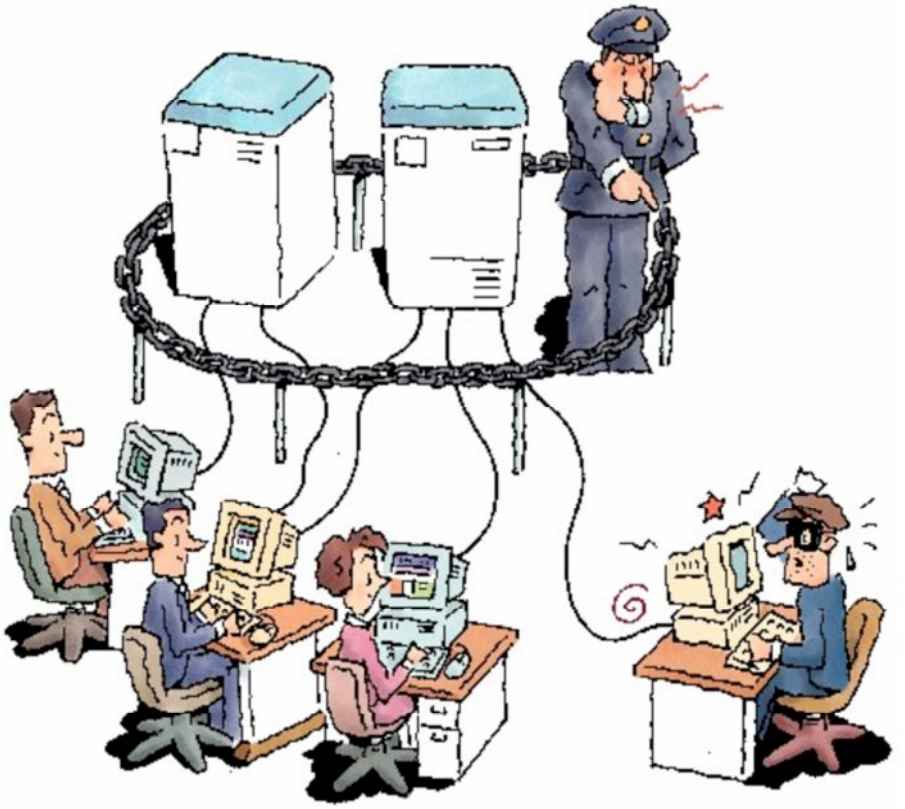
Travail de diplôme

Sécurité des applications Web



Auteur : Sylvain Tissot

Professeurs : Sylvain Maret
Stefano Ventura

Expert : Gérald Litzistorf

Yverdon, le 18 décembre 2003

Table des matières

1. Résumé

Ce travail de diplôme traite des problèmes de sécurité des applications Web en se plaçant entre le client et le serveur Web. Un autre travail de diplôme mené par Jonathan Rod s'intéresse plus particulièrement à la sécurité des services Web.

Le travail de semestre qui a précédé ce travail était l'occasion de faire une étude des principales vulnérabilités des applications Web et d'une solution imparfaite basée sur Apache et mod_rewrite pour s'en protéger. Ces sujets ne seront que sommairement abordés dans le cadre de ce travail de diplôme, le rapport de travail de semestre est donné en annexe pour référence-

1.1. Problématique

La sécurité des applications Web est critique car celle-ci sont généralement publiquement accessibles sur Internet, et donc la moindre vulnérabilité à ce niveau peut être exploitée par n'importe quel *hacker* dans le monde.

L'OWASP (*Open Web Application Security Project*) a dressé un "top 10" des vulnérabilités les plus souvent rencontrées dans les applications Web, tous produits confondus, et l'on constate que la plupart de ces vulnérabilités peuvent être exploitées par simple manipulation d'URL ou injection de paramètres dans un champ de formulaire pour, par exemple, contourner un mécanisme d'authentification ou faire exécuter du code malicieux au serveur. Ces attaques empruntent toutes le "canal sûr" du port TCP 80 et ne sont par conséquent pas bloquées par les firewalls IP conventionnels. L'utilisation de SSL ne résoud pas le problème puisque nous ne sommes pas en présence d'une attaque de type *Man In The Middle*.

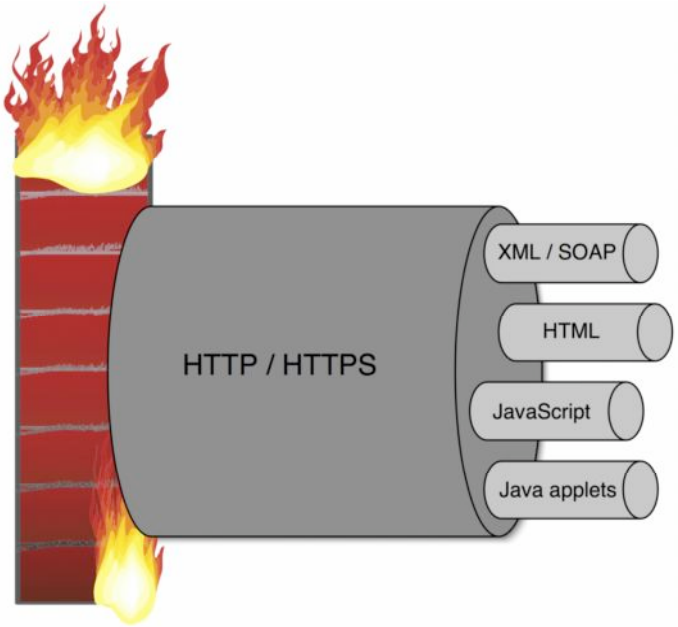


Figure 1.1-1: les attaques de niveau applicatif ne sont pas bloquées par un firewall conventionnel

Les vulnérabilités des applications Web se répartissent en 3 catégories :

1. Vulnérabilités du logiciel serveur Web ou d'application (IIS, Apache, Tomcat etc...)
2. Vulnérabilités du système d'exploitation du serveur (Windows, Linux, Mac OS, etc..)
3. Vulnérabilités du code de l'application elle-même (PHP, Perl, Java, etc...)

Les deux premières catégories de vulnérabilités peuvent être évitées en utilisant des produits réputés sûr (Apache 1.3 sur un système NetBSD, par exemple) correctement configurés et en appliquant systématiquement les derniers *patchs* de sécurité.

Les vulnérabilités dans le code même de l'application sont dues à des négligences de programmation de la part des développeurs Web, souvent peu sensibilisés aux impératifs de sécurité et pressés par le temps. Ceux-ci ne se rendent pas compte des failles qu'ils peuvent laisser dans leur code, et de graves vulnérabilités sont découvertes régulièrement dans des applications ou sites pourtant renommés. C'est à cette troisième catégorie de vulnérabilités que nous nous intéressons particulièrement dans ce travail.

1.2. Mandat

Ce document est composé de trois parties.

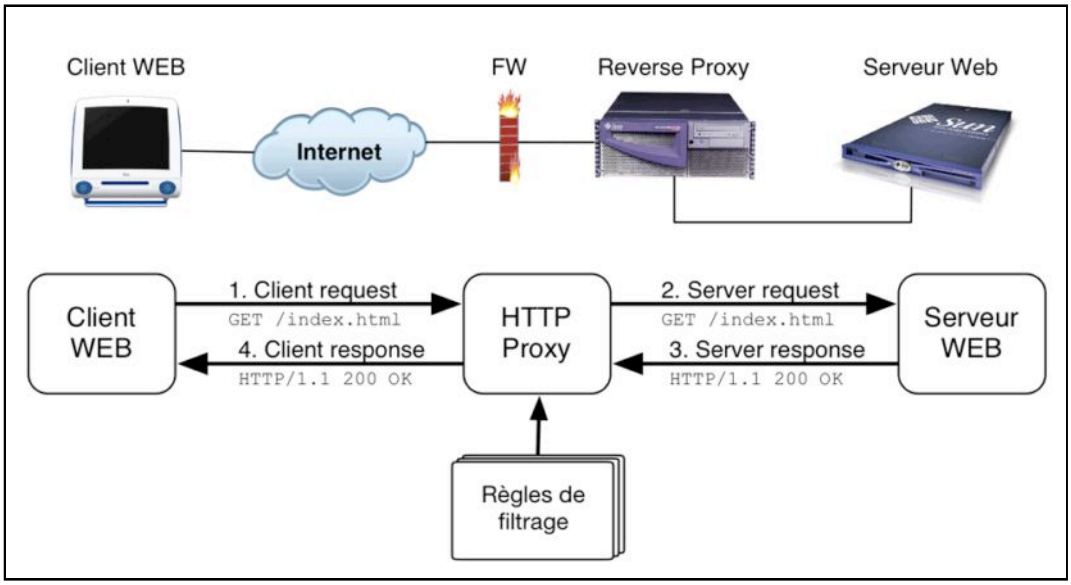
La première partie est une étude du protocole HTTP, il s'agit de poser les bases théoriques pour la suite, afin de bien comprendre ce qu'un reverse proxy cherche à filtrer.

La seconde partie est expérimentale. Il s'agit d'étudier le firewall applicatif Sanctum AppShield et de le configurer en vue de protéger une application Web. Cette étude doit mener au développement de notre propre firewall applicatif.

La troisième partie de ce travail s'intéresse au développement de *ProxyFilter*, un firewall applicatif HTTP basé sur Apache et Perl. Il doit notamment permettre de réécrire et filtrer les URLs, filtrer les paramètres GET et POST, filtrer les entêtes de la requête et de la réponse. Nous décrirons dans une premier chapitre le processus de développement, et dans un second chapitre le fonctionnement et la configuration du produit.

1.3. Fonctionnement

ProxyFilter joue le rôle d'un *reverse proxy* qui vient se placer entre le client et le serveur Web, généralement sur la DMZ (zone publique démilitarisée) d'un firewall d'entreprise. Il reçoit les requêtes en provenance du client, les filtre et les transmet ensuite au serveur Web. Le *reverse proxy* est un intermédiaire obligé pour atteindre le serveur Web, ce dernier étant généralement protégé d'un accès direct depuis l'extérieur par un firewall IP.



Figrue 1.3: schéma de principe de ProxyFilter

La configuration se fait au moyen de règles permettant d'indiquer au *reverse proxy* quelles sont les requêtes autorisées et quelles sont celles qui doivent être bloquées. Les modes de fonctionnement *Black List* (filtrage exclusif) et *White List* (filtrage inclusif) sont supportés. Le langage de programmation utilisé est Perl et la syntaxe de configuration est dérivée de XML.

1.4. Conclusion

Bien que *ProxyFilter* n'offre de loin pas tous les perfectionnements d'une solution commerciale comme AppShield, il permet en étant corrrectement configuré de protéger une application Web contre la plupart des attaques courantes (Cross-Site Scripting, SQL injection, Buffer overflow, Cookie poisoning, etc...). Un test effectué sur l'application d'exemple *securitystore.ch* à l'aide du scanner Nikto a reporté 202 vulnérabilités sur 1335 testées sans utiliser *ProxyFilter* et 5 vulnérabilités sur 2615 testées en passant par *ProxyFilter*.

ProxyFilter est un logiciel *open source* sous licence GPL qui est appelé à évoluer au-delà du présent travail de diplôme. Le code source et les explications détaillées de fonctionnement, d'installation et de configuration sont disponibles sur le Web à l'adresse :

http://proxyfilter.sourceforge.net

2. Le protocole HTTP

HTTP (*Hypertext Transfer Protocol*) est le protocole utilisé pour le World Wide Web. Il fonctionne selon un principe de requête/réponse : le client transmet une requête comportant des informations sur le document demandé et le serveur renvoie le document si disponible ou, le cas échéant, un message d'erreur. Chaque requête/réponse est indépendante, il n'y a pas de notion de session en HTTP au contraire de protocoles comme FTP ou SMTP.

Le protocole HTTP tire ses origines de l'invention du Web au CERN au début des années 1990, qui avait besoin d'un protocole de transfert très simple. Il existe deux versions de HTTP actuellement utilisées : la version 1.0 décrite dans la RFC 1945 de 1996 et la version 1.1 décrite dans la RFC 2616 de 1999. L'une des améliorations majeures apportées par la version 1.1 est la possibilité de ne pas refermer la connexion TCP après chaque requête (*Keep-Alive*), et ainsi de transmettre plusieurs requêtes sur une même connexion TCP, apportant un gain de performance appréciable, en particulier pour des documents comportant de nombreuses images.

2.1. Syntaxe d'une URL

Une URL (*Universal Ressource Locator*) permet d'identifier univoquement un document au niveau mondial. La documentation du serveur Apache parle plutôt de URI (*Universal Ressource Identifier*) qui est un concept plus global regroupant les URL. Nous les considérerons comme des synonymes.

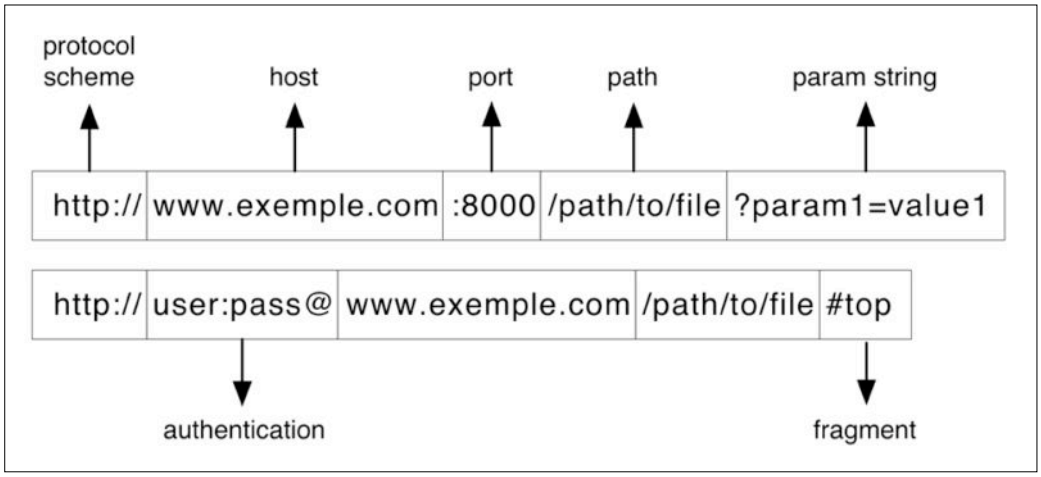


Figure 2.1.1 : syntaxe générale d'une URL

2.2. Méthodes HTTP

Les principales méthodes HTTP sont GET (pour demander un document), POST (pour transmettre des données, d'un formulaire par exemple) et HEAD (pour ne recevoir que les lignes d'entête de la réponse sans le corps du document). Des extensions à HTTP comme WebDAV définissent d'autres méthodes telles que PUT pour publier un document ou DELETE pour effacer un document. La méthode utilisée est indiquée dans le premier champ de la ligne de requête.

2.3. Requête HTTP

La requête transmise par le client au serveur comprend une ligne de requête qui contient la méthode, l'URL du document demandé et la version du protocole HTTP. La ligne de requête est suivie par une ou plusieurs lignes d'entêtes, chacune comportant un nom et une valeur.

La requête peut optionnellement contenir une entité (contenu). Celle-ci est notamment utilisée pour transmettre des paramètres avec la méthode POST. L'entité est transmise après les lignes d'entêtes, elle est séparée de la dernière entête par un double CRLF (*carriage return* et *linefeed*).

```
GET /index.html HTTP/1.1
Host: www.example.com
Accept: */*
Accept-Language: fr
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X; fr)
Connection: Keep-Alive
```

Figure 2.3-1 : exemple de requête HTTP

Dans cet exemple, le client demande le document à l'adresse http://www.example.com/index.html, il accepte tous les types de document en retour, préfère les documents en français, utilise un navigateur compatible Mozilla 5.0 sur un système Mac OS X et signale au serveur qu'il faut garder la connexion TCP ouverte à l'issue de la requête (car il a d'autres requêtes à transmettre).

2.4. Réponse HTTP

La première ligne du message de la répone est la ligne de statut. Elle est composée de la version du protocole HTTP, d'un code de statut et d'un libellé décrivant le code de statut. La ligne de statut est suivie de une ou plusieurs lignes d'entêtes, chacune comportant un nom et une valeur.

La dernière ligne d'entête est suivie d'un double CRLF marquant le début du corps du document retourné (les données HTML ou binaires par exemple). Une réponse ne contient pas forcément un corps : s'il s'agit d'une réponse à une requête HEAD, seule la ligne de statut et les entêtes sont retournés.

```
HTTP/1.1 200 OK
Date: Mon, 15 Dec 2003 23:48:34 GMT
Server: Apache/1.3.27 (Darwin) PHP/4.3.2 mod_perl/1.26 DAV/1.0.3
Cache-Control: max-age=60
Expires: Mon, 15 Dec 2003 23:49:34 GMT
Last-Modified: Fri, 04 May 2001 00:00:38 GMT
ETag: "26206-5b0-3af1f126"
Accept-Ranges: bytes
Content-Length: 1456
Content-Type: text/html

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
...
```

Figure 2.4-1 : exemple de réponse HTTP

Dans cet exemple, le code 200 nous indique que le document demandé a été trouvé et suit les entêtes. Pour faciliter la gestion du cache performance du client, le serveur transmet la date actuelle, la date de dernière modification du document et la date d'expiration (après laquelle le document doit être demandé à nouveau). L'entête *Content-Type* nous apprend que le document retourné est de type HTML et l'entête *Content-Length* indique que le corps du document a une longueur de 1456 octets. L'entête *Server* renseigne sur le logiciel serveur utilisé, ce qui n'est pas franchement souhaitable d'un point de vue sécurité.

