mangle et enfin par la chaîne INPUT de la table filter. Pour plus d'informations sur l'agencement des tables, se reporter à [4].

Les tables existantes actuellement sont :

1. **filter** qui recoit les paquets après leur routage par le noyau et décide du sort de ceux-ci,
2. **nat** qui voit les paquets dès leur arrivée sur le système et juste avant leur sortie, avant routage par le noyau. Cette table offre la possibilité de modifier les adresses sources et destination de manière à ce que cette modification soit prise en compte par le mécanisme de routage du noyau,
3. et **mangle**, qui permet de modifier différents champs dans les en-têtes des paquets.

Chaque règle comporte un certain nombre de paramètres, qui déterminent le sous-ensemble de paquets réseaux qui "matchent" cette règle. Si un paquet est filtré par une règle, il est alors redirigé vers la "target" de la règle, qui peut être soit une pseudo-chaîne prédéfinie, comme *DROP*, ou le nom d'une autre chaîne de la même table, dans ce cas le paquet est redirigé comme s'il entrait dans ladite chaîne. Si les paramètres de la règle ne correspondent pas au paquet, la règle est ignorée et le paquet est confronté à la règle suivante. Certaines targets effectuent une action avec le paquet mais ne stoppent pas son parcourt de la liste de règles (comme *LOG*, qui enregistre dans un fichier les informations sur le paquet, sans influencer sur son acceptation ou pas par le parefeu). Chaque règle comporte également deux compteurs (respectivement de paquets et d'octets), qui tiennent le décompte du nombre d'application de la règle.

Le pare-feu travaille uniquement au niveau paquets ip, et pas au niveau flux, cependant il existe des modules permettant de rattacher un paquet à un flux déjà vu, et d'appliquer une politique à tout un flux. *conntrack* est un de ces modules.

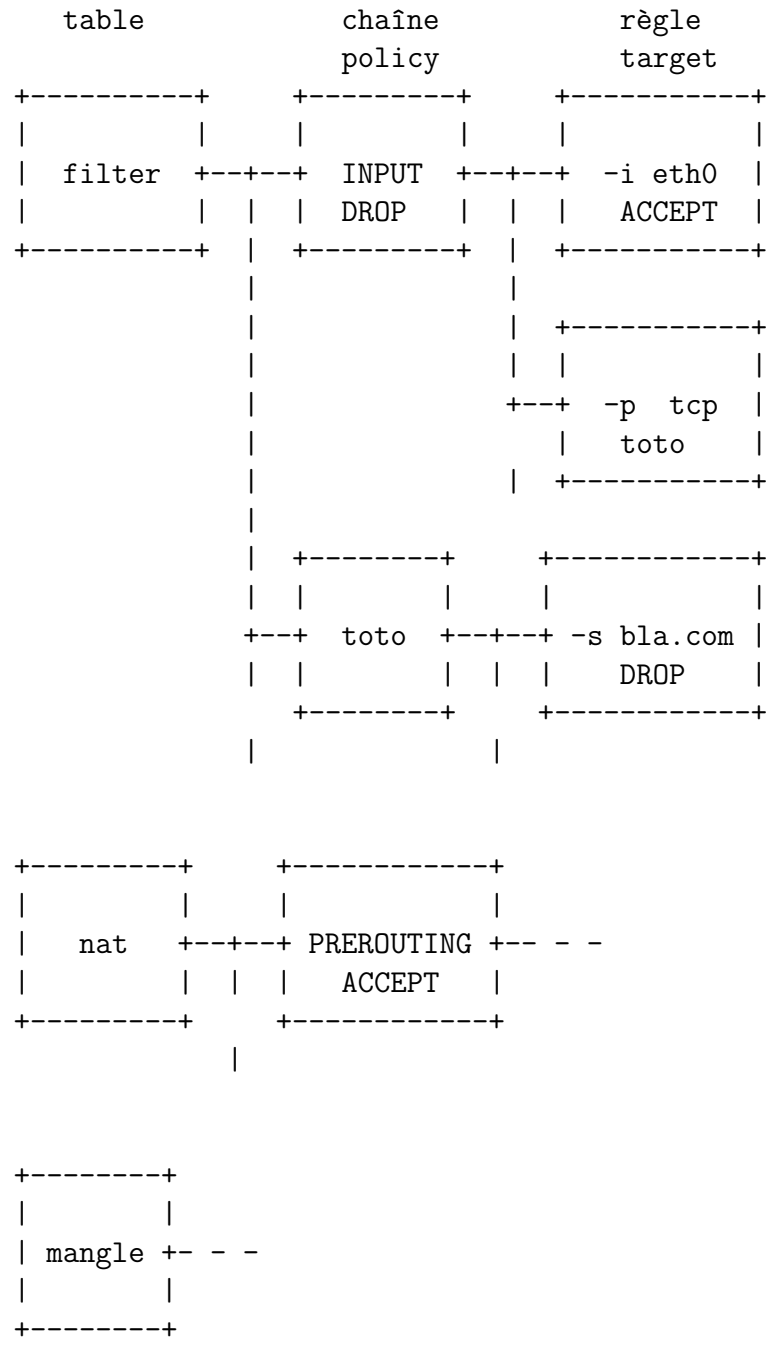



FIG. 2.5 – Hiérarchie de netfilter

### 2.4.2 Sauvegarde - Restauration

L'application de téléadministration doit être capable de recevoir une série quelconque de règles, qui forment un tout. On ne peut se permettre d'en appliquer quelques unes : il faut soit toutes les appliquer soit n'en appliquer aucune, afin de toujours laisser le pare-feu dans un état consistant. L'application doit donc implémenter une forme de transaction, ce qui assurera l'intégrité du pare-feu modifié, même en cas d'erreur lors de l'application d'une règle particulière de filtrage.

Pour cela, l'application va sauvegarder l'état du firewall avant d'appliquer la première règle, de manière à pouvoir le restaurer en cas de problème. Cependant la sauvegarde de l'état du firewall n'est pas une chose aisée, à cause du manque de documentation de l'interface netfilter du noyau. Il n'existe en effet pas de documentation sur l'interface "bas niveau" avec le noyau, et il est recommandé aux gens souhaitant manipuler les règles de filtrage de le faire par l'intermédiaire de la librairie *libiptc* (écrite par Rusty Russell). C'est cette librairie qui est utilisée par les différents utilitaires de ligne de commande. Cependant celle-ci présente une interface assez complexe, et l'examen de son code source révèle un code compliqué, qui utilise beaucoup de mémoire dynamiquement, ce qui n'est pas très adapté pour l'utilisation visée ici.

Une option serait d'utiliser les commandes iptables-save et iptables-restore, qui sont capable de sauvegarder ou de restaurer l'état du firewall sous forme d'un fichier texte, cependant ces programmes exécutent un code compliqué, pour être capable de transformer les règles d'un format en partie binaire en une chaîne de caractères. De plus il n'est pas recommandé de créer inutilement des fichiers sur une mémoire flash, car celle-ci n'a qu'un nombre de cycles d'écriture limité. Il est alors possible d'utiliser un système de fichiers résident entièrement en mémoire (comme *tempfs* ou un *ramdisk*, mais intuitivement, on se doute qu'il existe une manière plus simple de faire un backup complet du parefeu, sans forcément s'intéresser à chaque règle individuellement. Je me suis donc intéressé au fonctionnement des différents utilitaires existants, de manière à essayer de comprendre le fonctionnement de l'interface directe avec le noyau.

Celle-ci permet de traiter en bloc toutes les règles d'une table donnée, que ce soit en lecture ou en écriture. Ainsi il est effectivement plus simple de traiter avec le noyau nous-mêmes pour les besoins de sauvegarde totale et de restauration, plutôt que de déléguer ce traitement à des utilitaires plus versatiles mais plus complexes, qui traitent les règles une par une. Cela ne nécessitera qu'une compréhension minimale des structures manipulées, et sera beaucoup plus léger qu'un traitement par un utilitaire externe.

On peut ainsi de manière très économique sauvegarder complètement l'état du pare-feu ou le restaurer.

### 2.4.3 Format des messages

Intéressons nous maintenant à la transmission desdites règles.

Netfilter est un système extensible très simplement au moyen de modules noyau, qui peuvent rajouter des fonctionnalités au pare-feu. Ces fonctionnalités sont paramétrables, mais les utilitaires ou même le noyau ne peuvent savoir à l'avance quels seront les paramètres nécessaires à un module netfilter ou à un autre. L'interface avec le noyau permet donc de passer tout simplement une chaîne de caractères, qui sera lue par le module utilisé afin qu'il détermine les paramètres souhaités par l'utilisateur. Cette versatilité empêche d'écrire une

grammaire, ou tout simplement de formaliser le langage décrivant une règle de pare-feu. On ne peut dont encoder complètement les règles de filtrage dans un langage binaire : aussi, par soucis de simplicité, les règles seront transmises intégralement en tant que chaînes de caractères.

Le traitement d'une nouvelle règle, en vue de son insertion dans le noyau, est beaucoup plus complexe que la simple sauvegarde ou restauration de l'état complet du parefeu : il faut vérifier que la règle ait une syntaxe cohérente, que le nom de la chaîne donnée existe, ainsi que celui de la target, et faire toute une série de tests de validation des paramètres. Chacun de ces tests étant susceptible d'évoluer en fonction des nouveaux modules netfilter, de nouvelles tables, ou de toute évolution du pare-feu. Ne souhaitant pas assurer un support à vie de ce produit, il a été décidé de laisser ce traitement à l'utilitaire dédié, à savoir *iptables* (et *ip6tables* pour le pare-feu ipv6). Comparativement, la sauvegarde ou la restauration de l'état complet du parefeu ne nécéssite qu'une compréhension sommaire des données lues depuis le noyau ou écrites vers celui-ci, car on sait à la lecture que le bloc de données lu est valide, on peut donc le réécrire sans crainte.

Chaque règle sera donc transmise sous forme d'une chaîne de caractères, qui sera passée telle quelle en tant que liste d'arguments à l'utilitaire iptables ou ip6tables. Il serait cependant dangereux de laisser l'utilisateur spécifier lui-même le nom de l'utilitaire, car cela lui permettrait d'accéder à beaucoup plus que la simple administration du pare-feu, il pourrait par exemple spécifier l'utilitaire */bin/sh*... Aussi ne sera transmise sous forme de chaîne que la partie "arguments" de la commande, la commande elle-même sera codée dans un champ particulier du paquet transmis. Ceci permet également de pouvoir spécifier une caractéristique supplémentaire pour chaque règle, qui est l'autorisation ou non de provoquer une erreur. On peut ainsi dire "appliquer cette règle, en cas d'erreur annuler la transaction" ou bien "appliquer cette règle, en cas d'erreur ignorer l'erreur et continuer le traitement".

En cas d'erreur il est souhaitable de pouvoir spécifier quelle règle du groupe a causé l'erreur, aussi chaque règle sera numérotée lors de l'envoi.

Ceci nous amène au format de message suivant :

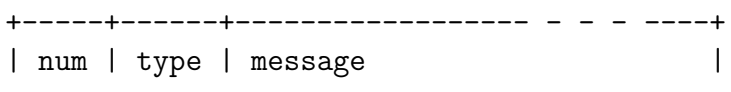



FIG. 2.6 – Format de message de transmission de règle

Le premier octet du message transmis contient le numéro de la règle, le second octet indique le type de message ou de règle, et enfin la suite spécifie une chaîne de caractères quelconques, de taille fixée (255 octets ici).

Les différents types de paquets possible sont :

- fin des règles
- acquittement
- notification d'erreur
- règle iptables
- règle iptables optionnelle
- règle ip6tables
- règle ip6tables optionnelle.

### 2.4.4 Interface

L'interface choisie pour l'administrateur (pour la maquette) est une interface en ligne de commande, sous forme d'un exécutable qui va lire sur son entrée standard une série de règles de filtrage (une par ligne), dont la syntaxe est la même que celle que la commande *iptables*.

Le premier mot de la ligne spécifie le type de règle :

- règle iptables : *iptables*,
- règle ip6tables : *ip6tables*,
- règle iptables pouvant échouer : *iptables-fail*,
- règle ip6tables pouvant échouer : *ip6tables-fail*.

Les paramètres de ligne de commande de l'exécutable spécifient la machine à administrer (paramètres de la connection sécurisée, adresse du serveur à administrer).

Une fois les règles lues, elles sont encodées sous forme de messages, et transmises au serveur (la machine à administrer). Le serveur lit toutes les règles, vérifie qu'elles ne contiennement que des caractères autorisés (afin d'éviter l'exploitation d'une éventuelle faille de l'utilitaire iptables), et attend la notification de fin de transmission du client. Le test de validité des règles ne peut être très strict, pour ne pas nuire à l'extensibilité d'iptables sus-citée. Une fois la notification de fin de transmission reçue, il envoie un message de confirmation indiquant qu'il a bien reçu toutes les règles, et qu'elles sont a priori correctes, et sauvegarde l'état du pare-feu.

Il applique alors les règles, une par une, en vérifiant le cas échéant qu'elles ont été appliquées correctement (en validant le code de retour du programme invoqué). Si tout se passe correctement, il envoie alors un second message de confirmation et met fin à la connection, sinon il restaure le pare-feu et envoie au client un message d'erreur contenant le numéro de la règle ayant provoqué l'erreur. Le client indique à l'utilisateur le succès ou l'échec de la transaction, au moyen de messages affichés sur la console.

Ce prototype permet ainsi d'administrer le pare-feu d'une machine distante, ce qui était le but recherché. Pour pallier à l'austérité (selon certains) de l'interface en mode texte, un petit programme CGI a été écrit (dans le langage de script Ruby, qui permet en très peu de lignes de code de créer un programme très complet). Il permet ainsi d'accéder à la puissance de la ligne de commande au moyen d'une interface conviviale, en html. Il permet également d'utiliser le client pour administrer plusieurs machines sans avoir à écrire de script shell, ce qui peut présenter un avantage pour certains.

Cette interface permet de spécifier l'adresse de chaque machine à contacter, le secret à utiliser pour s'authentifier auprès de celle-ci, ainsi que l'ensemble de règles à transmettre. Il est possible d'ajouter des boutons permettant de spécifier un ensemble-type de règles : par exemple il est possible actuellement de limiter toutes les connexions sortantes sur le port 25 TCP vers des IPs autres que celles d'un serveur spécifié.

### 2.4.5 Protection

Comme le programme contrôle le pare-feu, et écoute lui-même sur une interface réseau, il semble logique d'essayer de faire en sorte qu'il soit impossible de bloquer les connections qui lui sont destinées.

Cette tâche est assez difficile, étant donné la versatilité du pare-feu : il existe de nombreuses manières d'interdire la connection sur un certain port. On peut par exemple ignorer les

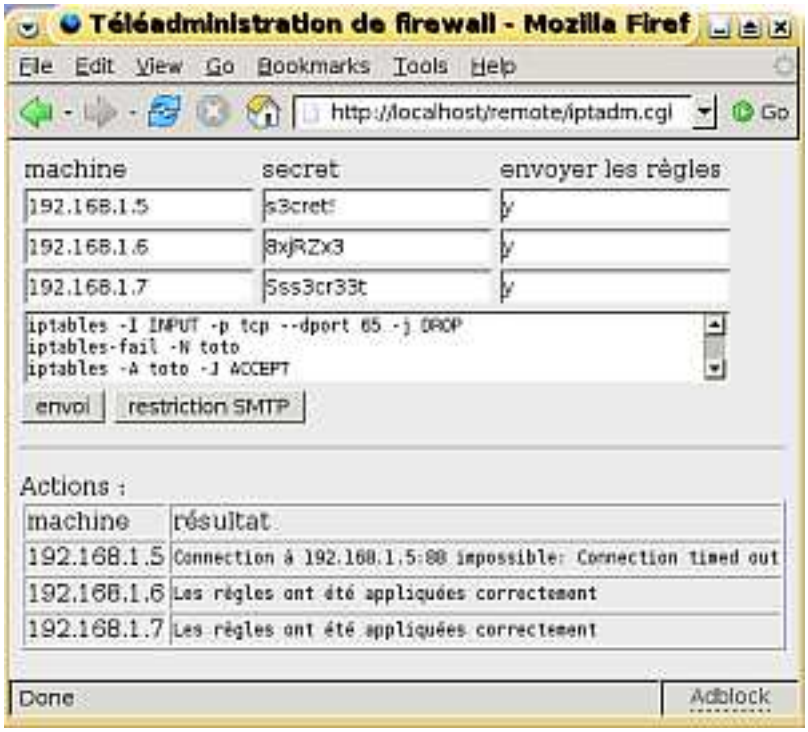



FIG. 2.7 – l'interface du CGI

paquets d'administration, ou les modifier de plusieurs façons de manière à ce qu'ils ne soient pas délivrés correctement. De plus, le programme n'essaye pas (il n'en serait de toute façon pas capable) de comprendre les règles qu'il doit appliquer ; il n'est donc pas capable de déterminer parmi les commandes reçues celles qui sont potentiellement nuisibles pour lui-même.

Il reste cependant une possiblité au programme pour se protéger : comme il applique lui-même les règles, il lui suffit d'appliquer toutes les règles demandées ; puis d'insérer en tête de chacune des chaînes traversées par les paquets d'administration une règle pour les laisser passer.