2.5. Entêtes HTTP génériques

Certaines entêtes peuvent se trouver aussi bien dans la requête que dans la réponse. Les principales sont données dans le tableau ci-après :

Champ	Description
Content-length	Longueur en octets des données qui suivent
Content-type	Type MIME des données qui suivent
Connection	Indique si la connexion TCP doit rester ouverte (*Keep-Alive*)

Figure 2.5-1 : entêtes HTTP génériques

2.6. Entêtes HTTP de la requête

Les entêtes données dans le tableau ci-après sont spécifiques à la requête transmise par le client :

Champ	Description
Accept	Types MIME que le client accepte
Accept-encoding	Méthodes de compression que le client supporte
Accept-language	Langues préférées par le client (pondérées)
Cookie	Données de cookie mémorisées côté client
Host	Hôte virtuel demandé
If-modified-since	Ne retourne le document que si modifié depuis la date indiquée
If-none-match	Ne retourne le document que s'il a changé
Referer	URL du document qui contenait le lien au document demandé
User-agent	Nom et version du logiciel client

Figure 2.6-1 : entêtes HTTP de la requête

2.7. Entêtes HTTP de la réponse

Les entêtes données dans le tableau ci-après sont spécifiques à la réponse du serveur :

Champ	Description
Allowed	Méthodes HTTP autorisée pour cette URI (comme POST)
Content-encoding	Méthode de compression des données qui suivent
Content-language	Langue dans laquelle le document retourné est écrit
Date	Date et heure UTC courante
Expires	Date à laquelle le document expire
Last-modified	Date de dernière modification du document
Location	Adresse du document lors d'une redirection
ETag	Numéro de version opaque du document
Pragma	Données annexes pour la navigateur (comme "no-cache")
Server	Nom et version du logiciel serveur
Set-cookie	Permet au serveur d'écrire un cookie sur le disque du client

Figure 2.7-1 : entêtes HTTP de la réponse

2.8. Codes de statut

Lorsque le serveur renvoie un document, il lui associe un code de statut qui renseigne le client sur le résultat de la requête (document non trouvé, requête invalide, etc...). Les principaux codes de statut HTTP et leur nom en anglais sont donnés ci-après.

Code	Nom	Description
Succès 2xx		
200	OK	Le document a été trouvé et son contenu suit
201	CREATED	Le document a été créée en réponse à un PUT
202	Accepted	Requête acceptée, mais traitement non-terminé
206	Partial Content	Une partie du document suit
204	No Response	Le serveur n'a aucune information à renvoyer
Redirection 3xx		
301	Moved	Le document a changé d'adresse de façon permanente
302	Found	Le document a changé d'adresse temporairement
304	Not Modified	Le document demandé n'a pas été modifié
Erreurs du client 4xx		
400	Bad Request	La syntaxe de la requête est incorrecte
401	Unauthorized	Le client n'a pas les privilèges d'accès au document
403	Forbidden	L'accès au document est interdit
404	Not Found	Le document demandé n'a pu être trouvé
405	Method Not Allowed	La méthode de la requête n'est pas autorisée

Code	Nom	Description
Erreurs du serveur 5xx		
500	Internal Error	Une erreur inattendue est survenue au niveau du serveur
501	Not Implemented	La méthode utilisée n'est pas implémentée
502	Bad Gateway	Erreur de serveur distant lors d'une requête proxy

Figure 2.8-1 : principaux codes de statut HTTP

Les codes de statut les plus rencontrés sont 200, qui indique que le document a été trouvé et la requête traitée avec succès, et 404 qui indique un document non-trouvé.

2.9. Cookies

Les *cookies* sont un moyen pour le serveur de mémoriser des données du côté client. Cela permet un suivi de l'utilisateur d'une requête à l'autre et d'implémenter une sorte de session que le protocole HTTP n'offre pas lui-même.

Pour écrire un cookie du côté client, le serveur place une entête *Set-Cookie* dans sa réponse, comprenant le nom du cookie, sa valeur, son contexte (path) et sa date d'expiration. Dans les requêtes suivantes et jusqu'à expiration, le client indique dans l'entête *Cookie* les informations qu'ils a reçues du serveur.

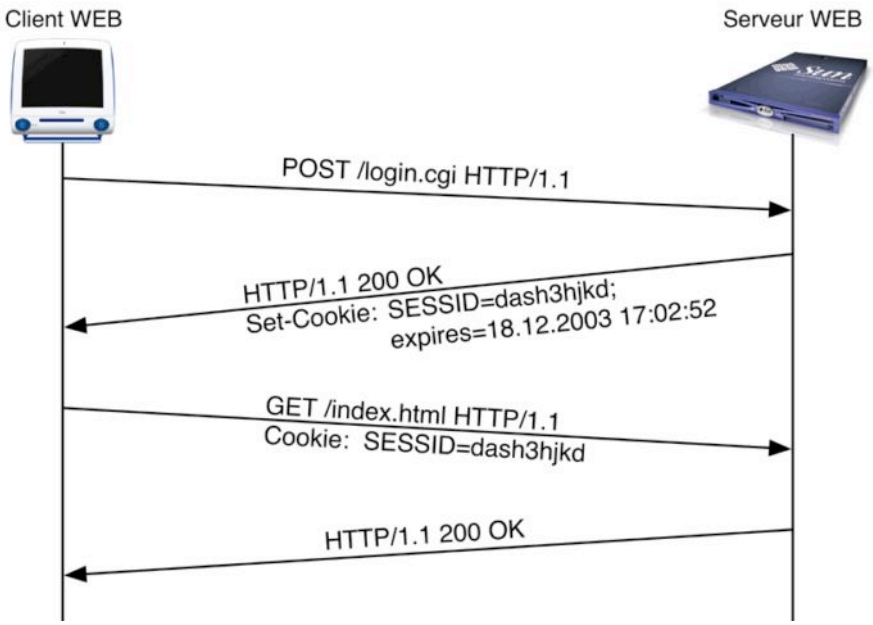


Figure 2.9-1 : principe de fonctionnement des cookies

On utilisera typiquement un cookie pour mémoriser une clé de session à validité limitée, mais pas pour mémoriser des données critiques comme un mot de passe ou un numéro de carte bancaire, car la base de données des cookies côté client n'est généralement pas cryptée et peut être facilement consultée par d'autres utilisateurs du poste client.

2.10. Redirections

Lorsque le serveur renvoie un code de statut de la série 3xx, il accompagne sa réponse d'une entête *Location* indiquant la nouvelle adresse du document. Dans ce cas, le client va automatiquement émettre une requête vers cette nouvelle adresse. L'URL indiquée dans l'entête *Location* devrait être absolue, mais l'on tolère dans la pratique une URL relative au document courant.

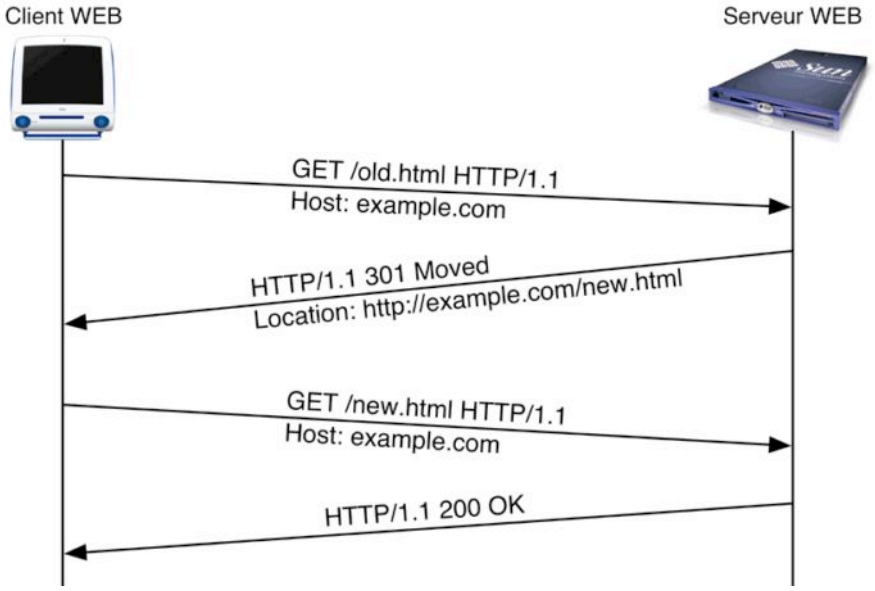


Figure 2.10-1 : principe d'une redirection

3. Étude de Sanctum AppShield

AppShield de la société Sanctum est un produit bien établi depuis 1999 dans le domaine de la sécurité des applications Web. Il s'agit d'un firewall applicatif HTTP d'une grande flexibilité et relativement aisé à configurer grâce à son interface graphique et son système de génération dynamique de règles.

Comme son concurrent InterDo, AppShield utilise un modèle de sécurité positif (*White List)*, ce qui signifie un plus haut niveau de sécurité que les produits qui utilisent un modèle de sécurité négatif (*Black List*), mais également un produit plus complexe et un prix élevé, environ 15'000$ par serveur dans le cas de AppShield.

Pour ce travail de diplôme et avec l'aide de M. Maret de eXpert Solutions, nous avons pu obtenir une licence d'évaluation monoposte de AppShield 4.0 permettant de tester le logiciel sans limitation de ses fonctionnalités durant 30 jours.

3.1. Caractéristiques

Voici un bref aperçu des fonctionnalités de AppShield :

- **Haute performance**

  AppShield est spécialement optimisé pour un haut niveau de disponibilité et performance. Il est adapté à des sites fortement dynamiques et au e-commerce.

- **Génération automatique des règles**

  En observant le trafic en mode passif, AppShield peut générer dynamiquement des règles qui seront appliquées une fois basculé en mode passif

- **Firewall stateful**

  AppShield utilise une technique unique pour générer des règles positives en temps réel : il évalue les liens hypertextes et les champs de formulaires de chaque document HTML retourné au client et crée les règles positives correspondantes. Cela permet de réduire drastiquement la configuration de AppShield propre à l'application.

- **Historique détaillé**

  Une base de données MySQL est utilisée pour sauvegarder l'historique et la configuration, permettant une grande souplesse d'affichage de l'historique (par client, par session, uniquement les requêtes bloquées, etc...). Une fonction de conversion de ligne d'historique en règle est proposée.

- **Support de SSL en entrée et en sortie**

  AppShield supporte SSL aussi bien avec le client qu'avec le serveur. Il est possible de terminer un connexion SSL sur le proxy pour arriver en clair sur le serveur (*Encrypted To Clear*), de communiquer en clair avec le client et par SSL avec le serveur (*Clear To Encrypted*) ou d'utiliser à la fois SSL avec le client et le serveur (*Encrypted To Encrypted*). Depuis la version 4.0, AppShield gère également l'authentification au moyen d'un certificat côté client.

- **Mapping d'URL**

  Des fonctions avancées de reverse proxy permettent de réécrire l'URL source par substitution de préfixe pour la faire pointer sur différents serveurs en fonction de son contexte. La réécriture inverse des redirections est également gérée.

- **Historique détaillé**

  Chaque noeud AppShield mémorise dans une base MySQL un historique des transactions qui peut être visualisé sous plusieurs forme au moyen de filtres.

- **Multiplateforme**

  La version 4.0 de AppShield s'installe sur Windows NT4, 2000 et Solaris 8.

- **Watchdog**

  AppShield est monitoré en permanence et, en cas de comportement anormal, l'administrateur est aussitôt notifié.

- **Support de OPSEC**

  En cas de tentative de *hacking*, AppShield peut notifier le firewall Check Point au travers de OPSEC (*Open Plateform for Security*) afin qu'il bloque l'adresse IP de l'attaquant au niveau réseau.

- **Pages d'erreur personnalisables**

  L'utilisateur peut personnaliser les pages d'erreur renvoyées par AppShield au client (statiques ou dynamiques) afin qu'elles collent mieux à la présentation de l'application et qu'elles ne révèlent d'information ni sur le firewall applicatif utilisé, ni sur le serveur Web ou d'application utilisé.

3.2. Installation

AppShield peut être installé sur une ou plusieurs machines du réseau (on parle de "noeuds" AppShield). Il est conseillé d'installer AppShield sur une machine dédiée, bien qu'il soit également possible, pour des charges pas trop élevées, de l'installer sur la même machine que le serveur Web.

Sur chaque noeud AppShield, il faut installer le *Security Engine* qui est la pièce maitresse de l'application, un démon réalisant la fonction de proxy et le filtrage du trafic proprement dit.

Si plusieurs noeuds AppShield coexistent sur le réseau, seul l'un d'entre-eux doit tourner le serveur de configuration, un élément logiciel centralisant la configuration pour tous les noeuds AppShield. Les autres noeuds téléchargent automatiquement leur configuration depuis le serveur de configuration, avec un mécanisme de recherche proche du protocole DHCP (utilisant le *broadcast*). Pour ces raisons, tous les noeuds AppShield de l'application doivent être situés dans un même domaine de multidiffusion (ne pas les séparer par un firewall).

Pour configurer AppShield, on utilise une interface basée sur Java qui établit une liaison sécurisée avec le serveur de configuration. L'interface de configuration ne doit pas obligatoirement tourner sur un noeud AppShield : l'administration peut se faire à distance, à travers un VPN par exemple.

Dans le cadre de ce travail, nous avons installé AppShield 4.0 sur un système Windows 2000 SP4, à savoir le *Security Engine*, le serveur de configuration et l'interface de configuration tous sur la même machine. Une station Mac OS X annexe avec Apache, PHP et MySQL a été utilisée comme serveur et comme client pour les tests.

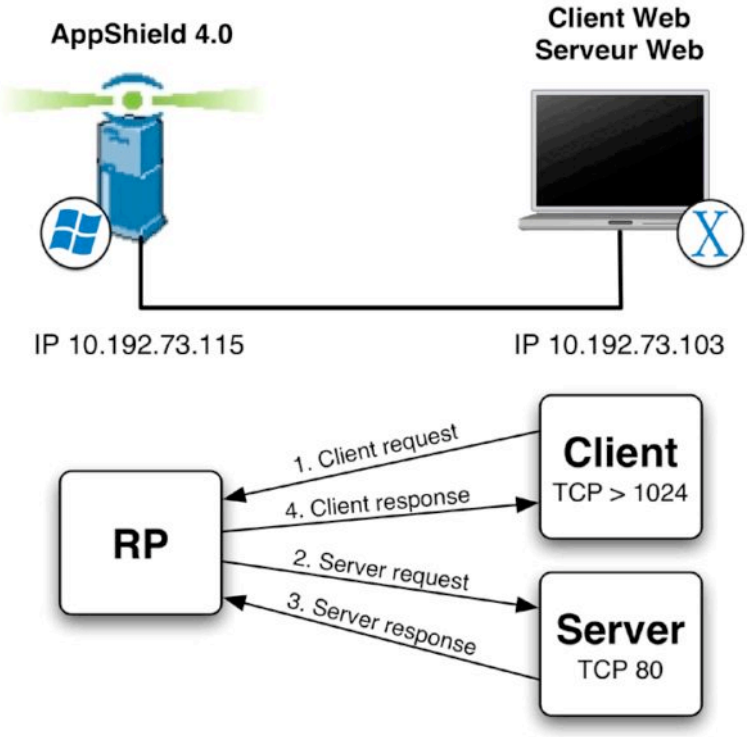


Figure 3.2-1 : maquette d'expérimentation de AppShield

Cette topologie ne correspond évidemment pas à un environnement de production réel où client, serveur et proxy sont des machines différentes, le proxy dispose de deux interfaces réseau et un firewall IP est installé devant le proxy. Elle est cependant suffisante pour tester AppShield en local. Le fait que le client et le serveur soient sur la même machine ne perturbe pas le fonctionnement, car ceux-ci utilisent des ports différents.

AppShield s'est révélé gourmand en mémoire : sur notre station dotée de 256 Mo de mémoire vive, des messages indiquant que AppShield commençait en manquer et risquait de devenir instable et vulnérable apparaissaient de temps en temps. De toute évidence, il vaut mieux ne pas tourner l'interface de configuration Java sur un noeud AppShield, et évidemment ne pas installer AppShield sur le serveur Web si l'on veut éviter les ennuis.

À la fin de l'installation, AppShield réclame l'ajout d'un fichier de licence. En l'absence de celui-ci, AppShield travaillera en mode passif, c'est à dire qu'il établira un historique détaillé des accès et des attaques, évaluera chaque requête relativement à sa politique de sécurité pour déterminer si elle doit être bloquée, mais ne la bloquera pas le cas échéant : il se contente d'observer le trafic et de reporter les attaques sans les filtrer. Pour passer en mode actif, une licence valide est demandée. Les licences sont délivrées au cas par cas par Sanctum, pour une durée.limitée et sont dépendantes de l'adresse MAC de la carte réseau (on ne peut pas utiliser la licence sur une autre machine). Cette politique très restrictive de licence s'explique sans doute par le prix élevé du logiciel.

3.3. Topologies

Différents noeuds AppShield peuvent être combinés de façon à s'adapter à toutes les structures de réseau : serveurs miroirs, répartisseurs de charge avant ou après les noeuds AppShield, répartition du contenu de l'application sur plusieurs groupes de serveurs, etc...

Quelle que soit la configuration retenue, on protégera les noeuds AppShield de l'extérieur au niveau réseau au moyen d'un firewall IP en ne laissant ouverts que les ports TCP 80 (HTTP) et/ou TCP 443 (HTTPS).

3.3.1. Configuration un à un

Ceci est la configuration par défaut : chaque noeud AppShield protège un seul serveur Web. L'entrée DNS du site pointe sur l'adresse de l'interface externe du noeud AppShield, celui-ci reçoit les requêtes, les vérifie et les transmet au serveur Web. Pour le serveur, toutes les requêtes ne semblent venir que d'un seul client, le noeud AppShield. Le client a l'impression que le noeud AppShield est le serveur Web.