Il faut pour cela être en mesure d'identifier de manière assez précise ces paquets, de manière à ne pas laisser passer trop de trafic (ce qui serait contraire à son but premier). Comme le démon connait (d'après sa configuration) le port sur lequel il écoute, il peut utiliser cette connaissance pour construire la règle. Afin d'être plus précis, une option de configuration a été ajoutée, qui permet de spécifier une interface réseau d'administration. Avec ces deux informations, le serveur construit une règle qui autorise le passage des paquets arrivant sur cette interface, utilisant le protocole TCP, et à destination du port du démon, qu'il ajoute dans la chaîne INPUT de la table filter.

Cette règle est uniquement présente dans un but d'exemple, et devra être modifiée dans le cadre d'une utilisation réelle, pour être plus précise : il faudrait par exemple spécifier une (ou plusieurs) adresse IP d'administration, qui seraient autorisées à se connecter au serveur. Pour être complètement efficace, il faudra également rajouter une règle du même type dans chacune des chaînes traversées : mangle :PREROUTING, nat :PREROUTING et mangle :INPUT.

On pourrait également penser, dans le cadre de la protection, à utiliser la démarche inverse, en restreignant l'accès au serveur à une certaine liste d'adresses, et en cachant la présence du démon au reste du réseau.

La principale faiblesse de cette protection est que le démon n'est potentiellement pas le seul programme capable de modifier le pare-feu : l'utilisateur peut utiliser la ligne de commande et invoquer l'utilitaire iptables lui-même, ou bien disposer d'une autre interface de configuration, distincte. Une protection totale doit être globale : il faudrait que toutes les possibilités de modification des règles de filtrages prennent en compte la protection du serveur de téléadministration (ainsi que celle de leur propre interface).

## 2.5 Honeypots

### 2.5.1 Introduction

Un honeypot est un système informatique dédié à la détection d'anomalies réseau.

Il fonctionne à la manière d'un leurre. Cette fonction est accomplie en revendiquant une certaine adresse IP sur un réseau, telle que cette adresse soit normalement inutilisée. Il est également possible, au moyen de techniques bas niveau (ethernet), de revendiquer toute une plage d'adresses. Le honeypot peut également, dans une configuration particulière, travailler sur une adresse réellement utilisée, dans ce cas il n'interviendra que pour les tentatives de connexion sur un port non utilisé par le serveur légitime.

Pour toutes les adresses dont il est en charge, le honeypot va faire en sorte de simuler le comportement d'un certain équipement (qui peut aller du PC sous Windows au routeur Cisco 4500 série 2), de manière très fine : émulation complète de la pile IP du matériel imité, de manière à tromper les différents outils disponibles ayant des capacités d'OS fingerprinting, comme le scanner *nmap* [11]. Le honeypot va également émuler les différents services que l'on peut retrouver sur ces systèmes. Ce faisant, il loggue toute activité qu'il reçoit. Comme l'adresse n'est pas censé être utilisée, cette activité est de manière certaine une anomalie. Cela peut aller du scan du réseau par un attaquant à une tentative de propagation d'un vers quelconque.

Il existe différents types de honeypots, que l'on classe selon leur manière de réagir à une tentative de connexion :

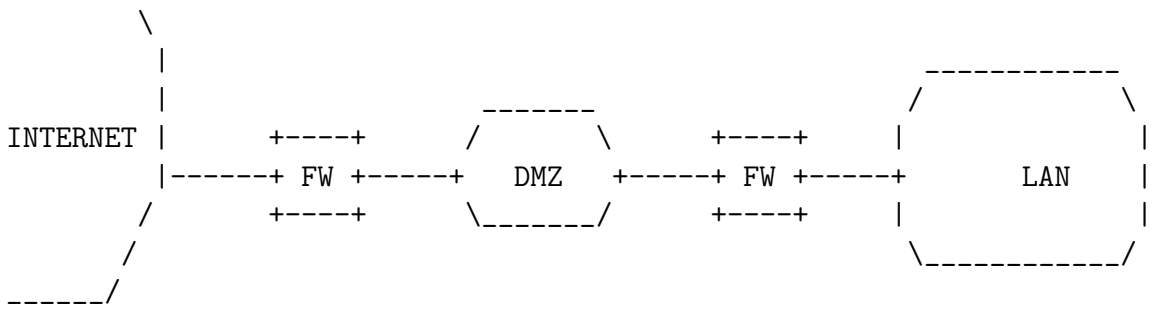
- Les honeypots *faible interaction*. Ceux-ci se contentent de lire sur le réseau les paquets qui leur sont destinés, mais n'émettent que le strict nécessaire pour simuler leur hote. En particulier ils n'émulent pas les différents services que l'on peut trouver sur la machine imitée, mais se contentent d'établir la session niveau TCP par exemple, de logguer le premier paquet recu, et de clore la session.
- Les honeypots *forte interaction*. Ceux-ci imitent de manière forte les services que l'on peut trouver sur la machine imitée : par exemple si un serveur web est émulé, une connexion sur celui-ci révèlera un vrai mini-site, il peut aller jusqu'a fournir un shell à un attaquant, de manière à étudier le comportement de celui-ci lorsqu'il atteint une machine. La configuration d'un tel système est assez complexe, car il ne faut pas permettre par exemple à un attaquant de lancer des attaques vers d'autres machines à partir du honeypot (ceci poserait des problèmes légaux au possesseur du honeypot).

Un honeypot est un outil précieux pour déterminer l'état d'un réseau à un moment donné, d'autant qu'on peut le placer à différents endroits dans l'architecture réseau : avant le pare-feu, dans la DMZ, ou sur le réseau interne, ce qui permet de savoir précisément quelles menaces ciblent quelles parties du réseau.

Il serait intéressant de pouvoir déployer rapidement plusieurs honeypots en des endroits distincts du réseau. Les intégrer sur des systèmes embarqués permettrait justement cette facilité, en étant moins cher et moins encombrant qu'un ordinateur.

### 2.5.2 Prototype

Le prototype réalisé consiste en l'application honeyd, couplée avec l'utilitaire snort, un ensemble de scripts de configuration, un petit programme chargé de logguer toute activité réseau, et un utilitaire créé de toutes pièces permettant l'administration distante du honeypot




FW: pare-feu
DMZ: zone contenant les serveurs accessibles depuis internet
LAN: réseau interne

FIG. 2.8 – Architecture réseau classique

par le biais d'un interpréteur de commande ou de transfert de fichier. Le honeypot réalisé sera de type faible interaction.

La cross-compilation de l'application honeyd en elle-même ne pose pas de problèmes, cependant les librairies dont elle dépend présentent quelques difficultés. En général il s'agit du script *configure* qui veut exécuter un programme afin de déterminer certaines caractéristiques du système, ce qui n'est bien entendu pas possible lors d'une cross-compilation : il a donc fallu voir quels tests le script voulait effectuer, écrire un programme qui les effectue et en affiche le résultat, et enfin patcher le script pour lui fournir lesdits résultats (l'architecture d'*autoconf* ne permet pas de fournir simplement les résultats d'un test particulier, il faut obligatoirement modifier le script directement).

Honeyd est extensible très facilement au moyen de scripts *python*, mais cette fonctionnalité a du être écartée du prototype, car un interpréteur python (ou n'importe quel autre langage de script) serait certainement trop lourd pour un système embarqué, d'autant plus que cette fonctionnalité n'est pas utilisée par le prototype. Elle est plutot utile pour les configurations à haute interaction, où elle permet d'émuler un service complexe donné par un script.

L'utilitaire Snort est un logiciel implémentant la détection de tentatives d'intrusion (IDS). Il accomplit ceci en comparant diverses caractéristiques des paquets reçus avec une base de données de signatures d'attaques connues, mise à jour très régulièrement. Il peut effectuer cette comparaison soit à partir d'un fichier contenant tous les paquets reçus, soit en écoutant directement l'interface réseau et en faisant l'analyse en temps réel. Il est également possible de l'utiliser en tant que module du pare-feu netfilter, où il peut sélectionner une cible ou une autre pour un paquet suivant sa nocivité détectée, ce qui permet par exemple d'interdire le passage aux paquets détectés comme étant une attaque potentielle.

Le portage de cet utilitaire présente le même type de difficultés mineures que les dépendances de honeyd (d'ailleurs snort partage certaines de ces dépendances, comme la librairie de compression *zlib*).

L'utilisation de snort est utile dans la phase de qualification des différents paquets reçus :

on peut en effet classer ceux-ci suivant différentes catégories, allant du paquet essayant d'exploiter une faille très ancienne et connue, vers le paquet relatif à une faille très récente, jusqu'au paquet inconnu, qui devient alors très intéressant, car il comporte peut-être une attaque sur une faille inconnue du public (ce type de failles est souvent appelé *zero-day*).