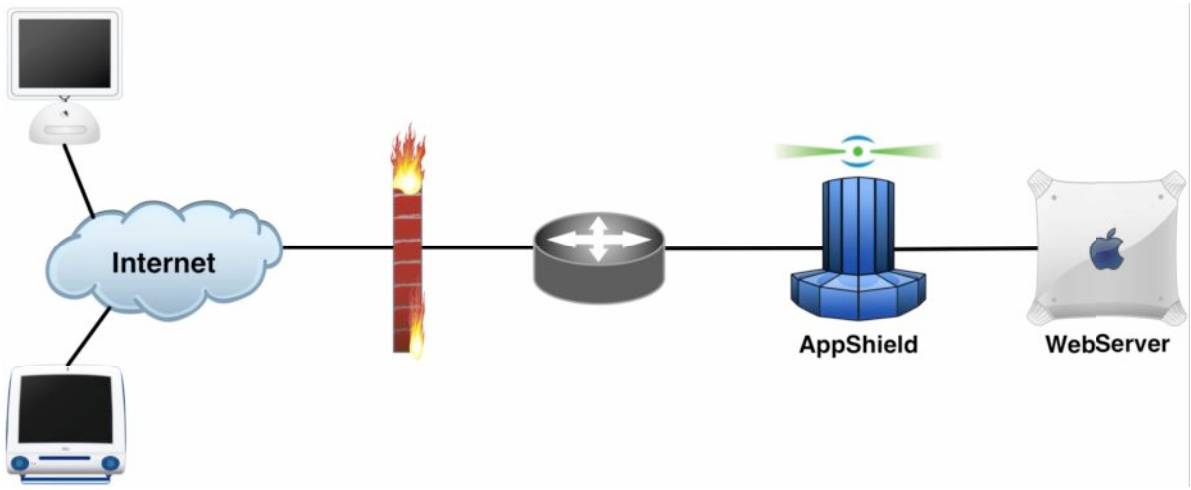


Figure 3.3-1 : Configuration "un à un"

Pour activer cette configuration, il suffit d'indiquer à AppShield l'adresse sur laquelle écouter les requêtes (s'il dispose de plusieurs adresses IP ou interfaces) et l'adresse du serveur Web où rediriger les requêtes. Il est également recommandé d'indiquer à AppShield les noms d'hôte ou de domaine valides pour lesquels il doit accepter des requêtes. Il est évident que le firewall est configuré de façon à ne pas permettre un accès direct au serveur Web depuis l'extérieur, sans quoi AppShield pourrait être contourné. On pourra placer les noeuds AppShield sur une première DMZ, et les serveurs Web sur une seconde DMZ, par exemple.

3.3.2. Configuration avec serveurs multiples

Dans cette configuration, un seul noeud AppShield protège plusieurs serveurs Web non-miroirs. L'interface externe de AppShield possède une adresse IP par serveur, ce qui lui permet de déterminer facilement à quel serveur est destinée une requête.

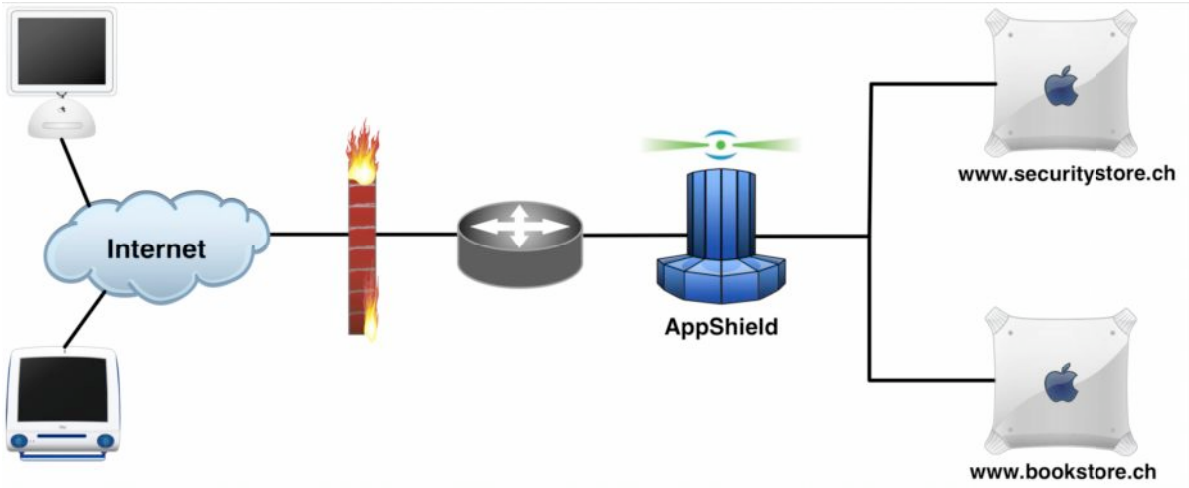


Figure 3.3-2 : configuration avec serveurs multiples

Il est possible de multiplier autant que nécessaire le nombre de serveurs protégés par un seul noeud AppShield dans les limites de charge mémoire et processeur la machine hébergeant AppShield, mais on remarquera que dans cette configuration, AppShield est le maillon faible de la chaîne.

3.3.3. Configuration avec serveurs miroirs

Chaque serveur Web héberge une copie de la même application. On attribue à AppShield une adresse IP par serveur Web. Un répartiteur de charge, placé avant AppShield, renvoie la requête à l'une des adresses IP de AppShield, qui la filtre et l'envoie sur le serveur correspondant. Le répartiteur de charge (LB) peut travailler en simple tourniquet ou, plus intelligement, connaître la charge instantanée de chacun des serveurs et envoyer la requête au serveur le moins chargé.

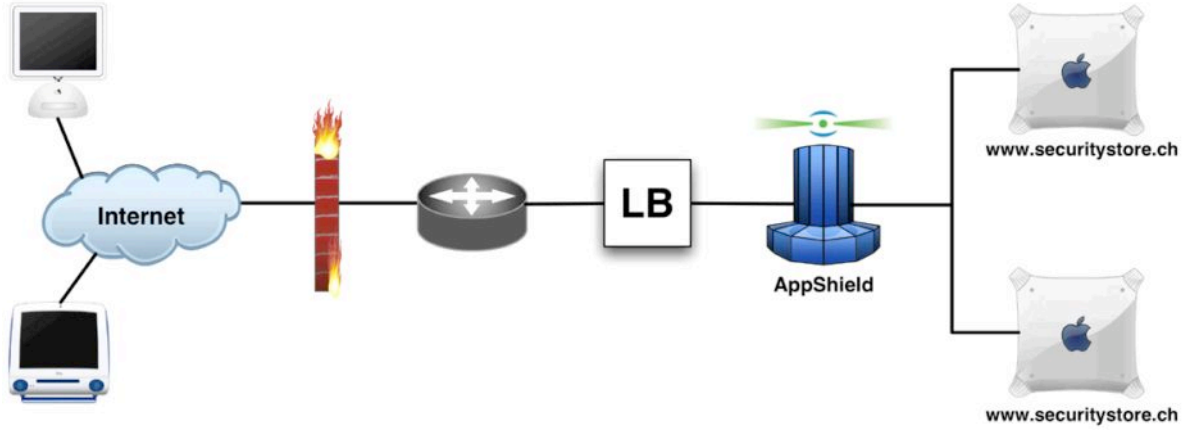


Figure 3.3-3 : configuration avec serveurs miroirs

Cette configuration pose des problèmes avec les applications qui emploient des fichiers temporaires sur le serveur pour mémoriser les informations de sessions, comme le fait PHP par exemple (dossier /tmp), car deux requêtes ne sont jamais certaines d'aboutir au même serveur. Dans ce cas, il faut exclusivement utiliser une base de données pour mémoriser les informations de session ou un volume partagé commun à tous les serveurs Web.

3.3.4. Configuration plusieurs à plusieurs

Cette topologie est un mélange entre serveurs multiples et serveurs miroirs. Plusieurs noeuds AppShield ont été installés sur le réseau, qui travaillent en parallèle pour fournir un débit plus élevé et une redondance matérielle. Chaque noeud AppShield dispose d'une adresse IP par serveur. Un répartiteur de charge, situé en amont des noeuds AppShield, dirige la requête vers l'un des noeuds AppShield, à l'adresse IP correspondante au serveur concerné. Le noeud AppShield filtre la requête et la transmet au serveur.

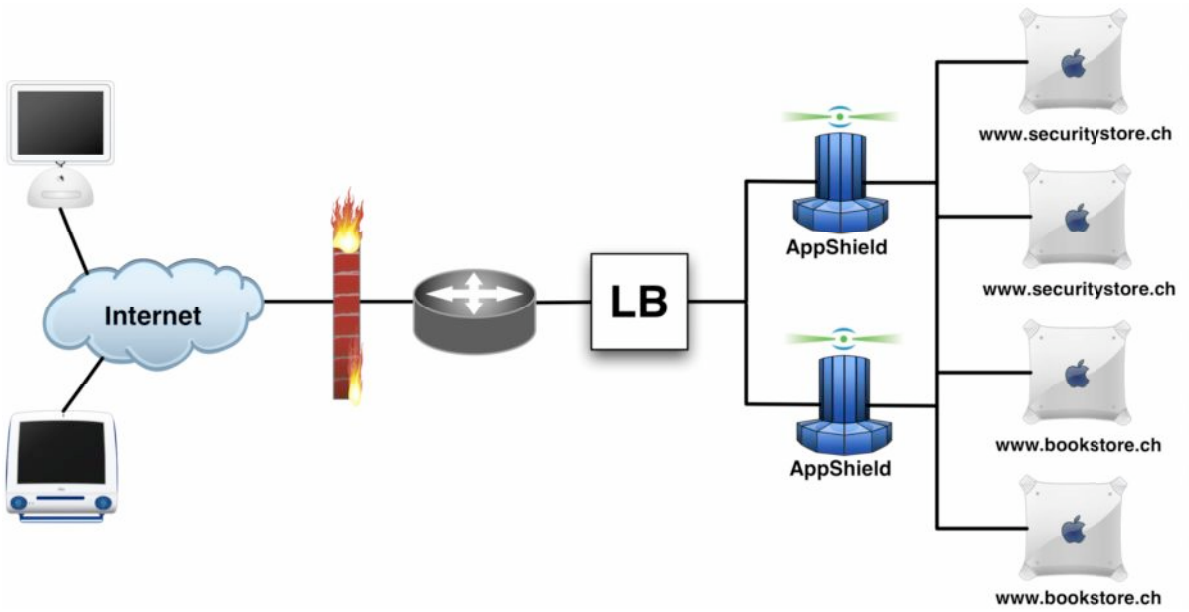


Figure 3.3-4 : configuration plusieurs à plusieurs

On parle de configuration "plusieurs à plusieurs", car plusieurs noeuds AppShield communiquent avec plusieurs serveurs. Pour la mettre en oeuvre, on désignera l'un des noeuds AppShield comme étant le serveur de configuration, puis on définira dans l'interface de configuration les adresses IP des différents serveurs Web et les adresses IP correspondantes à attribuer aux noeuds AppShield.

3.3.5. Configuration plusieurs à un

En ajoutant un second répartiteur de charge à la configuration précédente, on obtient une configuration plusieurs à un, recommandée pour un réseau comportant un grand nombre de noeuds AppShield et de serveurs Web. L'idée est de brasser deux fois le trafic : avant et après les noeuds AppShield.

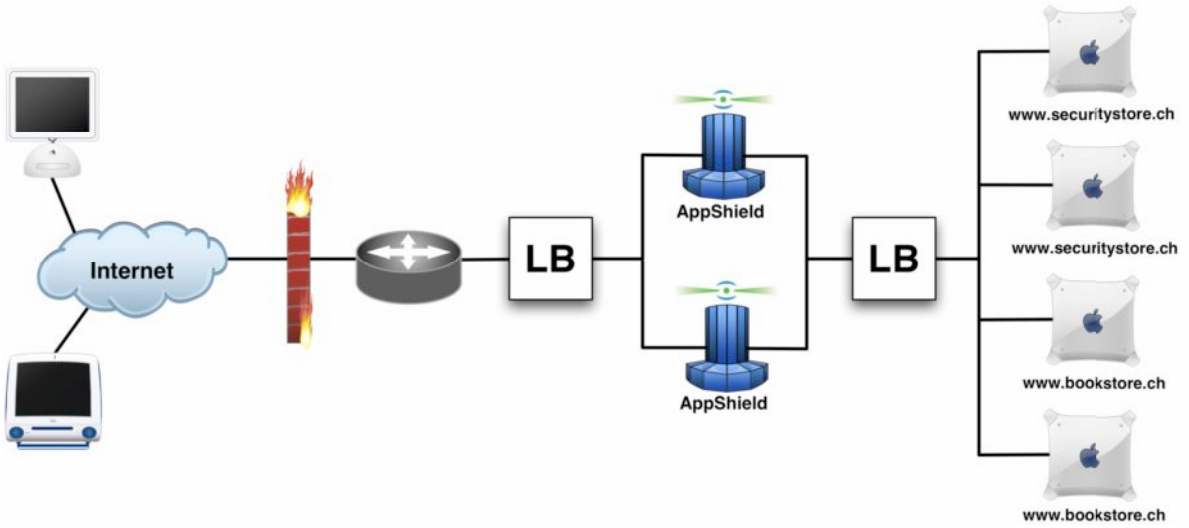


Figure 3.3-5 : configuration plusieurs à un

Ici encore, chaque noeud AppShield dispose d'une adresse IP par serveur. Le premier répartiteur de charge transmet la requête à l'un des noeuds AppShield, à l'adresse IP correspondant au serveur visé. Le noeud AppShield filtre la requête puis la transmet à l'adresse IP unique du second répartiteur de charge (c'est pour cela que l'on parle de configuration "plusieurs à un").

Le second répartiteur de charge utilise l'adresse IP du noeud AppShield dont il reçoit la requête pour déterminer à quel serveur la transmettre. Il peut aussi, et c'est préférable pour les sites avec session, utiliser l'adresse IP du client pour maintenir une association client-serveur. L'adresse du client est connue par un *cookie* dédié CN_SOURCE_IP généré par AppShield.

3.4. Configuration

Dans ce chapitre, nous passons en revue la configuration des différentes fonctions de AppShield, sans pour autant trop rentrer dans les détails. Le lecteur intéressé se reportera à la documentation électronique de AppShield (330 pages).

3.4.1. Réseau

L'onglet *Network* de l'outil de configuration permet de choisir un type de topologie (un à un, un à plusieurs, plusieurs à plusieurs, etc...), d'indiquer les adresses IP et/ou les noms d'hôte de chaque noeud AppShield et serveur Web (rappelons que la configuration est globale à tous les noeuds AppShield). On y définit également sur quels ports doivent écouter les noeuds AppShield, sur quels ports du serveur ils doivent renvoyer les requêtes, et quels ports doivent utiliser SSL.

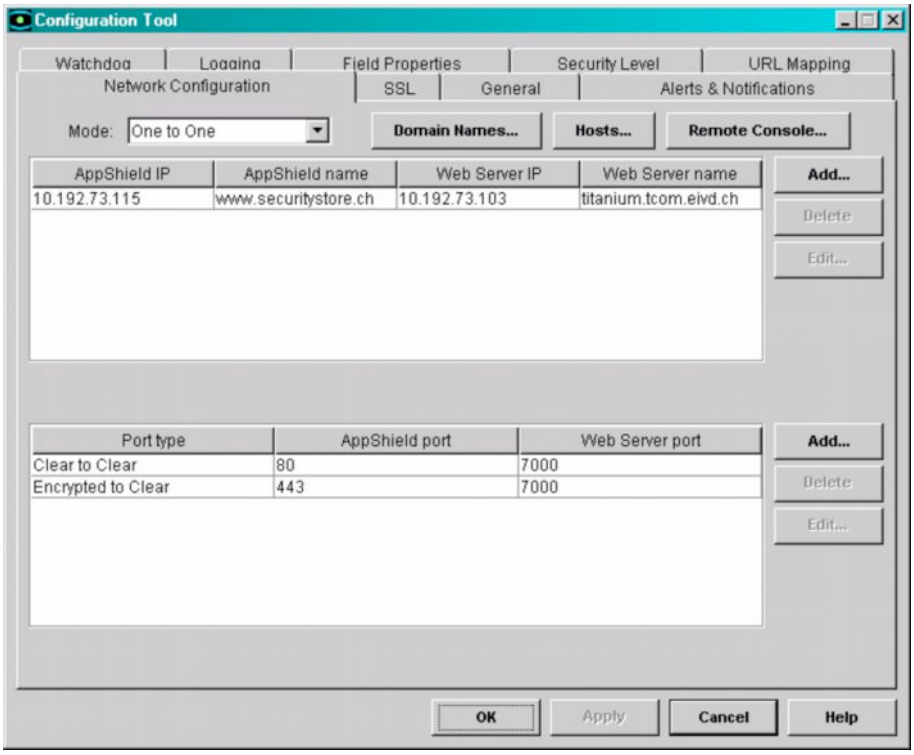


Figure 3.4-1 : panneau de configuration réseau

Les boutons *Domain Names* et *Hosts* permettent de définir quels sont les entêtes *Host* valides de la requête, afin d'éviter de renvoyer sur le serveur une requête qui ne correspondrait à aucun hôte virtuel configuré (et donc renverrait la page du serveur par défaut). Le bouton *Remote Console* permet de définir si l'on souhaite que l'interface de configuration emploie une connexion sécurisée et de limiter l'accès à certaines adresses IP de confiance à l'interface de configuration.

3.4.2. Mappage d'URL

L'onglet *URL Mapping* permet de définir les règles de réécriture d'URL. Cette fonction n'est pas à proprement parler celle d'une firewall applicatif mais plutôt celle d'un *reverse proxy*.

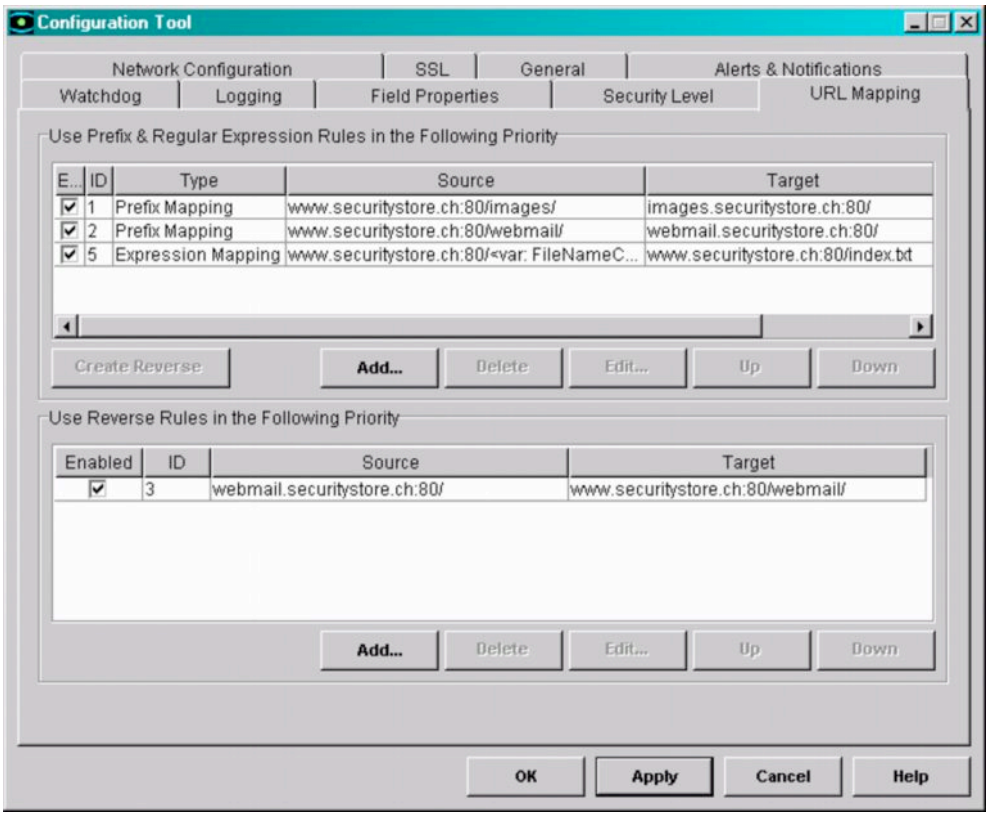


Figure 3.4-2 : panneau de configuration du mappage d'URL

Cette interface permet de définir 3 types de règles. Les règles *Prefix* et *Expression* sont listées dans la partie supérieure de la fenêtre, par l'ordre de priorité. Les règles *Reverse* dont listées dans la partie inférieure de la fenêtre, également par ordre de priorité.

Les règles *Prefix* servent à réécrire l'URL de la requête en substituant un préfixe de l'URL source à un autre pour composer l'URL cible. C'est l'équivalent de la directive *ProxyPass* avec mod_proxy.

Les règles *Expression* permettent de caractériser l'URL source au moyen d'une expression régulière et de remplacer toute l'URL par une autre URL fixe.

Les règles *Reverse* ont pour objectif de réécrire l'URL des entêtes *Location* lorsque le serveur renvoie un message de redirection au client pour lui signaler qu'un document a changé d'adresse. C'est l'équivalent de la directive *ProxyPassReverse* avec mod_proxy.

À la figure 3.4-2, nous avons défini 4 règles pour notre application *securitystore.ch*. En double-cliquant sur une règle, nous accédons à la fenêtre d'édition de la règle, qui est différente selon le type de règle.

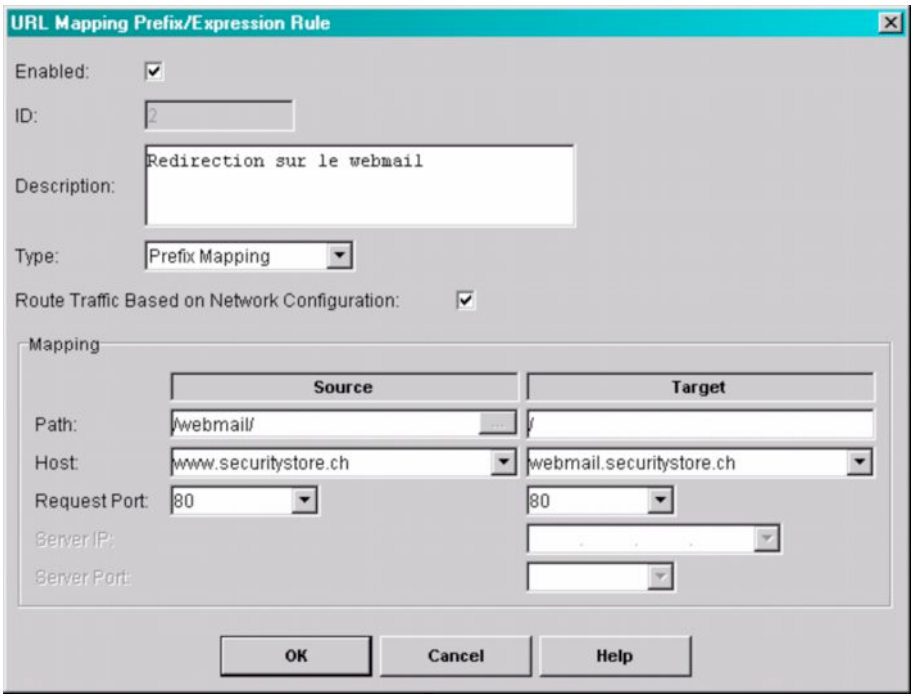


Figure 3.4-3 : édition d'une règle Prefix

Pour définir une règle *Prefix*, il indiquer l'hôte et le port source, le préfixe source, l'hôte et le port du serveur cible, le préfixe cible et éventuellement l'adresse IP du serveur cible.

À la figure 3.4-3, nous utilisons une règle *Prefix* pour réécrire dans l'URL de la requête le préfixe :

```
http://www.securitystore.ch/webmail
```

En :

```
http://webmail.securitystore.ch/
```

Ainsi, par exemple, si AppShield reçoit la requête :

```
GET  /webmail/imap/edit.cfm
Host:  www.securitystore.ch
```

Il réécrit l'URL et transmet la requête suivante au serveur :

```
GET  /imap/edit.cfm
Host:  webmail.securitystore.ch
```

Cette requête est envoyée à l'adresse IP du serveur `webmail.securitystore.ch` qui est obtenue par une résolution DNS, mais il est également possible de forcer AppShield à transmettre la requête à une adresse IP donnée sans tenter de résoudre le nom du serveur (utile si l'on est dans le cas d'une configuration "plusieurs à un" où AppShield est séparé du serveur par un répartiteur de charge).

Cette technique de réécriture d'URL permet de répartir une application Web sur plusieurs serveurs (non-miroirs) tout en donnant l'impression au client que l'application n'est constituée que d'un seul et même serveur.

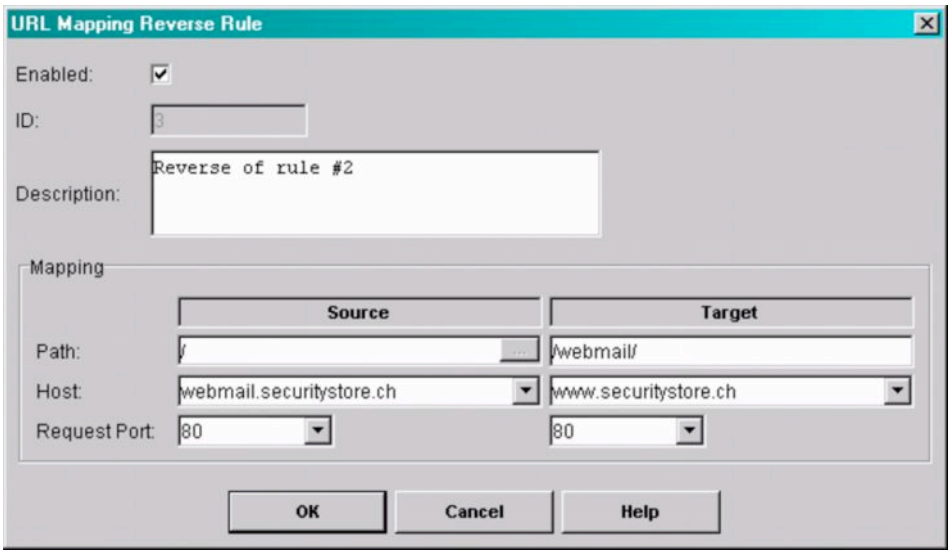


Figure 3.4-4 : édition d'une Reverse

Généralement, à chaque règle *Prefix* correspond une règle *Reverse*. Un bouton permet d'ailleurs de créer automatiquement une règle *Reverse* à *partir* d'une règle forward *Prefix*.

Le but d'une telle règle est de gérer la réécriture des URL de redirection, lorsque le serveur renvoie au client un code de statut de la série 3xx accompagné d'une entête *Location* indiquant qu'un document a été déplacé.

Avec la configuration ci-dessus, la réponse reçue du serveur par AppShield :

```
HTTP/1.1 200 OK
Location: http://webmail.securitystore.ch:80/adbook/index.cfm
```

Sera réécrite de la façon suivante avant d'être renvoyée au client :

```
HTTP/1.1 200 OK
Location: http://www.securitystore.ch:80/webmail/adbook/index.cfm
```

Cette fonction de AppShield est appréciable, car de nombreuses applications mal écrites effectuent des redirections à l'aide d'adresses absolues qui contiennent le nom interne du serveur, les rendant ainsi incompatibles avec la plupart des *reverse proxys*.

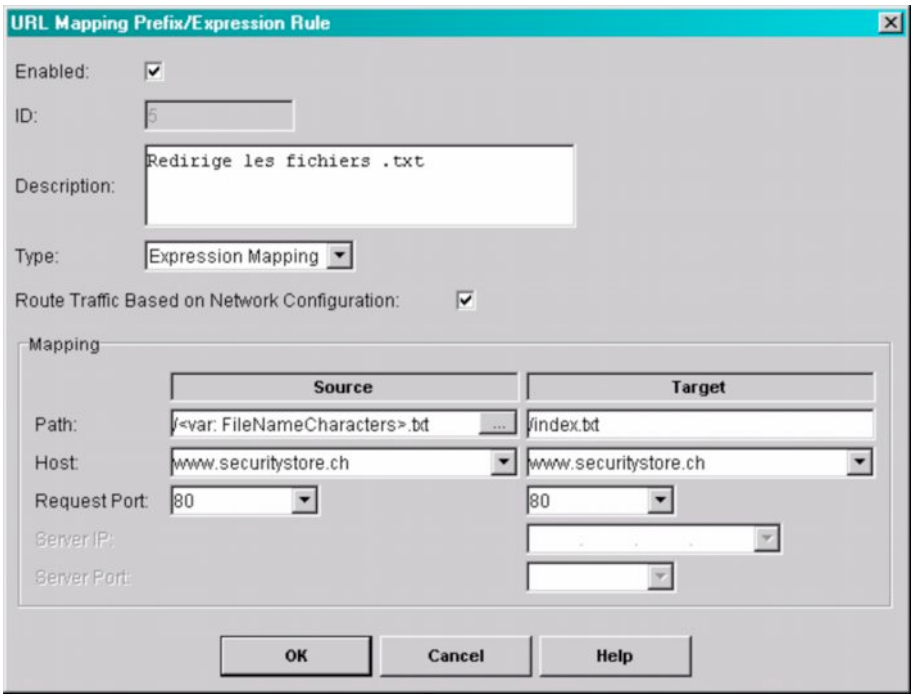


Figure 3.4-5 : édition d'une règles Expression

AppShield permet également de mapper les requêtes définies au moyen d'une expression en une requête fixe. Les expressions prennent la même forme que dans les règles d'affinement de la politique de sécurité (voir plus loin), c'est à dire que l'on peut utiliser aussi bien des *regexp* (expressions régulières) avec la syntaxe de EMACS que la syntaxe ASE propre à AppShield.

Dans l'exemple de configuration de la figure 3.4-5, chacune des requêtes :

```
http://www.securitystore.ch/toto123.txt
http://www.securitystore.ch/qwertz.txt
http://www.securitystore.ch/123456789.txt
```

Retournera au client un seul et même fichier :

```
http://www.securitystore.ch/index.txt
```

Cette réécriture se fait en interne au niveau du proxy : le client ne se rendra pas compte que sa requête est réécrite (l'URL affichée par son navigateur ne sera pas modifiée).

Par rapport aux possibilités de Apache et mod_rewrite en terme de réécriture d'URL avancée, AppShield paraît bien limité. En effet, il n'est pas possible d'utiliser une partie d'URL source dans l'expression de l'URL réécrite, ce qui limite l'intérêt de cette fonction.

Illustrons cette limitation de AppShield par un exemple concret. Nous souhaitons créeer une arborescence virtuelle d'URLs qui paraissent statiques afin de rendre l'application opaque et d'améliorer son indexation par les moteurs de recherche. Ainsi, la requête :

```
GET /news/2003/10/30
```

Doit être réécrite de manière interne en :

```
GET /news.cgi?year=2003&month=10&day=30
```

Avec mod_rewrite, le problème est simplement résolu en utilisant des expressions groupées :

```
RewriteRule  /news/([0-9]{4})/([0-9]{1,2})/([0-9]{1,2})/?
/news.cgi?year=$1&month=$2&day=$3   [L]
```

Lorsque l'on tente d'introduire dans la boîte de dialogue *URL mapping* de AppShield l'équivalent EMACS de cette expression (ici écrite en notation POSIX.2 adoptée par mod_rewrite), un message d'erreur apparaît signalant que le caractère "?" ne peut pas se trouver dans l'expression de l'URL cible. Même comportement en tentant d'échapper ce caractère avec un anti-slash.

La documentation de AppShield est plutôt vague sur le sujet, mais il apparaît que l'URL cible ne peut être qu'une expression statique. Excusons toutefois AppShield sur ce point dont la fonction première est de protéger une application Web et non de faire de la réécriture d'URL avancée.

3.4.3. SSL

Lorsqu'un client établit une liaison sécurisée par SSL avec un serveur Web, celui-ci lui transmet son certificat X.509 signé par une autorité de certification, qui atteste de l'identité de l'entreprise exploitant le serveur et de son CN (*Common Name*, le nom d'hôte du serveur). Cela permet de s'assurer que la phase d'échange des clés n'a pas été interceptée par un homme du milieu (*Man In The Middle*).

Si la connexion au serveur se fait en passant par un *reverse proxy*, alors le client a l'imprerssion que le *reverse proxy* est le serveur, et c'est le *reverse proxy* qui doit être le point terminal de la connexion SSL, qui héberge le certificat et la clé privée correspondante. Cette configuration est de plus attrayante dans le cas où l'application est composée de multiples serveurs : avec AppShield, les certificats et les clés privées résident à un seul endroit au lieu de chaque serveur, ce qui limite les risques et facilite la gestion.

AppShield supporte 4 modes de fonctionnement relativement à SSL :

- *Clear To Clear* — La communication arrive en clair sur le proxy et repart en clair, SSL n'est donc pas utilisé.
- *Encrypted to Clear* — La communication avec le client est sécurisée par SSL, mais la communication entre le proxy et le serveur passe en clair.
- *Clear to Encrypted* — Le proxy reçoit les requêtes en clair du client et les renvoie au serveur via un canal sécurisé par SSL.
- *Encrypted to Encrypted* — Le proxy reçoit les requête cryptées du client, les décrypte, les filtre et les recrypte avec un autre certificat pour les transmettre au serveur.

Dans la plupart des cas, on utiliser soit le mode *Clear To Clear* si l'application ne nécessite pas une confidentialité particulière des transferts ou *Encrypted To Clear* pour les échanges de données sensibles avec le client. Le cryptage entre le proxy et le serveur n'a de sens que si le réseau qui les sépare n'est pas de toute confiance, ce qui est rarement le cas puisqu'il s'agit généralement de communications locales à travers un firewall.

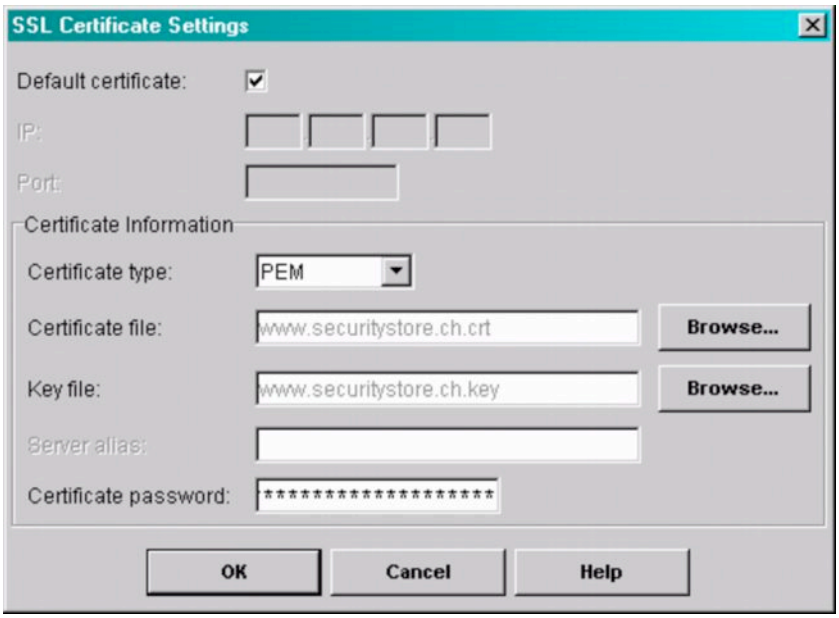


Figure 3.4-6 : installation d'un certificat SSL

Pour tester le bon fonctionnement de SSL entre le client et AppShield, nous avons demandé un certificat d'évaluation à validité limitée à l'autorité de certification de Verisign. Après avoir généré un demande de certificat (CSR) au moyen de l'outil intégré à AppShield (on aurait également pu le faire avec OpenSSL) et l'avoir transmise à Verisign, celui-ci nous retourne un certificat X.509 au format PEM, le format préféré des serveurs Netscape et supporté par AppShield.

L'interface de configuration illustrée à la figure 3.4-6 permet d'indiquer l'emplacement du fichier de certificat (*.crt), l'emplacement du fichier contenant la clé privée sous forme cryptée (.key) et le mot de passe à utiliser pour décrypter la clé privée qui, s'il n'est pas indiqué dans cette boîte de dialogue, sera demandé à chaque lancement de AppShield.

AppShield peut gérer ainsi un nombre quelconque de certificats, chacun d'eux pouvant être assigné à une application. On peut également définir un certificat par défaut qui sera utilisé pour les applications qui n'ont pas leur propre certificat défini.

Après avoir installé le certificat racine du CA dans la liste de confiance du navigateur Web du client, la communication HTTPS avec notre application Web *securitystore.ch* s'établit sans problème. Elle est cryptée entre le client et le proxy, elle passe en clair entre le proxy et le serveur (ce dernier n'a pas même mod_ssl activé).

3.4.4. Historique

Une fonction essentielle d'un *reverse proxy* est de pouvoir fournir un historique détaillé des transactions, afin de pouvoir étudier les tentatives d'attaque et la raison précise pour laquelle une requête a été refusée ou acceptée.