Snort est cependant un logiciel complexe, qui effectue un traitement important pour chaque paquet. Étant donné la (relative) faible puissance de calcul d'un Wrt54g, il lui est impossible de traiter tous les paquets reçus lors d'une utilisation intensive du réseau (comme un *flood* de la part d'un attaquant, lors duquel l'attaquant innonde la cible de paquets, autant que le permet le débit de sa connexion réseau). Il peut donc sembler plus approprié de passer tous les paquets reçus par le honeypot par snort de manière offline, c'est à dire pas en temps réel, mais en décalé, grace aux fichiers de logs d'activité réseau.

De fait, est apparu qu'il est donc plus simple de déporter l'utilisation de snort vers la machine d'administration, car elle est d'une part beaucoup plus puissante, de plus cela évite à chaque honeypot de devoir se maintenir à jour individuellement. Le logiciel peut tout de même être utilisé localement, si par exemple un honeypot est déployé seul, tout en étant conscient des limitations de celui-ci sur une telle plateforme.

L'enregistrement du trafic réseau vu par le honeypot est réalisé par un petit utilitaire nommé net2pcap, qui est volontairement très simple. Il est composé d'un simple fichier source, et n'a aucune dépendance, aucun travail n'est nécessaire pour le cross-compiler. Il est capable d'enregistrer toute trame ethernet recue par une ou plusieurs interfaces réseau, et de les enregistrer dans un ficher sous un format standard (celui défini par l'utilitaire *tcpdump*), que tous les programmes libres travaillant sur ce type de données sont capable de comprendre (en particulier snort).

Ce programme va donc enregistrer dans un fichier toutes les trames reçues par le honeypot, et ce fichier sera rappatrié à une certaine fréquence (tous les jours par exemple) vers la machine qui sera chargée de la supervision du groupe dans lequel se trouve le honeypot. Ce contrôleur sera alors en mesure d'analyser ces traces et d'en déduire les différentes menaces pesant sur les différentes parties du réseau.

Le logiciel utilisé pour administrer le honeypot utilise également la librairie de cryptographie développée au début du stage. Comme le logiciel de téléadministration de pare-feu, il consiste en un serveur TCP qui écoute sur un port donné de la machine à administrer, et en un client en ligne de commande qui s'y connecte et lui envoie les messages appropriés. Il comporte trois modes de fonctionnement :

- transfert de fichier montant,
- transfert de fichier descendant,
- exécution de commande interactive.

Afin de pouvoir fonctionner dans ces trois modes, un protocole très basique a été développé.

Le premier octet du premier paquet spécifie le mode de fonctionnement, selon sa valeur parmi 3 définies en dur dans les programmes.

Pour les applications de transfert de fichier, la syntaxe est alors la même :

`MOD` indique le mode de fonctionnement (upload : 3, download : 2 - sur 1 octet) `NSZ` est la taille du nom du fichier (4 octets, big endian) `nom` indique le nom du fichier sur le serveur (à lire ou à écrire) `FSZ` est la taille du fichier (4 octets, big endian) `contenu` est le contenu du fichier

Après transmission de l'intégralité du fichier, l'émetteur attend la confirmation du re-

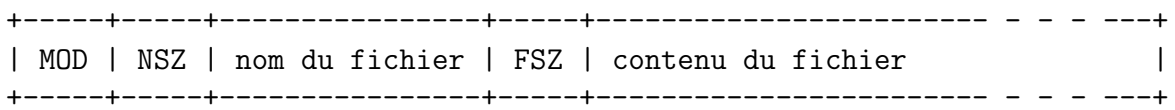



FIG. 2.9 – Format de message pour le transfert de fichier

ceveur, qui consiste en un simple byte d'une valeur prédéfinie. A la confirmation (ou à la fin de la réception en cas d'upload), le client informe l'utilisateur du succès ou non de la transmission.

Le comportement est très différent pour l'exécution d'une commande interactive. Dans ce cas le message initial suit le format suivant :

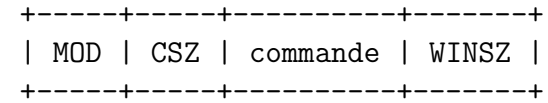



FIG. 2.10 – Format de message pour l'exécution de commande

`MOD` indique "exécution de commande" (valeur 1) `CSZ` indique la taille de la ligne de commande (1 octet) `commande` est la commande à exécuter (avec ses arguments éventuels) `WINSZ` contient la taille de la console coté client (2x2 octets, big endian)

Le client envoie au serveur la chaîne complète à exécuter (commande et arguments), ainsi que la taille actuelle de la console (largeur et hauteur). Pour une connexion impliquant l'exécution interactive d'une commande, les deux processus modifient les paramètres TCP de la connexion afin de désactiver le mécanisme de cache du noyau (qui permet de n'envoyer qu'un paquet de 5 octets s'il recoit 5 *write* successifs d'un octet), ce qui améliore l'interactivité perçue de la commande (au moyen de l'appel système `setsockopt()` avec l'argument `TCP_NODELAY`). Il modifie également les paramètres de sa console, pour la passer en mode de contrôle manuel (mode *raw*).

A la réception de ce message, le serveur crée un terminal virtuel (pty), de dimensions égales à celui du client, puis exécute le programme demandé dans ce terminal. Il se comporte ensuite comme un tunnel, relayant les entrées/sorties du programme vers le client et vice-versa.

De la même manière, le client se contente ensuite de transmettre son entrée standard au serveur, et d'afficher ce que celui-ci lui envoie.

A la terminaison du programme, le serveur envoie au client une chaîne au client indiquant la valeur de retour ainsi que la raison de la fin du programme (terminaison normale ou mort du processus à cause d'un signal), qu'il affiche.

Les honeypots sont généralement administrés localement. Ici, l'administration passe par le réseau, ce qui nécessite quelques modifications du comportement de l'ordinateur, afin qu'un attaquant éventuel ne voit pas le serveur d'administration par exemple, ou que le honeypot ne détecte les connexions d'administration comme des paquets anormaux du point de vue du réseau.

Dans cette configuration, le démon faisant office de honeypot (ainsi que l'utilitaire d'archivage des paquets reçus) est lancé sur une interface réseau virtuelle. On met alors en place une série

de règles de filtrage, qui ont pour but de laisser la machine accessible par la machine d'administration, mais qui redirige tout autre trafic vers l'interface virtuelle, de manière à être vu par le honeypot.

Pour cela on insère une simple règle ACCEPT sur les paquets de protocole TCP, venant de la machine d'administration, à destination du port d'administration dans la chaîne PREROUTING de la table nat, et une autre règle qui redirige tout le reste vers l'ip de l'interface virtuelle où sont installés les services de honeypot, au moyen d'une règle de DNAT. Ce système bénéficie de l'utilisation d'un serveur TCP simple pour l'administration, plutot qu'un serveur de shell couplé avec un serveur ftp par exemple, car celui-ci pourrait utiliser des ports dynamiquement, alors qu'ici il suffit d'ouvrir un seul port qui fournit toutes les fonctionnalités utiles.

Pour le déploiement de ce type d'équipement, j'ai développé un script, qui permet de transformer simplement un Wrt54g neuf (c'est à dire avec son système d'exploitation d'origine) en une machine utilisable en honeypot. Ce script est écrit en langage Ruby.

Il utilise l'interface web d'origine pour modifier la configuration du système de manière automatique, en mettant en place une variable qui autorise le flashage du système à chaque démarrage, pendant quelques secondes après la mise sous tension. Une fois cette mesure de sécurité validée, il utilise la procédure standard d'upgrade pour installer un système OpenWrt, et se connecte ensuite au port telnet de celui-ci pour installer le reste des utilitaires et désactiver et effacer les services non utilisés par le honeypot (comme les serveurs web, dns, telnet, etc..). Il uploade également tous les fichiers nécessaires au démarrage de la machine en mode honeypot (script de boot qui met en place les règles de filtrage, fichiers de configuration de honeyd, script de lancement des différents utilitaires).

## 2.6 Filtrage viral des mails

Les virus et vers sont un fléau extrêmement répandu parmi tous les ordinateurs connectés à Internet aujourd'hui. Ils sont cependant bien connus, et leur détection est généralement aisée, pour peu que l'on se donne la peine de les chercher.

Ils utilisent principalement deux vecteurs de propagation :

– la contamination directe (active),

– la contamination par intervention de l'utilisateur (passive).

La contamination active consiste, à partir d'une machine infectée, à essayer de se connecter à une autre machine sur internet (en tentant une addresse ip au hasard), à regarder si cette machine utilise un certain service vulnérable pour laquelle le vers possède un mécanisme d'exploitation, et à exploiter cette faille afin de se propager de manière autonome sur cette nouvelle machine. Ce type de propagation est généralement écarté par le simple ajout d'un routeur devant la machine cible, qui interdit les connections entrantes (du point de vue TCP ou UDP), et donc interdit l'accès au service vulnérable par le vers.