Chaque noeud AppShield mémorise dans sa base de données MySQL un historique des transactions. La taille de cet historique est configurable en nombre de transaction : lorsque la taille maximale est atteinte, les anciennes transactions sont écrasées. L'outil de configuration permet d'avoir une vue globale des transactions de toute l'application, ou de limiter l'affichage à un seul noeud, un seul serveur, une seule transaction ou un seul client au moyen de filtres.

Il est rarement intéressant de mémoriser l'historique des accès à des fichiers multimédias. Par fichiers multimédias, on entend des images JPEG, GIF, des fichiers vidéos ou sonores, des fichiers de feuilles de style, tous ces fichiers qui sont très présent dans l'application mais qui ne sont pas critiques du point de vue sécurité de l'application, car statiques. AppShield permet de définir les extensions des fichiers multimédias qui ne doivent pas être reportés dans l'historique, permettant ainsi un gain d'espace disque et de performance.

Une transaction est composée de une ou plusieurs sous-requêtes. Ainsi, la requête d'une page HTML et des images qui la composent font partie de la même transaction. Dans la fenêtre d'historique, AppShield affiche chaque transaction comme une seule ligne avec, dans une colonne, une indication du nombre de sous-requêtes. Cette visualisation est bien plus claire qu'un historique qui affiche une ligne par requête (il est tout de même possible d'afficher le détail des requêtes composant une transaction en double-cliquant dessus).

Les transactions refusées sont indiquées en rouge ainsi que la raison de ce refus (numéro d'une règle ou politique de sécurité par défaut, par exemple), comme illustré par la figure 3.4-7.

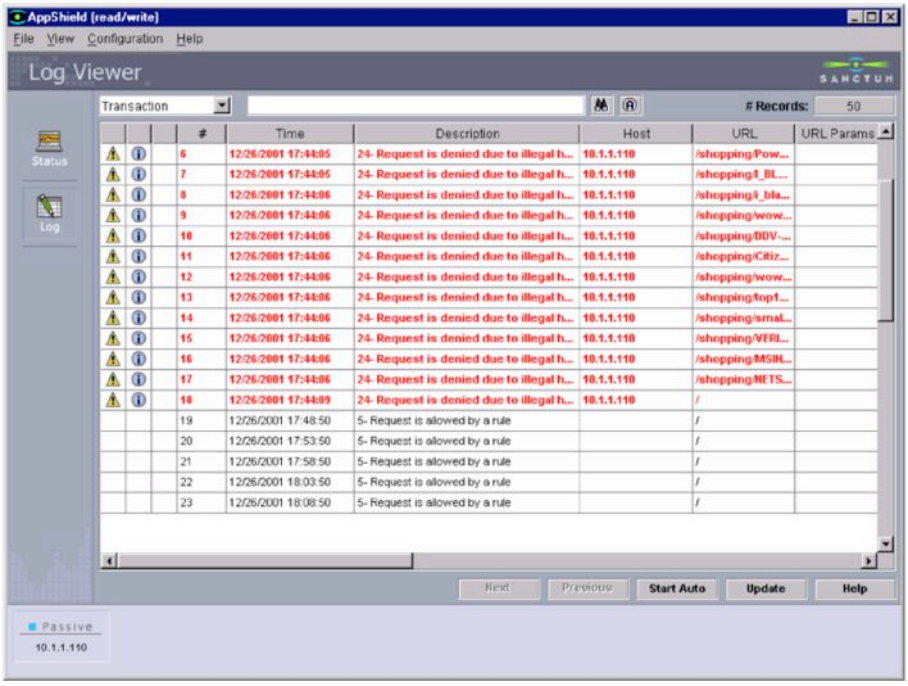


Figure 3.4-7 : visualisation de l'historique des transactions

En plus de son historique interne des transactions qu'il est capable d'exporter dans divers formats pouvant être importés dans un tableur ou une base de données tierce, AppShield offre la sortie dans un fichier texte d'un historique HTTP dans l'un des formats IIS, W3C ou NCSA, qui détaillent une requête sur chaque ligne. Ces formats standards sont reconnus par des outils de statistiques comme Analog ou AWStats.

3.4.5. Niveaux de sécurité

AppShield est un firewall de type *White List*, c'est à dire que par défaut il bloque un grand nombre de requêtes qu'il trouve suspectes et qu'il faut définir des règles d'affinement propres à l'application pour laisser passer certaine requêtes qui ressemblent à des attaques pour AppShield. Ce mode de fonctionnement nécessite un effort de configuration au départ, ce qui n'est pas au goût de tout le monde, surtout si l'on se satisfait d'un niveau de sécurité moyen.

Depuis sa version 4.0, AppShield offre donc le choix entre 3 niveaux de sécurité par défaut. Si le niveau de sécurité est élevé, il faudra définir un grand nombre de règles d'affinement pour que l'application puisse fonctionner. Si le niveau de sécurité est faible, presque aucune règle ne sera nécessaire. Le niveau de sécurité intermédiaire est un compris entre ces extrêmes.

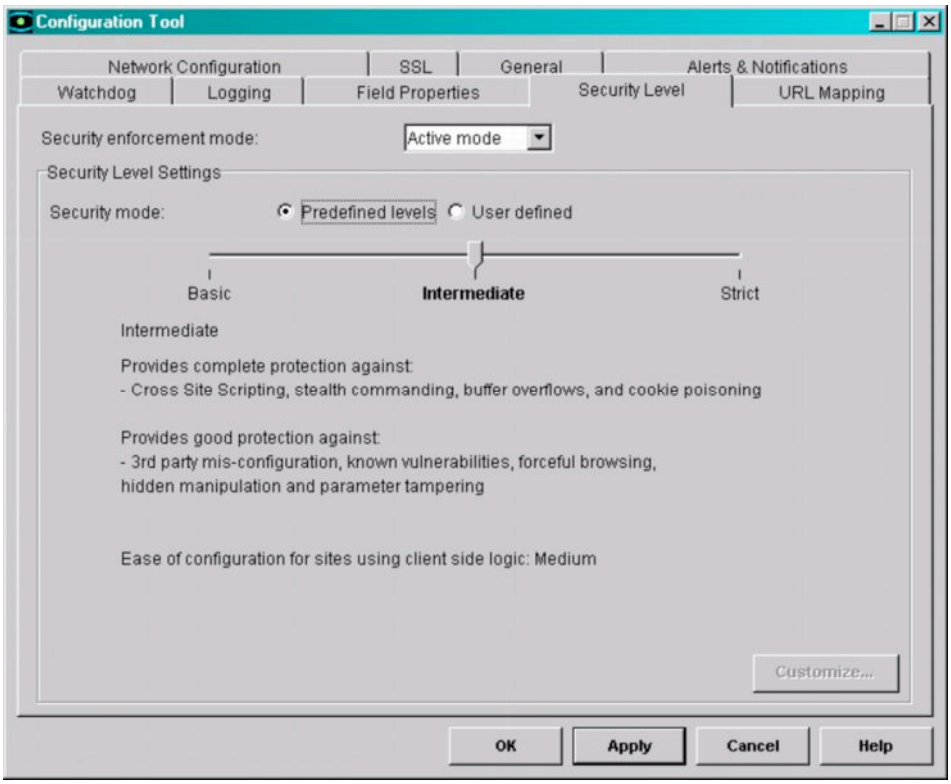


Figure 3.4-8 : réglage du niveau de sécurité par défaut

Le niveau de sécurité intermédiaire a été utilisé pour nos tests avec *securitystore.ch*. Comme indiqué dans la fenêtre de configuration, il protège complétement l'application contre des attaques *Cross Site Scripting*, *buffer overflows* et *cookie poisoning*. Il offre une protection partielle contre les manipulations de paramètres et les vulnérabilités propres à l'application.

Si les 3 niveaux prédéfinis de sécurité ne suffisent pas, un mode avancé "*User defined*" est disponible dans lequel on peut choisir précisément ce que AppShield doit contrôler et contre quels genres d'attaques il doit protéger.

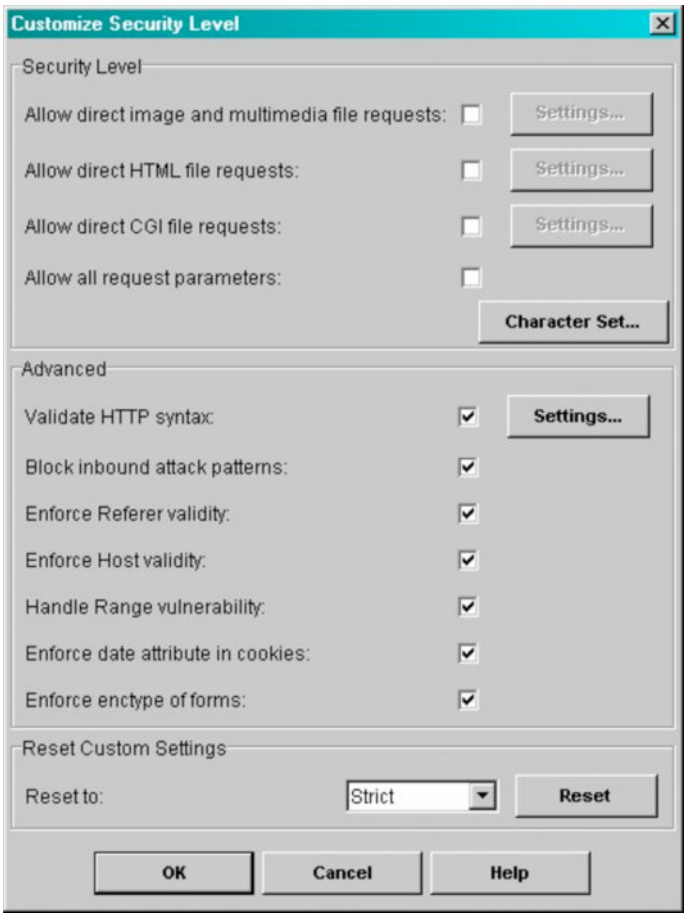


Figure 3.4-9 : personnalisation du niveau de sécurité

Cette fenêtre permet de personnaliser le comportement de AppShield. On a le choix d'autoriser ou non implicitement l'accès aux documents HTML, CGI ou aux fichiers multimédias (images). Les suffixes correspondant à chacun de ces types de documents peuvent être librement définis.

Validate HTTP Syntax permet de rejeter les requêtes HTTP mal formées.

Block inbound attack patterns recherche la présence de motifs d'attaque dans la requête du client.

Enforce Referer Validity s'assure que l'entête *Referer* d'une requête correspond bien à la page retournée qui contenait le lien au document demandé.

Enforce Host validity refusera les requêtes dont le champ *Host* (nom de l'hôte virtuel) ne correspond pas à un hôte définit dans le panneau *Network* de AppShield.

Handle Range vulnerability vérifie que l'entête *Range* de la requête corresponde à une entête *Accept-Range* précédemment renvoyée par le serveur. Les entêtes *Range* permettent de ne télécharger qu'une partie d'un document.

Enforce date attribute in cookies refuse les cookies qui ont expiré et vérifie qu'un cookie transmis par le client aie bien été envoyé précédemment par le serveur.

Enforce enctype of forms vérifie que les formulaires utilisent bien le type d'encodage défini dans l'attribut *enctype* de la balise <form> et qu'il corresponde à un type d'encodage accepté par l'application.

3.4.6. Règles d'affinement de sécurité

Si l'on constate, après avoir avoir configuré un niveau de sécurité, que l'application Web n'est pas entièrement fonctionnelle à travers le proxy, que AppShield affiche des messages d'alerte de sécurité et que certaines lignes de l'historique des transactions sont en rouge, cela signifie que l'application nécessite des règles d'affinement de sécurité.

Ces règles d'affinement de sécurité (*Security refinement rules* en anglais) permettent de définir des exceptions à la politique de sécurité par défaut. Plus le niveau de sécurité par défaut est élevé, plus il faudra configurer de règles.

Heureusement, AppShield nous aide dans ce travail en offrant une fonction permettant de générer une règle à partir d'une ligne de transaction refusée de l'historique. Si AppShield doit être configuré dans un environnement en production, le mieux est de faire travailler le firewall en mode passif et de définir les règles une par une jusqu'à ce qu'il n'y aie plus de lignes rouges qui apparaissent dans l'historique des transactions. On peut alors basculer le firewall en mode actif.

Lorsque AppShield renvoie une page HTML au client, il la scanne et mémorise tous les liens qu'elle contient, les formulaires, les fichiers inclus comme les images, les cookies. Il utilise ces informations pour dynamiquement générer des règles positives (*allow*) qui ne s'appliquent qu'au seul client qui a reçu la page. Pour cette raison, AppShield peut être qualifié de firewall applicatif *stateful*.

Si le client tente de modifier un champ de formulaire HIDDEN, AppShield va s'en rendre compte et bloquer la requête. De même en cas d'empoisonnement de cookie : AppShield sait très bien à quel client il a transmis quelle entête *Set-Cookie*, et vérifie que l'entête *Cookie* de la requête du client corresponde.

Cette technique de génération dynamique de règles assure un bon niveau de sécurité et limite en même temps le nombre de règles d'affinement nécessaires. On peut se contenter de définir les points d'entrée dans l'application, et AppShield fera le reste en observant les pages retournées.

Il existe cependant des cas que AppShield ne peut pas traiter : si du code mobile (JavaScript par exemple) génère du code HTML avec des liens côté client, AppShield sera incapable de le deviner et il faudra l'indiquer manuellement au moyen de règles d'affinement. De même si le code côté client génère dynamiquement des cookies.

Il existe 3 types de règles d'affinement :

Les règles Allow Request sont utilisées pour autoriser des requêtes qui seraient refusées autrement par AppShield. La syntaxe permet de définir le gabarit de tout type de requête, qu'elle soit très généraliste ou très spécifique.

Les règles Allow Request Modification sont utilisées pour autoriser l'utilisateur ou un script côté client à modifier certains paramètres par rapport à la version statique transmise dans la page transmise au client, des modifications qui seraient sinon refusées par AppShield.

Les règles Cookie sont utilisées pour autoriser des cookies générés par l'utilisateur, c'est à dire les cookies qui ne proviennent pas d'une entête *Set-Cookie* préalablement renvoyée au client.

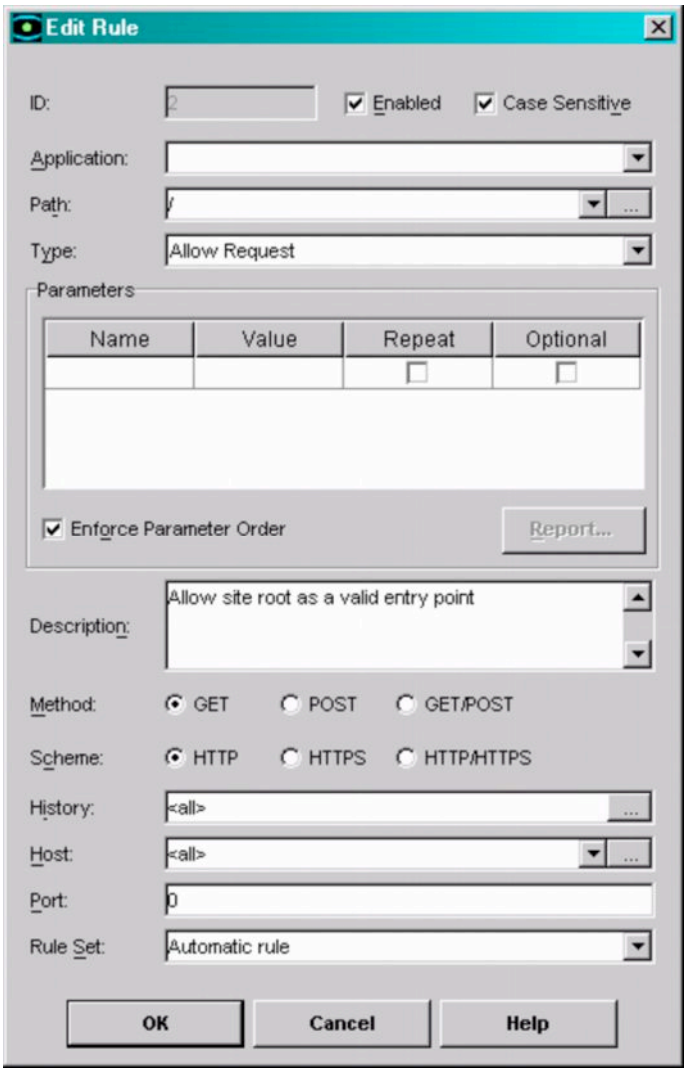


Figure 3.4-10 : édition d'une règle d'affinement Allow Request

La boîte de dialogue de la figure 3.4-10 est utilisées pour éditer des règles d'affinement.

Les champs *Application* et *Path* permettent de restreindre le contexte de la règle (un certain répertoire d'une certaine application).

Le champ *Type* permet de choisir entre les 3 types de règles précités.

La zone *Parameters* sert à définir quels sont les paramètres autorisées par la requête (leur nom, leur ordre et leur valeur).

Method permet d'indiquer si la règle s'applique uniquement à la méthode POST, GET ou aux deux.

Scheme définit si la règle s'applique au protocole HTTP, HTTPS ou aux deux.

History permet d'indiquer quelles sont les pages de l'application qui doivent avoir été visitées par le visiteur durant sa session avant d'arriver sur cette page. On peut indiquer une ou plusieurs URI dans ce champ, et l'on peut même forcer un certain ordre de visite.

Host et *Port* permettent de définir une règle qui ne s'applique qu'à un hôte et un port donné (rappelons que AppShield peut être utilisé pour gérer le trafic de multiples applications à la fois).

Rule Set permet de sélectionner le groupe de règles à laquelle est associée la règle. On peut ainsi activer/désactiver rapidement toutes les règles d'un groupe.

3.4.7. Syntaxe des expressions

AppShield permet de définir et nommer des ensembles de caractères qui reviennent de façon récurrente dans les champs de formulaires ou les URL, et ensuite de les utiliser dans une règle d'affinement de la politique de sécurité ou de réécriture d'URL. Cela est commode pour définir, par exemple, quels sont les caractères autorisés dans un nom de fichier ou dans un champ de mot de passe.