La propagation dite passive prend principalement deux formes. Par email : il s'agit d'envoyer à une série de destinataires (soit connus sous forme d'une liste, soit tentée au hasard à la découverte d'un relai SMTP) un mail, contenant le virus lui-même en pièce jointe, avec en corps de mail un message quelconque incitant l'éventuel lecteur du mail à ouvrire ladite pièce jointe et à l'exécuter. De cette manière le virus peut accéder aux ressources de l'ordinateur, et éventuellement lire le carnet d'adresses de celui-ci afin de continuer sa propagation. Une solution à ce problème va être analysée dans la suite.

La deuxième forme principale de propagation passive des virus consiste à se trouver dans un fichier sur un serveur web, et à inciter l'utilisateur à nouveau à télécharger le virus et à l'exécuter (en le faisant passer pour un programme dont l'utilisateur pourrait avoir envie), soit en utilisant des techniques de spam par mail, soit en construisant un site factice, construit de telle sorte qu'une recherche dans un des nombreux moteurs de recherche qui existent renvoie cette page avec un bon classement. Ce type d'infection peut être détectée en insérant une phase de recherche automatique de virus dans les flux http liés à la machine à protéger. Ceci sera l'objet de la section suivante.

### 2.6.1 Prototype

De nombreuses solutions libres pour le filtrage des mails ont été développées pour ordinateur personnel. Elles peuvent intervenir à différentes étapes de la consultation ou de l'envoi de mails : soit à la volée lors du téléchargement du mail avant sa visualisation, au moyen de proxies pour les protocoles POP3 ou IMAP, soit intégrées au client de consultation des mails, soit au moyen de proxies pour le protocole d'émission de mails, SMTP.

Le système embarqué de protection étant détaché du poste utilisateur, le champ d'investigation a été réduit à l'intégration d'un antivirus dans les différents proxies. Toutes les architectures existantes dans le domaine des logiciels libres reposent sur l'utilisation du seul antivirus libre existant, à savoir Clamav.

J'ai donc téléchargé le logiciel et l'ai cross-compilé afin de réaliser différents tests, notamment sur l'adaptabilité à un système embarqué. La cross-compilation de celui-ci à généré quelques problèmes. En particulier le logiciel utilise un programme annexe, qui est compilé avant le logiciel principal, et qui est utilisé pour générer un fichier de données nécessaire à la compilation du programme principal. Cependant les concepteurs n'ont pas prévu le cadre

de la cross-compilation : il a donc fallu recompiler ce programme ainsi que ses dépendances pour exécution sur la machine de développement, en faisant plusieurs modifications dans les différents fichiers *Makefile* du programme.

Ensuite, une fois la compilation effectuée, les exécutables générés sont cachés dans des sous-répertoires obscurs, et les "exécutables" visibles par l'utilisateur ne sont en fait que des scripts qui appelent les véritables programmes après avoir effectués quelques modifications, sur des variables d'environnement par exemple. Il a donc fallu rendre les programmes exécutables exécutables directement.

### 2.6.2 Problèmes rencontrés

Clamav est livré avec un ensemble de fichiers de tests, dont les signatures doivent être reconnues par l'antivirus comme des fichiers nocifs. Ils servent à valider la compilation et l'installation du logiciel. Cependant lors de l'exécution du programme sur ces fichiers de tests sur un Wrt54g, au bout de quelques secondes le système rebootait, sans que le test de viralité n'ait abouti.

L'environnement Wrt54g étant minimal, il ne dispose pas de tous les outils d'investigation disponibles sur un système standard, comme *strace* ou *top* (il existe un version de top, cependant celle-ci est extrèmement limitée en fonctionnalités, et ne permet que de lister les processus s'exécutant au moment où on le lance).

J'ai donc développé un petit outil, qui permet d'étudier l'évolution de l'utilisation de la mémoire d'un processus cible, en lui envoyant périodiquement le signal `SIGSTOP`, qui a pour effet de stopper le processus qui le reçoit, jusqu'a réception du signal `SIGCONT` qui relance son exécution. Lorsque le processus est stoppé, l'outil va afficher le contenu du pseudo-fichier */proc/pid/status*, qui affiche entre autres informations les statistiques sur l'utilisation mémoire du processus. Il relance ensuite le processus pendant quelques instants, et répète l'opération.

L'utilisation de cet outil révèle que le programme antivirus consomme toute la mémoire disponible sur le système, et celui-ci ne résiste pas à ce manque de ressources et plante ou reboote immédiatement. Un test sur ordinateur montre que le processus utilise environ 12 mégaoctets de mémoire, ce qui doit être sensiblement la même chose sur le système embarqué. Or sur le Wrt54g, le simple fait de charger le système d'exploitation de base utilise plusieurs méga-octets, donc la mémoire physique disponible est insuffisante pour permettre l'exécution dudit processus. Comme de plus sur ce type de système les services utilisent tous le compte utilisateur *root*, les mécanismes de protection du système d'exploitation sont inefficaces et la stabilité du système entier est mise en cause, ce qui explique les plantages constatés. Cela ne change pas le fait que le programme est inutilisable, meme sous un autre compte utilisateur.

Le manque de moyens d'investigation pour le système en cours de fonctionnement m'a poussé à écrire un petit programme qui utilise l'interface *ptrace* de débuggage du noyau, qui fournit des fonctionnalités similaires à celles de strace : il permet de suivre en direct tous les appels systèmes effectués par ou plusieurs processus, en affichant éventuellement leur valeur de retour ou leurs arguments. Ceci m'a permis d'utiliser véritablement l'appel `ptrace()`, ce qui m'intéressait mais que je n'avais jamais eu l'occasion de faire. Cela a également été l'occasion de découvrir de manière beaucoup plus fine l'architecture MIPS (c'est la famille de microprocesseurs utilisés sur Wrt54g), car l'appel `ptrace()` oblige à manipuler des structures très dépendantes du processeur, comme les registres. Cela fournit un utilitaire très pratique, et je sais qu'il a été réutilisé pour d'autres tâches au sein de France Télécom R&D, je suis

donc content de cette réalisation.

Son utilisation sur le programme antivirus montre que la mémoire utilisée est due au chargement de la base de données de virus. Il serait donc probablement possible de modifier le programme afin qu'il utilise des appels moins gourmands en mémoire, comme `mmap()`, pour lire ces fichiers, mais je n'ai pas eu le temps de me plonger dans le code source de cette application (qui n'est pas petite).

Il apparait donc que la réalisation de ce type de prototypes, impliquant de scanner des fichiers ou des flux à la recherche de virus sur le système embarqué n'est pas réalisable en pratique, compte tenu des limitations actuelles du programme antivirus.

J'ai eu accès par la suite à un matériel supérieur, le Wrt54gs, qui dispose lui de 32Mo de mémoire, sur celui-ci l'antivirus peut s'exécuter, et détecte les virus dans les fichiers de tests. Cependant il n'est pas concevable de faire tourner sur un système embarqué en situation réelle un programme capable de prendre à lui seul plus de la moitié de la mémoire disponible, sachant qu'il sera également couplé à un proxy qui peut être lourd, et sachant qu'il faudra également stocker en mémoire le mail considéré, afin de ne pas consommer trop vite le nombre limité de cycles d'écriture sur la mémoire flash servant de stockage permanent au système. De plus la base de données de l'antivirus est nécéssairement de plus en plus grosse, au fur et à mesure de l'apparition de nouveaux virus ou de nouvelles variantes.

Ce prototype n'a donc pas pu être réalisé.

## 2.7 Filtrage d'URL et de contenu

Comme on vient de le voir, le filtrage de flux http, tout du moins dans l'optique de filtrer les virus, semble impossible si le traitement doit être fait localement.

Heureusement, il existe un protocole qui nous permettrait de faire exactement ce type de filtrage, et bien plus encore (comme le filtrage de contenu de type contrôle parental, le filtrage de certains scripts des pages html visionnées...). Ce protocole est normalisé sous le nom d'Internet Content Adaptation Protocol. Il n'existe pas à ma connaissance d'autre système existant permettant ce type d'actions.

### 2.7.1 Le protocole ICAP

Le but de ce protocole est de fournir un système permettant à un client ICAP et à un serveur ICAP de communiquer des données, le client envoyant des requêtes au serveur qui peut lui renvoyer modifiée, telle quelle, ou lui signifier de rediriger l'utilisateur vers une page d'erreur. Il a été conçu pour être utilisé avec le protocole HTTP. Les schémas suivants décrivent le fonctionnement du protocole HTTP par l'intermédiaire d'un proxy, suivant que celui-ci est configuré ou non pour utiliser le protocole ICAP.