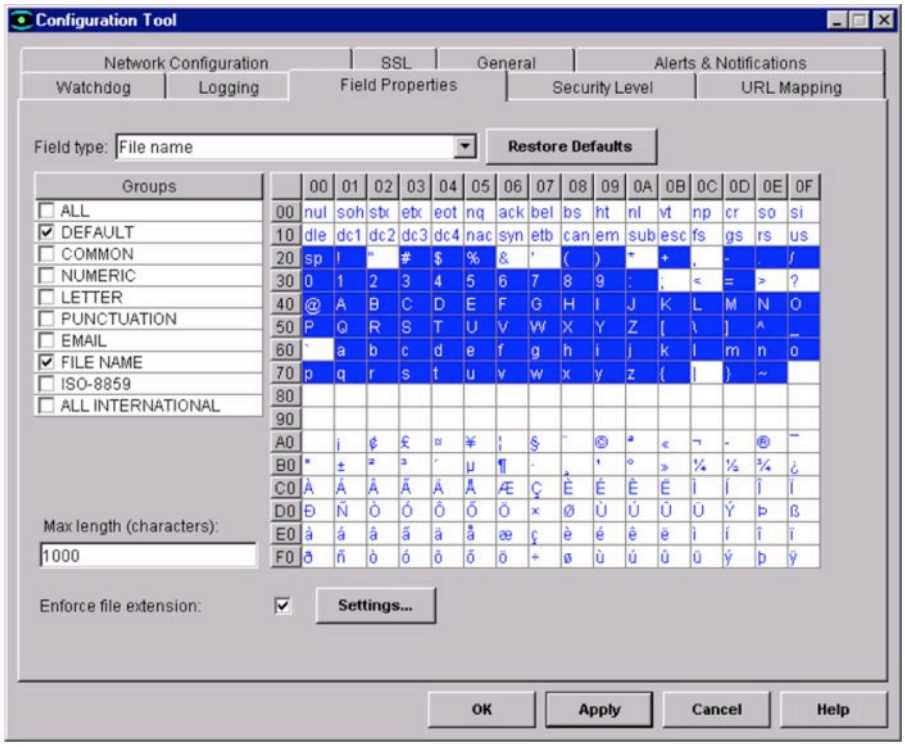


Figure 3.4-11 : définition de jeux de caractères

L'interface présente à l'utilisateur une grille contenant tout l'alphabet ASCII étendu ou Unicode (AppShield est compatible avec les langues latines et asiatiques). Il peut sélectionner les caractères qui sont autorisés pour ce champ. Le jeu de caractère se voit attribuer un nom (*Filename* dans notre exemple) pour pouvoir l'utiliser facilement dans les règles. Il est également possible de borner la longueur du champ en nombre de caractères. La partie gauche de la fenêtre présente une liste de groupes de caractères prédéfinis, pour plus de commodité.

Lors de la création d'une règle de mappage d'URL basée sur une expression ou d'une règle d'affinement, AppShield utilise une sytaxe propre pour définir le gabarit de chaînes de caractères : la syntaxe ASE (*AppShield Syntax Expressions*). Cette syntaxe ressemble aux expressions régulières, mais est plus simple à utiliser. Nous présentons quelques rudiments de cette syntaxe ci-après.

`<rng:>` Définit un **intervalle de caractères**, par exemple `<rng:a-z123>` qui définit toutes combinaison de lettres minuscules et des chiffres 1 à 3.

`<num:>` Définit un **intervalle de nombre entiers**, par exemple `<num:0-255>`.

`<len:>` Limite **la longueur en caractères** d'une valeur. Par exemple l'expression `<len:10-20 rng:a-z>` définit un champ pouvant contenir entre 10 et 20 lettres minuscules.

`(x-y)` Borne la **longueur totale d'un champ**, c'est à dire aussi bien les parties statiques que les parties dynamiques du champ en plaçant un intervalle entre parenthèse à la fin de l'expression. Par exemple `/images/<rng:a-z>.gif(20-30)` décrit un chemin d'accès à un fichier GIF comportant au moins 20 et au plus 30 caractères de longueur totale.

`<reg:>` Permet d'utiliser une **expression régulière** en notation EMACS à l'intérieur d'une expression ASE. Par exemple `<reg:"[a-z]*">` qui est l'équivalent de `<rng:a-z>`.

Ces exemples ne sont pas exhaustifs, on se reportera à la documentation de AppShield pour plus de détails sur les possibilités de la syntaxe ASE.

3.4.8. Alertes de sécurité

Comme tout le trafic de l'application transite par AppShield, la sécurité d'un noeud AppShield et sa disponibilité sont des points critiques. C'est pourquoi Sanctum a inclu un *watchdog* (chien de garde) à son produit qui vérifie à intervalles réguliers le bon fonctionnement du noeud AppShield, et avertit l'administrateur par e-mail ou pager le cas échéant.

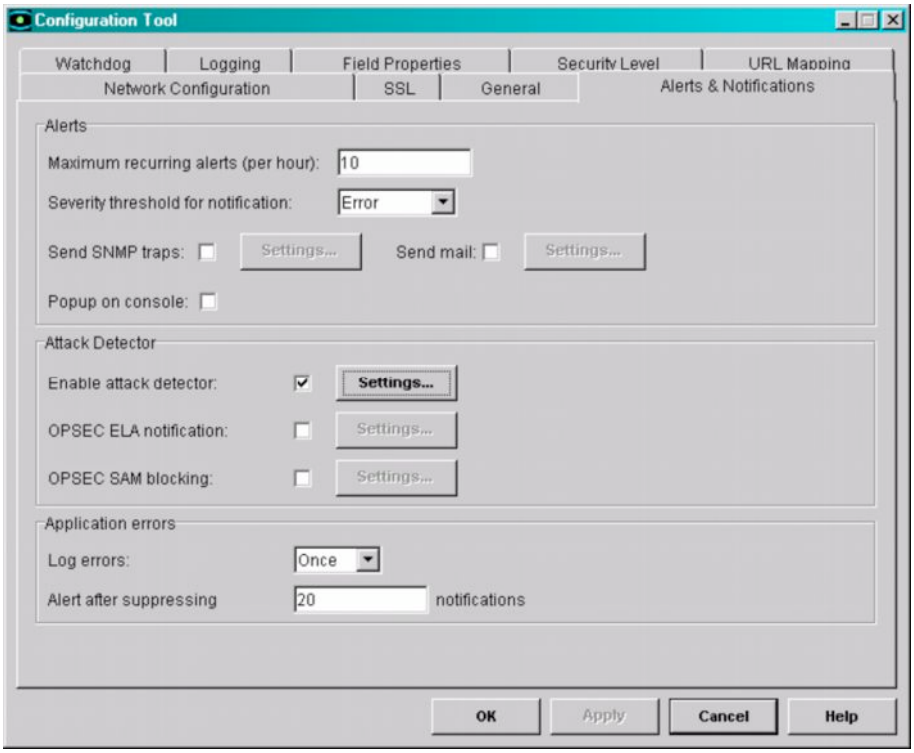


Figure 3.4-12 : configuration des alertes et avertissements

Cette boîte de dialogue permet à l'administrateur de définir à partir de combien de messages et de quel niveau d'importance il souhaite être alerté, et par quel moyen (popup dans la console, e-mail, trap SNMP). OPSEC, nouveauté de cette version 4.0, est un protocole permettant à AppShield de communiquer avec d'autres équipements de sécurité, et notamment de notifier un firewall *Check Point* d'une tentative d'attaque afin qu'il bloque l'IP de l'attaquant au niveau réseau.

3.4.9. Test de fonctionnement

Pour tester AppShield, nous avons utilisé une application Web *securitystore.ch* développée dans le cadre du travail de semestre en PHP et MySQL, tournant sur Apache et Mac OS X, et elle-même volontairement vulnérable à toutes sortes d'attaques (Cross Site Scripting, SQL Injection et manipulation de cookies notamment).

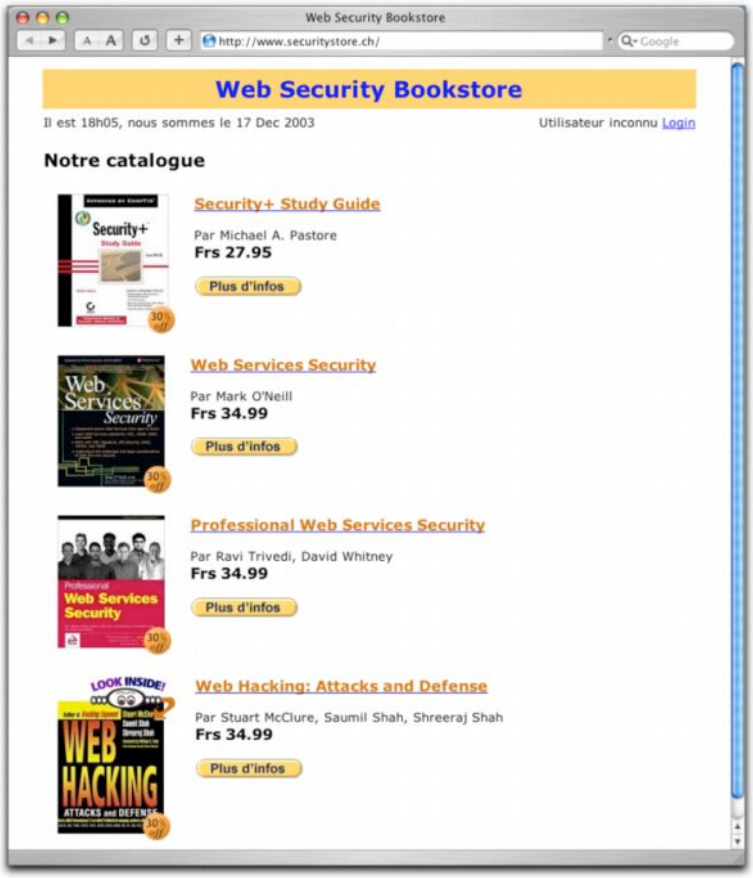


Figure 3.4-13 : application Web d'exemple securitystore.ch

Bien que l'application utilise bon nombre d'artifices de HTTP (redirections par code de statut 3xx, paramètres GET et POST, cookies), elle est écrite de manière standard et n'a pas posé de problème de compatibilité avec AppShield.

Il a fallu définir une règle de URL mapping de type *forward prefix* qui redirige une requête entrante du port 80 de AppShield sur le port 80 du serveur Web, ainsi que la règle *reverse* correspondante pour gérer les redirections utilisant des URLs absolues.

Le niveau de sécurité par défaut a été réglé sur intermédiaire. En l'état, AppShield refusait l'accès à l'application, mais il a suffit de définir une règle autorisant l'accès à la racine de l'application (point d'entrée sur le site) pour que le reste de l'application fonctionne correctement, AppShield scrutant les pages retournées au client à la recherche de liens hypertextes ou de formulaires HTML. Tous les autres paramètres étaient la configuration par défaut de AppShield 4.0.

Toutes les attaques tirant parti de vulnérabilités connues de l'application ont été déjouées haut la main par AppShield, alors que ces mêmes attaques aboutissaient en accédant directement à l'application sans passer par le *reverse proxy*. Lorsque AppShield bloque, une requête, il retourne au client un message d'erreur standard comportant le logo de Sanctum et une explication de l'erreur (figure ci-après), mais l'administrateur peut lui substituer une page d'erreur personnalisée située sur le serveur, statique ou dynamique, afin de mieux l'intégrer dans l'application et ne pas

révéler la présence de AppShield au client. Si la page est dynamique, AppShield lui transmet en paramètres des informations sur la requête d'origine et la raison du refus, que le script peut choisir ou non d'afficher.

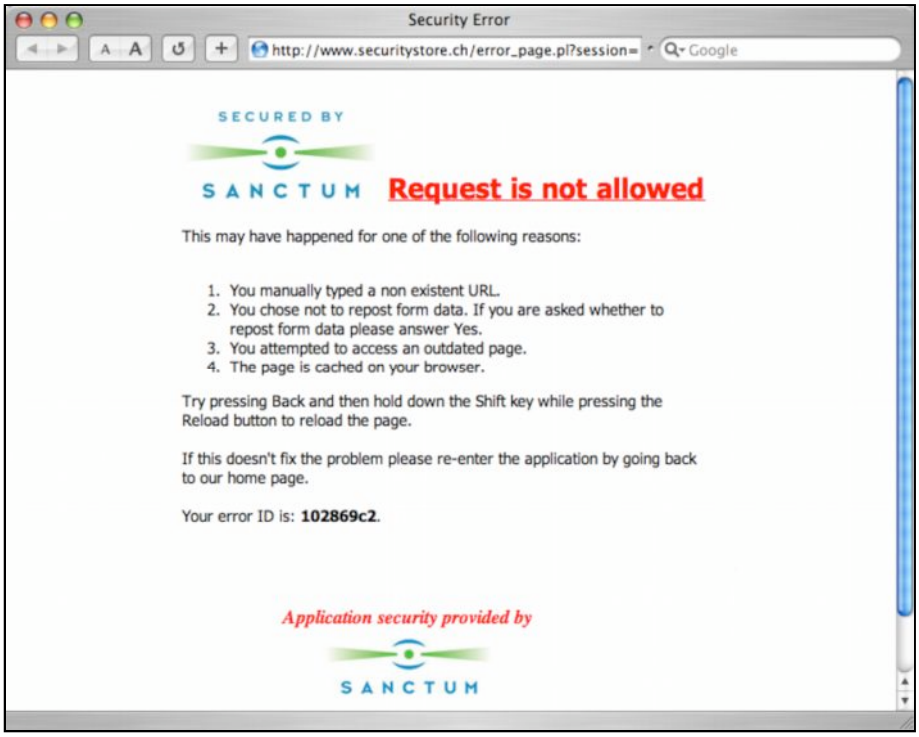


Figure 3.4-14 : page d'erreur par défaut de AppShield

On peut certes regretter que AppShield n'aie pas été mis à l'épreuve d'un vrai scanner de vulnérabilités (je n'en disposais d'aucun lors de mes tests de AppShield courant octobre 2003), mais on devine qu'un produit de cette renommée n'aurait pas laissé passer beaucoup d'attaques.

3.5. Conclusion

AppShield est une produit très abouti, sans aucun doute le haut de gamme de ce qui se fait en matière de sécurité des applications Web. Sa politique de tout bloquer par défaut (*white list*) peut sembler contraignante, mais grâce à la génération dynamique des règles, à un mode d'auto-apprentissage et à la possibilité d'affiner manuellement les règles générées, la configuration se passe sans problème pour la plupart des applications Web.

Le but visé par cette étude de AppShield n'était pas tant de trouver les qualités et défauts du produit (il aurait fallu pour cela le comparer à des produits concurrents, ce qui aurait réclamé beaucoup de temps), mais plutôt de s'ouvrir l'esprit sur le genre de fonctionnalités que peut offrir un firewall applicatif professionnel en vue du développement de notre propre solution de firewall applicatif lors de la seconde partie de ce travail de diplôme.

4. Développement de ProxyFilter

Le domaine des firewalls applicatifs HTTP est relativement nouveau, et s'il existe plusieurs solutions sur le marché (dont l'une d'elle, le produit AppShield de Sanctum, vient d'être étudiée), presque toutes sont commerciales. Les prix élevés des licences les réservent encore à de grosses structures, comportant souvent des dizaines de serveurs et drainant un trafic élevé.

La deuxième partie de ce travail de diplôme s'intéresse au développement d'un firewall applicatif HTTP entièrement basé sur des technologies *open source* (Apache et Perl) et lui même également *open source*. S'il ne concurrence pas les solutions commerciales à plusieurs dizaines de milliers de francs, il offre l'essentiel des fonctionnalités utiles à la protection d'une application Web.

4.1. Caractéristiques

- Programmé en Perl, intégration dans Apache 1.3.x au travers de mod_perl
- Supporte le fonctionnement multithread de Apache
- Syntaxe de configuration basée sur XML
- Filtrage générique d'URLs à l'aide d'expressions régulières compatibles Perl
- Réécriture d'URL émulant le fonctionnement de *ProxyPass* et *ProxyPassReverse*
- Filtrages des entêtes HTTP de la requête et de la réponse
- Filtrage du type MIME des documents (entêtes *Content-Type*)
- Filtrage des méthodes HTTP (GET, POST, HEAD, etc...)
- Filtrages des cookies en entrée et en sortie (entêtes *Cookie* et *SetCookie*)
- Vérification de la longueur et syntaxe des paramètres de scripts (URLs et données POST)
- Utilisation d'un fichier *charsets* pour mémoriser les jeux de caractères souvent utilisés
- Log détaillé avec plusieurs niveaux de verbosité, dans le log de Apache ou dans un fichier externe
- Supporte un fonctionnement mixte en mode *White List* (inclusif) ou *Black List* (exclusif)
- Compatible SSL (HTTPS) avant et après le proxy

4.2. Composants logiciels

4.2.1. Apache

Le serveur Web de Apache n'a plus à faire ses preuves : issu des efforts du NCSA et du CERN qui ont chacun développé leur propre serveur Web de leur côté, il détient aujourd'hui plus de 50% du marché mondial des serveurs Web, et a l'avantage d'être très extensible et gratuit. Son API (*Application Programming Interface*) ouverte aux programmeurs Perl et C permet aux développeurs de modules de personnaliser le comportement du serveur pour développer de nouvelles applications ou protocoles (WebDAV, authentification par LDAP ou SQL, négociation de contenu multilingue, utilisation de *fingerprints* pour déterminer le type MIME d'un document, limitation de la bande passante, quotas de transfert, etc...). Des centaines de modules ont déjà été développés pour Apache, la plupart sont gratuits et *open source*, tout comme le serveur Apache lui-même.

4.2.2. Perl

Le cahier des charges précise que le firewall applicatif doit être implémenté sous forme de module Apache (et non comme un CGI), ce qui est dicté pour d'évidentes raisons de performances : un module Apache est plusieurs fois plus rapide que son équivalent en script CGI, car il est compilé et préchargé en mémoire au lancement du serveur Web, alors qu'un CGI nécessite de relancer un interpréteur (une commande du système hôte) à chaque nouvelle requête, technique qui offre une très mauvaise montée en charge.