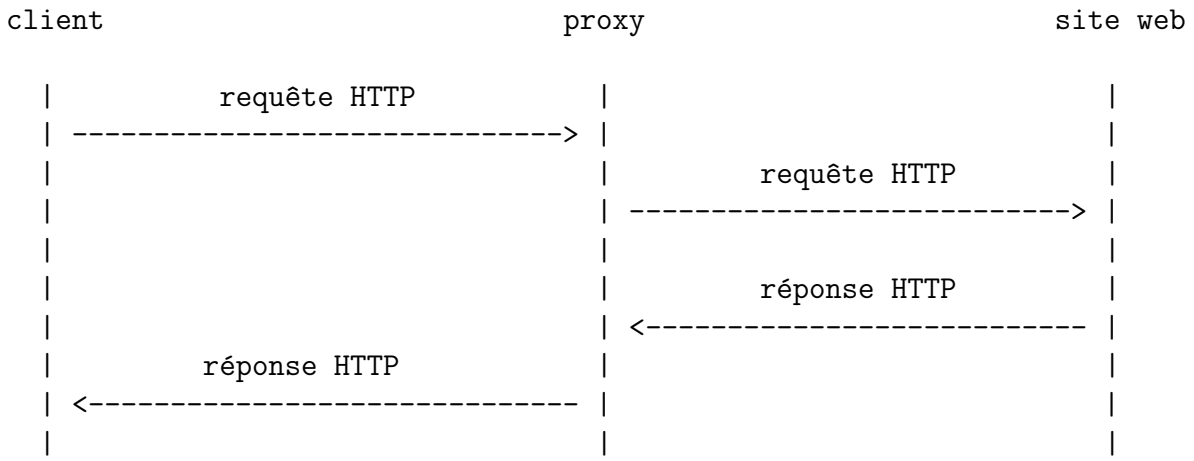



Fig. 2.11 – Une requête HTTP normale

Dans une configuration normale, le proxy se contente de relayer la requête du client vers le serveur qu'il demande, en implémentant éventuellement un mécanisme de cache.

Dans le fonctionnement avec un proxy ICAP, le proxy peut (selon sa configuration) faire une requête ICAP de validation de la requête du client à son serveur ICAP. Celà permet par exemple au serveur de regarder si la requête tente d'accéder à un site connu pour héberger une catégorie de contenu interdite par la configuration du client au moyen d'une blacklist par exemple. Le serveur dispose alors de 3 options :

– il peut notifier au proxy que la requête est autorisée
– il peut renvoyer au proxy la requête modifiée
– il peut renvoyer au proxy une réponse HTTP

Dans le premier cas, le proxy se comporte alors normalement et effectue la requête auprès du serveur demandé.

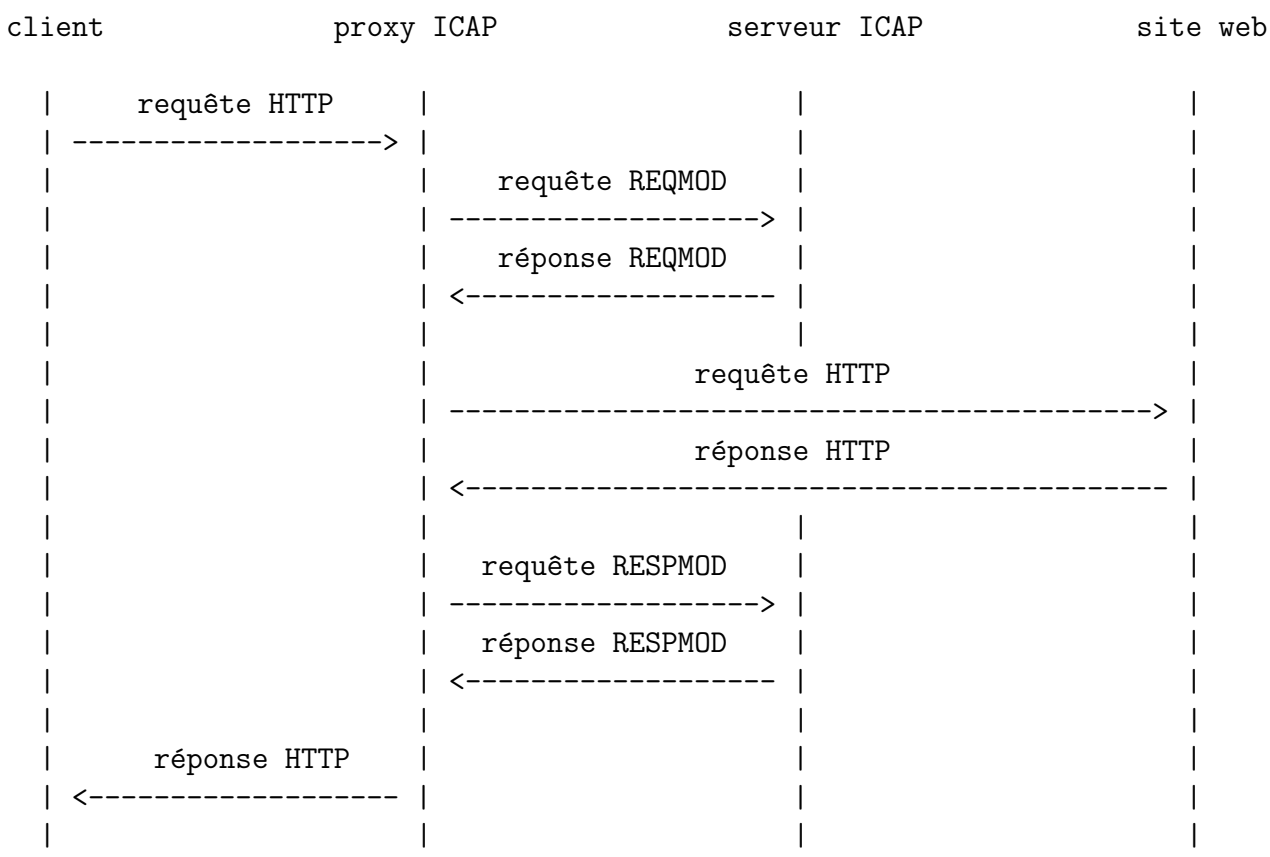



FIG. 2.12 – Une requête HTTP normale avec un proxy ICAP

Dans le second cas, le proxy considère l'url renvoyée par le serveur ICAP comme si c'était celle que le client avait demandée. Elle peut par exemple rediriger vers une page web indiquant que le site est listé comme ne correspondant pas à la configuration du client, ou éventuellement vers un autre site quelconque choisi par le serveur comme plus adapté pour le client.

Dans le dernier cas, le proxy renvoie directement au client la réponse reçue du serveur ICAP comme si c'était la réponse reçue à l'url demandée par le client. Cette réponse peut être une page d'erreur indiquant que l'url demandée est bloquée, la raison, ainsi que tout autre information nécessaire.

Quand le proxy fait la requête au site web, il peut suivant sa configuration renvoyer le résultat reçu au serveur ICAP pour validation. C'est à cette phase par exemple que le filtrage antivirus pourrait avoir lieu. Le serveur a alors 2 choix : soit il renvoie le contenu tel quel, soit il le renvoie modifié. Par exemple si un virus est détecte, il remplacera la page par une page standard indiquant la présence du virus. Cette réponse est alors transmise au client comme si c'était la réponse à sa requête initiale.

En pratique le protocole implémente quelques autres fonctionnalités utiles, comme par exemple la possibilité pendant la phase RESPMOD de ne transmettre au serveur qu'une partie du document, afin d'éviter les communications inutiles (si par exemple le serveur veut ne scanner que les fichiers exécutables, il lui suffit de voir les premiers octets d'un fichier pour déterminer si il a besoin de la suite ou non).

### 2.7.2 Prototype

Un module pour le proxy libre Squid a été développé afin d'apporter des capacités ICAP au proxy. J'ai donc réalisé le travail de cross-compilation, qui se passe assez bien, l'habitude aidant.

C'est alors que deux problèmes majeurs sont apparus. Tout d'abord, il n'existe pas d'implémentation libre d'un serveur ICAP, ce qui est plutôt gênant pour effectuer des tests, et encore plus pour une maquette. Le second problème est que le module squid a vu son développement arrêté depuis fin 2003, alors qu'il n'est pas complêtement finalisé : plusieurs fonctionnalités prévues dans le protocole n'ont pas été implémentées.

J'ai donc laissé le prototype dans cet état, c'est à dire juste compilé, sans pouvoir seulement le tester. La fin du stage approchant, je n'ai pu travailler sur ce prototype, qui constitue pourtant une piste très intéressante.