Seuls deux langages s'offrent à nous pour le développement d'un module Apache : Perl et C. Dans la mesure où l'ouvrage *Programming Apache Modules With Perl & C* dispense la plupart de ses exemples en Perl, que les auteurs de cet ouvrage encouragent l'usage du Perl et que je n'avais pas plus de connaissances préalables en C qu'en Perl, le développement de *ProxyFilter* s'est fait en langage Perl.

Perl est un langage de script évolué fonctionnant en mode interprété et disponible sur une large variété de plateformes (Unix, Windows, Mac, BeOS, Amiga, etc...). Il est prévu pour être très souple dans sa syntaxe, de sorte qu'il existe systématiquement plusieurs manières d'écrire les mêmes instructions. Ces raccourcis de syntaxe peuvent rendre difficile la compréhension du code écrit par quelqu'un d'autre, de sorte que l'on peut dire que Perl est un langage conçu dans l'optique du programmeur expérimenté avare sur le nombre d'instructions, plutôt que dans celle de l'étudiant débutant en Perl.

Comme tout langage généraliste, Perl a cependant son domaine de prédilection : il est adapté pour des programmes effectuant 80 à 90% de traitement de données (*parsing* d'un document, filtrage de flux de données) grâce à l'intégration et la puissance des expressions régulières. Au niveau du Web, Perl est l'un des langages préférés pour développer des scripts CGI, de nombreux modules lui permettent de générer du code HTML d'une façon orientée objet, de gérer des formulaires, établir des connexion à des bases de données ou générer dynamiquement des images, par exemple.

Pour le développement de modules Apache, Perl offre l'avantage de ne pas avoir à recompiler l'exécutable du serveur à chaque modification du code source. Avec Perl, il suffit de relancer le serveur pour que les modifications soient prises en compte, et encore cela n'est pas toujours nécessaire selon la configuration de mod_perl. De plus, comme Perl est un langage de plus haut niveau que C, le développement est plus rapide, et il est moins vulnérable aux attaques de type *buffer overflows* que le langage C, puisque Perl fait l'abstraction des adresses mémoire. L'inconvénient par rapport à C est une petite perte de performances, mais les possibilités d'un module Perl par rapport à l'API de Apache sont sensiblement les mêmes que celles d'un module écrit en C.

4.2.3. mod_perl

Mod_perl est un interpréteur Perl intégré à Apache, il permet de faire le lien entre l'installation de Perl du système hôte et le serveur Apache. Mod_perl n'est pas strictement nécessaire à l'exécution de simples CGIs écrits en Perl, mais son utilisation apporte un gain de performance important en préchargeant et précompilant les scripts Perl au lancement du serveur, et il met à disposition un API Apache pour Perl permettant au développeur de personnaliser le fonctionnement intrinsèque du serveur, beaucoup plus que ne le peut un script CGI qui communique avec le serveur uniquement au moyen de l'entrée/sortie standards et de variables d'environnement.

4.2.4. XML

XML (*Extensible Markup Language*) est un langage à balises dérivé de SGML permettant de structurer des données sous une forme arborescente. À la différence de HTML, XMl permet de définir ses propres balises et attributs, ce qui permet de l'employer dans des domaines très divers, non-limités à l'écriture de pages pour le Web. À titre d'exemple, le système d'exploitation Mac OS X mémorise les préférences de toutes ses applications sous une forme XML (fichiers .plist), qui ont ainsi l'avantage d'être lisibles et éditables par l'utilisateur.

Plutôt que d'opter pour une syntaxe "maison" pour le fichier de configuration de *ProxyFilter*, qu'il faut *parser* soi-même et dans laquelle les erreurs de syntaxe sont difficiles à déceler, il était logique d'opter pour le format XML, d'autant que la description arborescente de la structure d'une application Web s'y prette bien.

4.2.5. Expat

Un avantage d'utiliser le format XML est qu'il existe des *parsers* largement éprouvés pour ce format. On distingue deux types de *parsers* : ceux travaillant en mode flux et ceux travaillant en mode arbre.

Les *parsers* en mode flux ont l'avantage de la rapidité. Ils fonctionnent au moyen de *handlers* qui sont appelés dans l'ordre où les différents éléments du fichier source sont rencontrés. Le programmeur doit lui-même garder la trace des éléments rencontrés, ce qui a pour effet de complexifier le code du programme.

Les *parsers* en mode arbre (DOM pour *Document Object Model*) retournent en une seule étape un arbre du document. Ils sont un peu plus lents que les *parseurs* en mode flux, mais le parcours du document en est facilité.

Pour lire la syntaxe de configuration de *ProxyFilter*, le choix s'est porté sur le parseur Expat, écrit en C, qui peut être utilisé depuis un programme Perl au moyen du module XML::Parser. Il travaille essentiellement en mode flux, ce qui est adapté étant donné la relative simplicité du fichier de configuration et le besoin de performances du firewall. Il existe également des *parsers* écrits entièrement en Perl, mais leurs performances en terme de vitesse sont loin d'égaler celles de Expat.

4.2.6. LWP

Un (reverse) proxy joue à la fois le rôle de serveur et de client HTTP : il reçoit des requêtes du client à la façon d'un serveur Web, la filtre et effectue une requête auprès du serveur final à la façon d'un client Web. Pour gérer la partie cliente de *ProxyFilter*, mod_proxy s'est vite avéré trop limité (voir les explications plus bas), et l'on s'est donc tourné vers le module LWP::UserAgent qui fait partie de la distribution de libwww-perl, une collection de modules fournissant des APIs pour le protocole HTTP à Perl. Supportant HTTP et HTTPS, LWP permet de définir précisément les entêtes et l'URL de la requête, et retourne un objet contenant toutes les indications utiles sur la réponse du serveur.

4.3. Processus de développement

Fort de l'expérience acquise lors de la découverte de AppShield, il devenait un peu plus simple de répondre aux questions "que doit-on filtrer ?" et "comment doit-on filtrer ?". Le développement du module *ProxyFilter* n'en a pas moins été un long parcours semé d'embuches...

4.3.1. Apprentissage du langage Perl

La première (longue) étape a été l'apprentissage du langage Perl. La lecture de l'ouvrage de référence *Programming Apache Modules With Perl And C* m'a appris que le langage à choisir pour développer rapidement des modules Apache était Perl, sauf si des impératifs de performance dictaient le contraire et donc mon choix s'est naturellement porté sur ce langage. Le problème est qu'une bonne connaissance du langage Perl est un prérequis pour comprendre cet ouvrage. J'ai donc entrepris de lire en parallèle le livre *Learning Perl* et d'expérimenter la programmation en Perl au moyen de programmes simples n'ayant rien à voir avec l'environnement Apache et mod_perl. Les structures de contrôle, les types, les tableaux, les entrées-sorties, l'accès aux fichiers, les expressions régulières, tout cela devait être maitrisé en un minimum de temps pour pouvoir s'attaquer à l'étude de Apache et mod_perl.

4.3.2. Étude préliminaire de mod_perl

L'idée de mod_perl est de mettre à disposition du développeur Perl presque toutes les fonctionnalités de l'API Apache accessibles à un développeur C. À ce titre, mod_perl, lui-même un module Apache écrit en C, joue le rôle d'intermédiaire entre l'environnement Perl et l'API Apache.

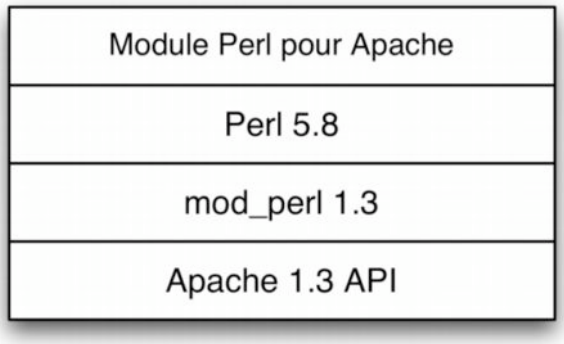


Figure 4.3-1 :environnement d'exécution de modules Perl pour Apache

Un module Apache peut gérer diverses phases du cycle de la requête, par exemple la phase de traduction de l'URI en nom de fichier, ou la phase d'identification, de contrôle d'accès, de déterminaison du type MIME, etc... Tout module écrit en C doit, dans son code source, indiquer à Apache quelles phases de la requête il souhaite gérer, on appelle cela des *hooks* (littéralement crochets). C'est à la compilation de mod_perl que l'on détermine quelles phases de la requête lui seront confiées, et par là durant quelles phases de la requête les modules Perl pour Apache pourront intervenir. Par défaut, mod_perl est compilé avec l'option EVERYTHING=1 qui signifie que tous les *hooks* sont activés (se reporter à la notice d'installation de Perl et Apache en annexe).

Sans trop rentrer dans les détails, il est intéressant de connaître quels sont les différentes phases du cycle des requêtes Apache et ce que l'on est supposé y faire. On parle de cycle, car chaque processus enfant de Apache, une fois lancé, entre dans un boucle perpetuelle dans laquelle il attend une nouvelle requête, la sert, et attend une nouvelle requête.

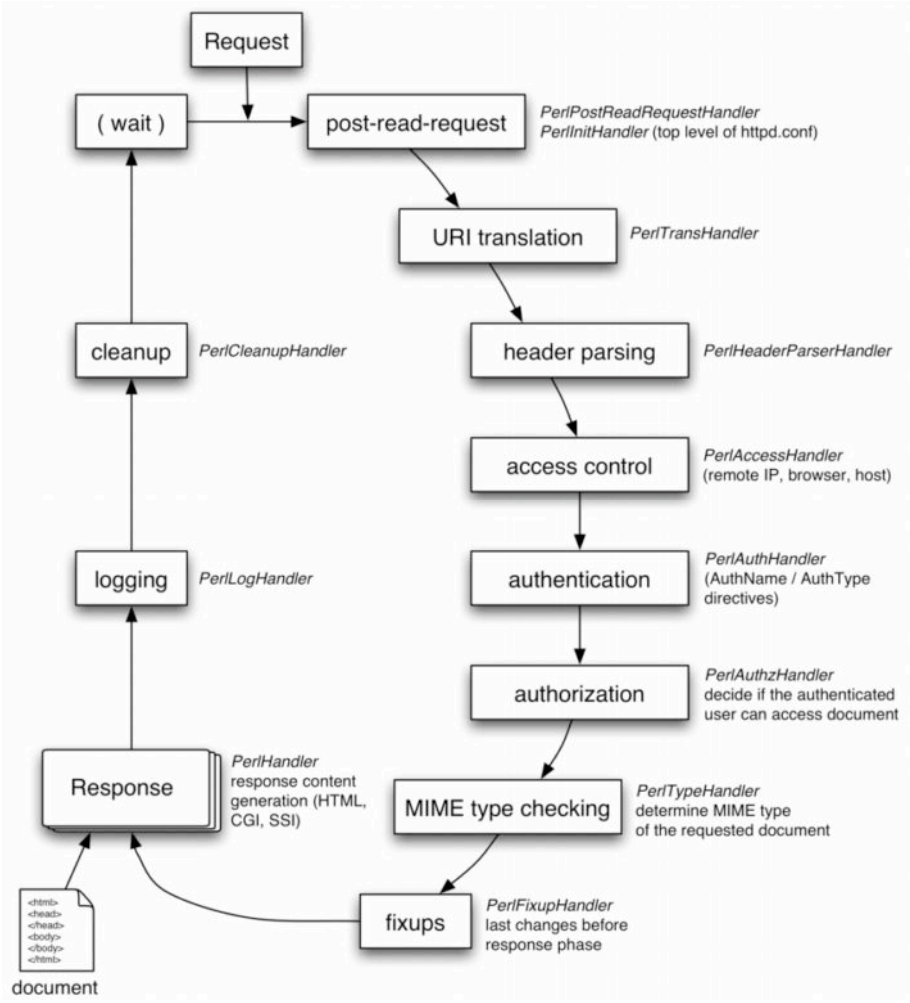


Figure 4.3-2 : cycle d'une requête Apache

À côté de chaque phase est indiqué, en italique, le nom de la directive de mod_perl à placer dans le fichier httpd.conf permettant à un module de "s'abonner" à la phase en question. Tous les directives sont de la forme de *Perl*Handler* dans laquelle l'étoile est remplacée par le nom de la phase.

Child init (*PerlChildInitHandler*)

Sous Unix, Apache est une application multiprocessus : un seul processus parent tournant comme utiliseur *root* lance plusieurs processus enfants tournant comme utilisateur limité *www* chargés de servir les requêtes. Cette phase *Child init* permet au module d'être appelé à chaque fois qu'un processus enfant est lancé, pour des tâches d'initialisation par exemple. Sous Windows, le *handler* ne sera appelé qu'une seule fois lancement du serveur, car la version Windows de Apache comporte un seul processus lourd (mais plusieurs processus légers, les *threads*).

Post-read-request (*PerlPostReadRequestHandler*)

Ce *handler* permet à un module d'être appelé à chaque requête, juste après que Apache aie lu les entêtes et les données de la requêtes, mais avant la phase de traduction de l'URI en nom de fichier. Cette directive ne peut apparaître que dans la partie principale du fichier de configuration ou dans un bloc <VirtualHost>, mais pas dans un bloc <Directory>, <File>, <Location> ou dans un fichier .htaccess, car à ce stade la requête n'a pas encore été associée à un chemin d'accès sur le disque.

URI translation (*PerlTransHandler*)

C'est durant cette étape que l'URI de la requête peut-être réécrite et durant laquelle on lui associe un chemin d'accès physique sur le disque. Le *handler* par défaut de Apache pour cette étape se contente de coller l'URI à la suite du DocumentRoot pour le <VirtualHost> concerné (ou celui du serveur par défaut). Les modules interviennent durant cette phase sont typiquement mod_rewrite, mod_alias et mod_proxy.

Header parsing (*PerlTransHandler*)

Cette étape est une occasion pour les modules de vérifier et modifier les entêtes de la requête. Par rapport à la directive *PerlPostReadRequestHandler*, elle a l'avantage de pourvoir prendre place dans un bloc <Directory>, <File, <Location> ou dans un ficher .htaccess, car le chemin d'accès au fichier sur le disque est maintenant déterminé.

Access control (*PerlAccessHandler*)

Le contrôle d'accès permet aux modules d'accepter ou de refuser la requête en se basant sur des critères simples ne nécessitant pas d'authentification tels que l'heure, la date, l'adresse IP du client, son navigateur Web, etc... Le module mod_access livré avec Apache est un exemple d'implémentation de ce *handler*.

Authentication (*PerlAuthenHandler*)

Cette phase a pour objectif d'authentifier l'utilisateur, c'est à dire déterminer son nom, pour pouvoir ensuite restreindre l'accès une ressource à certains utilisateurs. Il existe une pléthore de modules pour cette phase, des plus simples comme mod_auth ou mod_auth_dbm aux plus complexes faisant appel à des bases de données SQL ou des annuaires LDAP pour l'authentification.

Authorization (*PerlAuthzHandler*)

La phase *Authorization* permet, une fois l'utilisateur authentifié, d'effectuer du contrôle d'accès sur la base de celui-ci. Ce *handler* effectuera typiquement des vérifications d'appartenance de l'utilisateur à un groupe d'utilisateurs au moyen d'une base de données, et retournera son verdict à Apache : autorisé ou non-autorisé. Souvent, les mêmes modules gèrent l'étape d'authentification et celle de contrôle d'accès, maladroitement appelée *Authorization*.

MIME type checking (*PerlTypeHandler*)

Lorsque Apache renvoie un document au client, il doit indiquer dans les entêtes de la réponse le type MIME du document, pour permettre au navigateur Web d'interpréter correctement son contenu. Le document peut être du texte simple, un document HTML, une image, un fichier sonore, une vidéo, etc... Les modules qui prennent en charge cette étape doivent déterminer le type MIME du document. Ils peuvent le faire de diverses façons : au moyen du suffixe du nom de fichier, en comparant les premières lignes du fichier avec une collection de *fingerprints*, etc...

Fixups (*PerlFixupHandler*)

Les modules qui ont besoin d'effectuer des tâches juste avant que ne commence la phase de réponse peuvent s'abonner à la phase *Fixup*.

Response (*PerlHandler*)

C'est durant la phase de réponse que le document à retourner au client est généré. Tous les systèmes de pages dynamiques tels que mod_cgi, mod_include,. mod_php4 ou Apache::Registry interviennent durant cette phase très importante. Pour les modules jouant le rôle de proxy, la phase de réponse est le moment choisi pour effectuer une requête interne vers le serveur cible et renvoyer la réponse au client.

Logging (*PerlLogHandler*)

Après la phase de réponse, Apache entre dans le phase de *logging* durant laquelle les historiques sont mis à jour. Ce *handler* est appelé même si la requête a échoué. À ce stade, toutes les informations récoltées sur la requête au cours des précédentes phases sont disponibles, comme le type MIME, le nombre d'octets transférés, la durée de traitement de la requête, le code de statut retourné, etc... L'historique peut être mémorisé dans un fichier comme le fait par défaut Apache (*ErrorLog, AccessLog*) ou l'on peut imaginer stocker ces informations dans une base de données relationnelle, par exemple.

Cleanup (*PerlCleanupHandler*)

À la fin de chaque transaction, Apache libère les ressources occupées durant la requête. La phase *Cleanup* est une occasion pour les modules Perl de libérer des espaces mémoire partagés, refermer la connexion à une base de données, effacer des fichiers temporaires, etc... Cette phase est toujours appelée, même si un *handler* a prématurément terminé la requête en retournant un code d'erreur.

Lorsqu'un module Perl pour Apache souhaite gérer une phase de la requête, il doit définir une procédure *handler* (c'est son nom par défaut) qui sera appelée par Apache au moment opportun. Dans tous les cas, l'objet *server_rec* permettant de faire le lien avec l'API Apache sera passé comme unique paramètre au *handler*, mais son contenu ne sera pas le même selon les phases de la requête. Un module peut parfaitement prendre en charges plusieurs phases de la requête (et donc définir plusieurs procédures *handler*), mais cela n'est pas très fréquent.