### 2.7.3 Travail nécessaire

Pour rendre ce prototype opérationnel, il faudrait tout d'abord terminer le codage du module squid. D'après ce que j'en ai vu, la plupart des fonctionnalités sont déjà implémentées, donc cela ne devrait pas prendre beaucoup de temps.

La deuxième partie consiste en le codage d'un serveur ICAP. Il existe plusieurs projets opensource ayant ce but, cependant la plupart sont abandonnés de leurs développeurs, dans un état peu avancé, donc de ce côté il y aurait certainement besoin de beaucoup plus de travail. Le serveur est en plus le goulôt d'étranglement au niveau du réseau, car il devra probablement gérer un grand nombre de clients, il faudrait donc le coder avec soin.

Ces deux tâches accomplies, il serait alors possible de réaliser une maquette.

De là à passer à une utilisation réelle, il faudrait faire une grosse étude sur les performances que cela implique, à la fois au niveau du client (la consultation internet doit rester agréable et rapide) qu'au niveau du serveur (combien un serveur peut-il supporter de clients...). Il faudrait également concevoir un ensemble de configurations pour les clients, adapté aux différents types d'ordinateurs à protéger : système linux / windows (par rapport au filtrage de virus), configuration pour la protection des mineurs (contrôle parental), concevoir une interface de configuration par client.

Le plus complexe sera probablement la configuration des serveurs : leur dire quel type de fichiers scanner, comment les scanner, quelles réponses renvoyer en fonction du résultat ; comment faire le contrôle pour l'interdiction de certaines catégories de sites internet ; combien de serveurs déployer par rapport aux clients, ....

Cela semble une tâche de grande envergure.

# Chapitre 3

# Conclusion

Les systèmes embarqués seront dans les années à venir de plus en plus présent dans notre vie de tous les jours, à tous les niveaux. Aussi il fut extrêmement formateur de pouvoir travailler sur ce type d'équipement, qui consistue pour moi monde totalement différent de l'ordinateur classique : nouveau système d'exploitation, nouvelle architecture processeur.

Il n'existe dans ce domaine pas beaucoup d'initiatives libres, ce qui est gênant pour entrer dans ce monde, aussi ce stage fut une bonne opportunité pour s'y intéresser.

Il m'a également permi d'intégrer une entreprise d'avenir, qui embrasse un large panel technologique, et dont le pôle de recherche et développement est très actif, dans tous les domaines de pointe de l'informatique.

Je regrette seulement de n'avoir pas pu réaliser certains prototypes, qui auraient certainement pu trouver une utilité, mais le temps à manqué.

## 3.1 Remerciements

Je remercie chaque personne du département, qu'il est très agréable d'intégrer. Plus particulièrement Julien Tinnès pour m'avoir aidé dans ma recherche de stage et d'emploi ; Jean-Marc Hospital pour avoir su éclairer les jours sombres de son optimisme inébranlable ; l'ensemble des stagiaires, dont messieurs Andréotti, Sanchez et Jaubert, pour leur bonne humeur ; et bien évidemment Fabrice Stevens, pour cet excellent stage.

# Chapitre 4

# Annexe

## 4.1 Implémentation de la librairie cryptographique

La librairie à été écrite en langage C, elle dépend uniquement de la libtomcrypt et de la librairie standard.

La librairie cryptographique implémente une surcouche de sécurité, destinée à être utilisée de manière transparente à la place des appels systèmes *read()* et *write()*, après une séquence d'initialisation et d'établissement de session. Le fonctionnement interne des substituts à *read()* et à *write()* oblige également à quelques différences d'interface par rapport aux appels standards, qui vont être listées dans la suite.

### 4.1.1 Interface

Toutes les fonctions fournies par la librairie sont préfixées de `crypto_` afin de ne pas interférer avec les éventuelles fonctions du programme hôte. Ce préfixe comme peut être vu comme un spécificateur d'espace de noms. Les fonctions renvoyant une valeur renvoient une valeur positive ou nulle ne cas de succès, et renvoient -1 en cas d'erreur, en définissant *errno* de manière appropriée. Deux valeurs spécifiques sont utilisées pour cette variable : `EIO` indique une erreur interne à la librairie *libtomcrypt*, alors que la valeur `EILSEQ` indique une erreur dans le protocole (hash/clé invalide, ...). Dans ce dernier cas, le descripteur de session associé au descripteur incriminé est libéré, puisque le descripteur n'est plus utilisable (AES en mode CBC ne tolère aucune erreur).

La librairie présente une interface haut niveau, et utilisant des fonctions offertes par la *libtomcrypt*. Comme cette dernière nécessite une phase d'initialisation, le programme utilisateur de la librairie doit commencer par invoquer la fonction d'initialisation `crypto_setup()`.

Une session est mise en place en invoquant la fonction `crypto_session_setup()`, qui prend en arguments un descripteur de fichiers (fd), un buffer contenant le secret, un entier égal à la taille du secret, et un octet, “asym”. Cet octet doit être différent dans l'application courante et dans l'application avec qui la communication va être établie (de manière à éviter une attaque consistant au renvoi des données émises).

Cet appel est bloquant, il réalise l'échange Diffie-Hellman d'établissement de session, et initialise les structures internes nécessaires au chiffrement AES lié à ce descripteur de fichier.

La durée maximale d'exécution est paramétrable, grace à la fonction `crypto_set_dh_timeout()` qui prend comme argument le nombre de millisecondes à attendre avant d'abandonner les

communications réseau lors de l'échange Diffie-Hellman (le reste de la fonction n'est pas bloquant).

La taille du nombre $X$ utilisé dans l'exponentiation est également paramétrable par l'appel `crypto_set_dh_keylen()`, qui prend en argument le nombre de bits de ce nombre. La valeur minimale est 512, et la valeur recommandée est 1536 bits.

À la fin de la session, le programme doit exécuter `crypto_session_cleanup()` en lui passant en argument le descripteur de fichier, de manière à libérer les ressources utilisées par la librairie pour cette session, et à effacer les données sensibles de la mémoire (clé et IV utilisés pour l'AES).

Pour l'échange normal de données, le programmeur doit utiliser `crypto_read()` et `crypto_write()`, de la même manière qu'il utiliserait `read()` ou `write()`.

L'envoi de données est réalisé par `crypto_write()`, qui prend en arguments un descripteur de fichier, un buffer de données et la taille des données à envoyer, de la même manière que l'appel système `write()`. Lors de la première invocation de cette fonction, les données sont recopiées dans un buffer interne à la librairie et mises en forme avant chiffrement. Le bloc ainsi obtenu doit alors être envoyé en intégralité avant de pouvoir envoyer d'autres données. Lors des appels suivants, les paramètre texte et taille sont ignorés tant qu'il reste des données chiffrées non émises.

La fonction renvoie une valeur correspondant au nombre d'octets de texte clair qui ont pu être transmis lors de cet appel. Il est de 0 si le bloc n'a pu être envoyé en entier, de la taille du texte clair envoyé sinon. La fonction renvoie -1 en cas d'erreur, il faut alors consulter la valeur de la variable `errno` pour en déterminer la cause exacte.

La lecture de données et leur déchiffrement est réalisé par `crypto_read()`. Cette fonction prend comme arguments un descripteur de fichiers, un buffer ainsi que le nombre d'octets à mettre dans ce buffer. Cette fonction va lire depuis fd l'intégralité des données chiffrées, les déchiffrer et en vérifier l'intégrité, avant de les renvoyer à l'utilisateur. Il est donc possible qu'une lecture depuis fd ne renvoie pas l'intégralité du texte chiffré attendu : dans ce cas, la fonction renverra 0, ce qui est différent de la sémantique de l'appel système `read()`, pour lequel il signifie "fin du flux". Ici le 0 est une valeur normale, la fin du flux étant signalée par la valeur de retour -1 et une valeur d'errno définie à `ERRNO_READ0` (constante définie dans le header de la librairie, à la compiilation de celle-ci). En cas d'erreur, la fonction renvoie -1 et errno est défini de la manière appropriée, sinon elle renvoie le nombre d'octets copiés dans le buffer fourni.

Une situation possible est celle dans laquelle la fonction lit un certain nombre d'octets chiffrés, les déchiffre, obtient un texte de longueur $L$, alors que l'utilisateur a demandé d'en lire $L'$, tel que $L' < L$. Dans ce cas, il convient de rappeler la fonction `crypto_read()`, même si le descripteur de fichier ne semble pas avoir de données à recevoir (on peut savoir cela grace à `select()` ou `poll()` par exemple). On peut savoir si l'on se trouve dans cette situation grâce à la fonction `crypto_should_read()`, qui prend en paramètre un descripteur de fichiers et renvoie 0 ou 1 suivant la situation (0 si aucune donnée n'est disponible dans les buffers internes à la librairie).

Il existe également deux autres fonction auxiliaires, `crypto_want_write()` et `crypto_want_read()`, qui prennent en argument un descripteur de fichier et renvoient 0 ou 1 selon que la librairie à besoin d'écrire des données où d'en lire (la librairie voudra en lire si elle n'a pas pu lire

l'intégralité d'un texte chiffré, et donc n'a pas pu en vérifier l'intégrité ; et elle voudra écrire si le buffer chiffré n'a pu être émis en intégralité).

Les deux fonctions d'écriture et de lecture sont autant bloquantes que les appels systèmes correspondant dans la même situation.

Dans le cas où le programmeur aurait besoin de deux fonctions ayant la même interface pour lire sur un descripteur de fichier chiffré ou non, la librairie présente également un stub `crypto_nocrypto_read()`, qui se contente d'invoquer `read()`, et intervient uniquement si celle-ci retourne 0, auquel cas elle change la valeur de retour en -1 et définit `errno` comme `crypto_read()`.

### 4.1.2 Fonctionnement interne

Les fonctions de lecture et d'écriture ont besoin d'une série de données pour chaque flux qu'elles traitent. Cependant par souci d'homogénéité avec l'interface des appels systèmes standards, il a été choisi de ne rien demander d'autre au programme qu'un descripteur de fichier. Voici donc comment le lien est fait entre un descripteur de fichier et la liste de variables nécessaires à son traitement :

Comme les descripteurs de fichiers sont des entiers, alloués par le système de manière croissante, ils sont en fait utilisés par la librairie comme un index dans une table, chaque élément de cette table référençant une structure contenant les données nécessaires à un flux. La table en question est dynamique, et est agrandie ou réduite au fur et à mesure de son utilisation.

Ces structures comportent les données suivantes :

- un buffer de lecture,
- un buffer d'écriture,
- des pointeurs dans ces buffers,
- une structure de contrôle AES de la *libtomcrypt*, utilisée pour le chiffrement,
- une structure de contrôle AES utilisée pour le déchiffrement.

Le buffer de lecture contient les données chiffrées lues, ainsi que le texte en clair une fois le déchiffrement effectué. Il est accompagné de trois pointeurs, marquant respectivement le début du texte en clair à renvoyer à la prochaine lecture (utilisé dans le cas où la première lecture après déchiffrement n'ait pas consommé l'intégralité du buffer), la fin du texte chiffré lue jusqu'à présent (dans le cas d'une lecture partielle depuis le fd), et la fin du texte chiffré attendu (ce dernier pointeur vaut 0 tant que cette taille n'est pas connue, c'est à dire tant que le premier bloc contenant le nombre de blocs de texte n'a pas été lu et déchiffré).

Le buffer d'écriture est l'endroit où les données à émettre sont mises en forme (voir plus loin) avant hashage et chiffrement. Il est accompagné de 2 pointeurs, indiquant respectivement les données déjà envoyées et la fin des données à envoyer. Un troisième champ contient la longueur du texte en clair, il est utilisé comme valeur de retour lors d'un envoi effectué avec succès.

Voici comment fonctionne la fonction `crypto_write()` :

Elle recopie le texte fourni par l'utilisateur dans son buffer interne, en laissant la place pour un octet au début du buffer. Celui-ci sera initialisé avec la taille du buffer par la suite. Une fois la copie effectuée, des octets de padding sont ajoutés à la fin du texte, de manière à obtenir une taille totale (y compris l'octet de taille) qui soit un multiple de 16 octets (taille des blocs chiffrés par AES). Si le texte arrive déjà sur une limite de 16 octets, 16 octets de

padding sont ajoutés. Les octets de padding sont initialisés par leur nombre : si on ajoute 2 octets de padding, chacun contiendra la valeur 2.

On regarde alors le nombre de blocs de 16 octets occupés par le message, et on met cette valeur moins 1 dans le premier octet, réservé à cet effet. Notons que cet encodage limite la taille totale du texte en clair que l'on peut émettre en une seule fois à (255+1)*16-2 octets : la valeur maximale stockable dans le premier octet est 255, qui correspond à 256 blocs de 16 octets, dont 1 réservé pour la taille, et au moins 1 pour le padding. On peut donc envoyer un message d'au plus 4094 octets. Ceci n'est pas contraire à l'interface, puisque `write()` peut tronquer le message d'après POSIX, tant qu'il renvoie la bonne valeur de retour.

On calcule ensuite un hash du message obtenu, que l'on rajoute à la suite de celui-ci, avant de chiffrer le tout en AES CBC. Le buffer est alors prêt à être envoyé, au moyen de l'appel `write()`.

La fonction `crypto_read()` réalise l'opération inverse :

Elle commence par lire les 16 premiers octets disponibles depuis le descripteur de fichiers, et les déchiffre pour connaitre la valeur du premier octet, dont elle se sert pour calculer la taille totale de données restant à lire. Une fois ces données lues, elles sont déchiffrées, et la valeur du hash est vérifiée. Si le résultat est positif, les données sont alors retournées à l'utilisateur. Tant que l'intégralité du message n'a pu être lue, la fonction renvoie la valeur 0, sans modifier le buffer fourni. La valeur du dernier octet du texte en clair, c'est à dire d'un octet de padding, est utilisée pour déterminer la fin des données réellement émises à l'autre bout du tuyau.

L'établissement de session réutilise ces composants.

Tout d'abord un descripteur de session est alloué pour le descripteur de fichier. Ensuite, le chiffrement AES est initialisé grâce au secret fourni en paramètre de `crypto_session_setup()`. Ensuite une clé Diffie-Hellman est généré, la clé publique est copiée dans un buffer, après l'octet "asym" trouvé dans les paramètres de la fonction, et le tout est envoyé à la fonction `crypto_write()`, qui va se charger du chiffrement et de la vérification d'intégrité. De même, `crypto_read()` est utilisé pour lire la clé publique de l'autre interlocuteur. L'octet asym est vérifié, et enfin le secret commun est généré. Ce secret est alors passé à la fonction `setup_cipher()`, qui va hasher celui-ci pour génerer la clé de chiffrement, et le rehasher pour calculer l'IV qui sera utilisé par AES pour cette session.

La mémoire où sont copiées des données sensibles est réinitialisée.

# Bibliographie


[1] The internet content adaptation protocol. http ://www.faqs.org/rfcs/rfc3507.html.

[2] The openssl project. http ://www.openssl.org/.

[3] Openwrt. http ://openwrt.org/.

[4] Rule order in iptables. http ://www.faqs.org/docs/iptables/traversingoftables.html.

[5] Mick Bauer. Paranoid penguin : Using iptables for local security. 100, aug 2002. http ://www.linuxjournal.com/article.php ?sid=6091.

[6] Kernighan Brian and Dennis Ritchie. *The C programming language*. Prentice Hall, 1998.

[7] Joan Daemen and Vincent Rijmen. *The design of Rijndael : AES — the Advanced Encryption Standard*. 2002.

[8] Tim Dierks and Christopher Allen. The tls protocol version 1.0. jan 1999. http ://kaizi.viagenie.qc.ca/ietf/rfc/rfc2246.txt.

[9] FIPS. *The Keyed-Hash Message Authentication Code (HMAC)*, 2002. http ://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf.

[10] Alan Freier, Philip Karlton, and Paul Kocher. The ssl protocol version 3.0. November 1996. http ://wp.netscape.com/eng/ssl3/draft302.txt.

[11] Fyodor. Remote OS detection via TCP/IP stack fingerprinting, oct 1998. http ://www.insecure.org/nmap/nmap-fingerprinting-article.html.

[12] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[13] et. al. R. Fielding. Hypertext transfer protocol – http/1.1. RFC 2616, Internet Engineering Task Force, jun 1999. ftp ://ftp.isi.edu/in-notes/rfc2616.txt.

[14] John Rhoton. *Programmer's guide to Internet mail : SMTP, POP, IMAP, and LDAP*. 2000.

[15] RSA Data Security, Inc. *PKCS #3 : Diffie-Hellman Key Agreement Standard*, jun 1991. Version 1.3.

[16] Werner Schindler. A timing attack against rsa with the chinese remainder theroem. pages 109–124, 2000.

[17] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, 1994.

[18] Lance Spitzner. The Honeynet Project : Trapping the hackers. *IEEE Security & Privacy*, 1(2), March/April 2003. http ://www.computer.org/security/j2015abs.htm ; http ://dlib.computer.org/sp/books/sp2003/pdf/j2015.pdf.

[19] W. Richard Stevens. *TCP/IP Illustrated, Volume 1 : The protocols*. Addison-Wesley, 1996.

[20] Andrew S. Tanenbaum. *Modern Operating Systems*. Internals and Design Principles. Prentice-Hall, 1992.