Un module pour Apache communique avec le serveur, d'une part au moyen de l'objet *server_rec* accessible en lecture/écriture, d'autre part en retournant un code de statut à la fin de l'exécution du *handler*. Ces codes de statut sont définis dans le paquetage Apache::Constants, certains correspondent à des codes de statut HTTP, d'autres sont propres à mod_perl. Voici les principaux :

Code HTTP	Constante	Description
200	DOCUMENT_FOLLOWS	L'URI a été trouvée, le document suit
400	BAD_REQUEST	La requête contient une erreur de syntaxe
401	AUTH_REQUIRED	Le client n'a pas fourni les informations correctes permettant de l'authentifier
403	FORBIDDEN	Le client n'a pas l'autorisation d'accéder à ce document
404	NOT_FOUND	Le document demandé n'existe pas
405	METHOD_NOT_ALLOWED	La méthode de la requête (par ex. PUT) n'est pas autorisée ici
500	SERVER_ERROR	Une erreur est survenue au niveau du serveur

Figure 4.3-3 : codes de statut HTTP et constantes Perl correspondantes

Hormis les codes de statut définis dans la norme HTTP, il existe quelques constantes spéciales propres à Apache qu'un *handler* peut retourner pour donner une indication au serveur :

Constante	Description
OK	Cette constante signifie que le *handler* s'est exécuté avec succès. Pour la plupart des phases, Apache va maintenant soumettre la requête à un autre *handler* enregistré pour la phase courante, mais pour certaines phases comme *URI translation* ou *Response*, Apache ne tolère qu'un seul *handler*, par conséquent la phase est terminé dès qu'un *handler* retourne le status OK.
DECLINED	Ce code est retourné par un *handler* qui a décidé de ne pas prendre en charge la phase de la requête. Apache va faire comme si le *handler* n'avait jamais été appelé et soumettre la requête à un autre module qui s'est inscrit pour gérer la phase en cours, ou gérer la phase de lui-même avec son *handler* par défaut le cas échéant.
DONE	Ce code est un moyen d'arrêter la transaction sans générer un code d'erreur : lorsque DONE est retourné, Apache ferme la connexion avec le client et saute directement à l'étape de *logging*.
SERVER_ERROR, etc...	En plus des 3 codes de status propres à mod_perl, un *handler* peut retourner un code de status HTTP dont la liste non-exhaustive est donnée dans le tableau qui précède. Apache va alors créer l'entête HTTP appropriée et l'envoyer au navigateur. C'est de cette manière que les *handler* indiquent qu'une erreur est survenue, qu'un document n'a pu être trouvé, etc...

Figure 4.3-4 : autres codes de statut propres à Apache

Ces constantes seront le moyen employé par notre module *ProxyFilter* pour indiquer au serveur Apache si la requête est autorisée ou refusée. Plus exactement, *ProxyFilter* retourne essentiellement 3 codes de statut : FORBIDDEN lorsque la requête est refusée parce que la requête ne satisfait aucune règle Allow ou qu'elle satisfait un règle Deny, SERVER_ERROR lorsqu'une erreur de configuration du module est survenue (par exemple en cas d'erreur de syntaxe dans le fichier XML ou de fichier de configuration introuvable ou illisible) et OK lorsque la requête est autorisée. Le code de statut DOCUMENT_FOLLOWS (synonyme de HTTP_OK) ne doit pas être confondu avec la constante OK : le premier indique au navigateur Web que le document a été trouvé et peut être retourné, le second indique à Apache que le *handler* s'est exécuté avec succès et qu'il peut continuer à traiter la requête. Ainsi, lorsque le serveur derrière le *reverse proxy* renvoie une erreur HTTP 404 (*Not Found*), celle-ci est propagée au client par le proxy mais n'est pas considérée comme une erreur d'exécution du *handler* (il renvoie un code OK à Apache).

4.3.3. Déterminer quelle(s) phase(s) de la requête traiter

Comme nous venons de le voir, un module Apache a la possibilité d'intervenir durant de nombreuses phases de la requête, certaines étant plus judicieuses que d'autres en fonction de la fonction remplie par le module.

ProxyFilter est un module de nature hybride : il assure à la fois la fonction de réécriture d'URL, celle de proxy (plus exactement *reverse proxy*), celle d'historique et, dans une certaine mesure, de contrôle d'accès puisque la requête est acceptée ou refusée en se basant sur des critères tels que l'URL, les entêtes et les paramètres de scripts. Pour ces raisons, il n'était pas évident de choisir les phases de la requête durant lesquelles *ProxyFilter* doit intervenir.

Le cahier des charges, assez vague sur ce point, réclame que le module à développer soit capable de travailler avec mod_proxy et mod_rewrite, étudiés dans le cadre du travail de semestre, et qui à eux deux offrent des capacités de filtrage d'URL limitées.

L'idée derrière cette exigence était de ne pas avoir à réinventer ce qui existe déjà : *ProxyFilter* devrait ainsi déléguer la réécriture et le filtrage d'URI à mod_rewrite, et la partie (reverse) proxy à mod_proxy, d'autant que ces modules sont écrits en C et ont été largement éprouvés.

Cependant, la réalité du terrain n'est pas si simple : pour la phase dite de *URI translation*, Apache ne tolère qu'un seul *handler*, et donc si l'on veut laisser la main à mod_rewrite, notre module ne peut pas intervenir durant cette phase (ou alors il doit retourner DECLINED pour que Apache soumette la requête au *handler* suivant pour la-dite phase). De même pour la phase *Response* qui est occupée par mod_proxy : Apache ne tolère par le chaînage des *handlers* pour cette phase.

Un autre problème est que rien n'est prévu pour permettre aux modules Apache de collaborer entre-eux : un module ne reconnaît que ses directives de configuration statiques situées dans le fichier *httpd.conf*. Dès lors, comment, lorsqu'une règle de filtrage d'URL est définie dans le fichier de configuration de *ProxyFilter*, communiquer une nouvelle règle à mod_rewrite sans modifier dynamiquement le fichier *httpd.conf* et redémarrer le serveur ?

Mod_proxy intervient une première fois durant la phase *URI translation* pour vérifier sur la base du préfixe de l'URI si la requête doit être acheminée par le (*reverse*) *proxy*, compte tenu des directives *ProxyPass* et *ProxyPassReverse*. Le cas échéant, il configure dynamiquement son *handler* pour gérer la phase *Response* de la requête, en lieu et place du *handler* par défaut de Apache. Si mod_rewrite est actif, celui-ci peut également dynamiquement forcer une requête à être transmise à mod_proxy (flag *[P]* d'une règle *RewriteRule*, voir travail de semestre à ce sujet), mais dans ce cas seule la phase *Response* est gérée par mod_proxy, et non la phase *URI translation*.

Au lieu de simplifier le problème, il apparaît que le fait que *ProxyFilter* doive cohabiter avec mod_proxy et mod_rewrite le complexifie. Pour ces raisons, il a été décidé que *ProxyFilter* devrait être **compatible avec mod_proxy et mod_rewrite**, mais qu'il ne **ferait pas appel à leurs services**.

Cela est possible si *ProxyFilter* se contente de gérer la phase *Response* de la requête. Comme mod_rewrite n'intervient que durant les phases préliminaires de la requête, cela rend possible l'usage de mod_rewrite pour réécrire l'URI en amont de *ProxyFilter*. Même chose pour mod_proxy : s'il détecte durant la phase *URI translation* que la requête est destinée à passer par son proxy, il configure dynamiquement son *Response handler* et prend ainsi la main au *Response handler* de *ProxyFilter* (configuré statiquement par une directive *PerlHandler* dans le fichier *http.conf*). Ainsi, il devient possible d'utiliser en même temps ces trois modules, bien qu'ils restent indépendants les uns des autres.

Pour *ProxyFilter*, ne gérer qu'une seule phase de la requête a certains avantages : un code simplifié puisque ne comportant qu'un seul *handler*, ainsi que la possibilité d'utiliser d'autres modules pour l'authentification (mod_auth, mod_auth_dbm) et le contrôle d'accès (mod_access) qui font leur travail en amont de *ProxyFilter*, et ainsi de limiter l'accès au *reverse proxy* à certains utilisateurs ou à certaines machines (cela pourrait également être fait au niveau du serveur Web, il est vrai). Par contre, le fait que chaque requête doive passer plusieurs phases de traitement avant de pouvoir être bloquée par le firewall applicatif signifie plus de travail au niveau du proxy que si une URL contenant des caractères invalides avait pu être détectée directement dans la phase *URI translation*, et donc que le *reverse proxy* est plus vulnérable aux attaques par déni de service

(DOS). Nous avons décidé de vivre avec ces limitations, sachant que le module développé dans le cadre de ce travail de diplôme n'est qu'expérimental, et qu'il pourra toujours être amélioré par la suite s'il est publié comme un projet *open source* (Rome n'a pas été bâtie en un jour).

4.3.4. Mon premier CGI écrit en Perl

Bien que l'usage de l'objet *server_rec* soit requis pour tirer parti de toutes les fonctionnalités de mod_perl, les scripts CGI conventionnels peuvent être exécutés par mod_perl au travers du module Apache::Registry, avec à la clé un important gain de performance, puisque les scripts sont précompilés et préchargés en mémoire ou moment du démarrage du serveur. Rappelons qu'un CGI est un script qui lit la requête HTTP sur l'entrée standard, obtient des informations sur le serveur ou le client au travers de variables d'environnement (l'adresse IP du client, la date, le nom du serveur, etc...) et retourne un document sur la sortie standard. Fort heureusement, les scripts qui tournent au sein de Apache::Registry voient leur entrée et sortie standards automatiquement redirigés vers des appels aux méthodes *read()* et *print()* de l'objet *server_rec*, et donc la plupart des CGIs peuvent être exécutés sans modification au travers de cet environnement d'émulation.

Voici un exemple simple de script CGI :

```
#!/usr/bin/perl
# File : hello

print "Content-Type: text/html\n\n";

print <<END;
<html>
<head>
   <title>Hello CGI</title>
</head>
<body>
<h1>Hello $ENV{REMOTE_ADDR}</h1>
This is a traditional CGI script in the ScriptAlias folder.
Apache loads the Perl interpreter everytime I am called, so it's not so fast
<hr><address>&copy; Sylvain Tissot 2003</address>
</body>
</html>
END
```

Figure 4.3-5 : script CGI conventionnel n'utilisant pas l'API Apache

Un tel script peut être directement testé dans la console à l'aide de la commande `perl` : il retourne un document HTML sur la sortie standard. Pour l'exécuter comme un CGI dans Apache, on utilise traditionnellement la directive *ScriptAlias* qui permet de définir un répertoire local dans lequel tous les fichiers seront interprétés par mod_cgi, qui a son tour appelera l'interpréteur Perl du système dont l'emplacement est indiqué sur la première ligne du script.

```
# Chargement des modules
LoadModule cgi_module     libexec/httpd/mod_cgi.so
LoadModule alias_module    libexec/httpd/mod_alias.so
AddModule mod_cgi.c
AddModule mod_alias.c
# Répertoire contenant les scripts CGI
ScriptAlias    /cgi-bin/    /usr/perl/cgi-bin/
```

Figure 4.3-6 : configuration pour l'exécution de scripts CGI au travers de mod_cgi

Après un redémarrage de Apache, le script, situé dans `/usr/perl/cgi-bin/hello`, est accessible localement au travers d'un navigateur Web à l'adresse `http://127.0.0.1/cgi-bin/hello` :

Figure 4.3-7 : appel d'un simple script Perl au travers de mod_cgi

Cette technique n'utilise pas mod_perl. L'interpréteur Perl du système hôte est relancé et le script recompilé à chaque nouvelle requête, ce qui est peu performant.

4.3.5. Mon premier CGI tournant dans mod_perl et Apache::Registry

Voyons maintenant comment exécuter le même script au travers de mod_perl et Apache::Registry :

```
# Chargement de mod_perl
LoadModule perl_module   libexec/httpd/libperl.so
AddModule mod_perl.c

Alias /perl/     /usr/perl/cgi-bin/
PerlModule Apache::Registry
<Location /perl>
    SetHandler perl-script
    PerlHandler Apache::Registry
    PerlSendHeader on
    Options +ExecCGI
</Location>
```

Figure 4.3-8 : configuration pour l'exécution d'un script CGI avec mod_perl et Apache::Registry

Dans cet exemple, la directive *PerlModule* a pour effet de précharger et précompiler Apache::Registry au démarrage du serveur : il sera plus rapide d'accès par la suite. Dans le bloc <Location>, on utilise la directive *SetHandler* pour que Apache délègue à mod_perl les requêtes concernant `/perl`. La directive *PerlHandler* indique à mod_perl que la phase de génération du document retourné (*Content handler*) doit être confiée à Apache::Registry. Enfin, *PerlSendHeader* est nécessaire puisque notre CGI ne génère pas de lui-même les entêtes HTTP de la réponse, et *Options +ExecCGI* autorise l'exécution de scripts dans `/perl`, pour autant que les permissions Unix du fichier comportent le flag x (eXecute).

Le même script que précédemment est maintenant accessible sans modification de son code au travers de mod_perl et Apache::Registry à l'adresse `http://127.0.0.1/perl/hello` :

Figure 4.3-9 : appel d'un script CGI au travers de mod_perl et Apache::Registry

Dans ce dernier exemple, nous n'avons pas écrit à proprement parler un module Perl pour Apache : c'est Apache::Registry qui jouait le rôle de module, émulant ainsi un environnement d'exécution de scripts CGI.

4.3.6. Mon premier module Apache en Perl

Nous allons maintenant écrire notre premier vrai module Perl, qui utilise exclusivement l'API de Apache, accessible au travers de l'objet *server_rec* que l'on a l'habitude de mémoriser dès le début du *handler* dans une variable `$r` :

```
package Apache::Hello;
# File : Apache/Hello.pm

use strict;
use Apache::Constants qw(:common);

# Content handler
sub handler {
     my $r = shift;
      $r->content_type('text/html');
      $r->send_http_header;
      my $host = $r->get_remote_host;
      $r->print(<<END);
<html>
<head>
      <title>Hello World</title>
</head>
<body>
<h1>Hello $host</h1>
This is my first Perl module, thanks to Apache and mod_perl !<p>
<hr><address>&copy; Sylvain Tissot 2003</address>
</body>
</html>
END
     return OK;
}

1;
```

Figure 4.3-10 : un premier module Perl pour Apache

Le code se compose essentiellement d'une routine *handler()* qui sera appelée par Apache au moment de la phase de génération du document retourné de chaque requête. On y extrait l'objet *server_rec*, règle le type MIME du document retourné, transmet les entêtes de la réponse au client, puis utilise la méthode *print()* pour retourner un document HTML. Le nom d'hôte du client est copié dans la variable `$host` pour être retourné dans la page de réponse.

Pour installer ce module Apache::Hello sur le serveur, il faut le placer de telle sorte qu'il soit accessible à Perl dans le fichier Hello.pm du paquetage Apache. Au même titre que Java et la variable `CLASSPATH`, Perl recherche ses paquetages dans une liste de répertoires mémorisée dans la variable `@INC`, dont il est intéressant d'afficher le contenu :

```
[powermac:~] stissot% perl -e 'print join ("\n", @INC);'
/sw/lib/perl5/darwin
/sw/lib/perl5
/System/Library/Perl/darwin
/System/Library/Perl
/Library/Perl/darwin
/Library/Perl
/Library/Perl
/Network/Library/Perl/darwin
/Network/Library/Perl
/Network/Library/Perl
```

Figure 4.3-11 : affichage de la liste des répertoires de recherche des paquetages Perl

Plutôt que de les disperser dans le système, il peut paraître logique de regrouper les modules Perl pour Apache dans le dossier où réside le serveur (*ServerRoot*), dans notre cas l'emplacement standard `/usr/local/apache/`. Il s'agit de créer un répertoire `/usr/local/apache/perl` et de l'ajouter au contenu de `@INC` pour que Perl puisse trouver les modules qui y sont placés. Cela peut se faire de plusieurs manières : il est envisageable de le faire au démarrage du système, mais cela n'est pas très logique sachant que ce répertoire n'est utile que dans l'environnement mod_perl. Une meilleure solution consiste à faire exécuter un script Perl au démarrage du serveur Apache, qui règle la variable `@INC` en conséquence et importe les paquetages dont nous avons fréquemment besoin au sein de modules Apache afin de pouvoir y accéder facilement et rapidement par la suite.

Créons donc un fichier `/usr/local/apache/perl/startup.pl` avec le contenu suivant :

```
#!/usr/bin/perl

### initialisation script for mod_perl

# modify the include path before we do anything else
BEGIN {
        use Apache ();
        use lib Apache->server_root_relative('perl');
}

# commonly used modules
use Apache::Registry ();
use Apache::Constants ();
use CGI qw(-compile :all);
use CGI::Carp ();

1;
```

Figure 4.3-12 : exemple de fichier d'initialisation de l'environnement mod_perl

Use lib permet d'ajouter un répertoire de recherche à `@INC`, la fonction *server_root_relative* est un moyen commode de convertir un chemin d'accès relatif à la racine du serveur en chemin absolu. Les paquetages importés par *use* ne sont qu'un exemple : on pourra y placer ce que l'on semble bon, et c'est autant de paquetages qu'il ne sera plus nécessaire d'importer au début de chaque script.

Pour indiquer à mod_perl d'exécuter ce fichier de configuration à chaque démarrage du serveur, on placera les instructions suivantes quelque part dans le *httpd.conf* (en-dessous du chargement de mod_perl, dans tous les cas) :

```
PerlRequire                perl/startup.pl
PerlFreshRestart           On
```

Figure 4.3-13 : exécution d'un script Perl au démarrage du serveur Apache

Notre module situé dans `/usr/local/apache/perl/Apache/Hello.pm` doit maintenant être visible à mod_perl. Il est temps de configurer mod_perl de sorte à confier la phase de génération du document retourné (*Response*) à Apache::Hello :

```
<Location  /Hello>
        SetHandler perl-script
        PerlHandler Apache::Hello
</Location>
```

Figure 4.3-14 : mapping sur une URI d'un module Perl pour Apache

Comme précédemment, la première directive indique à Apache de confier à mod_perl la gestion des requêtes pour `/Hello`. Mod_perl est compilé pour supporter un certain nombre de *hooks* (littéralement crochets) qui sont autant de phases de la requête/réponse durant lesquelles un module Perl pour Apache peut intervenir. Ici, nous configurons mod_perl au moyen de *PerlHandler* pour que la phase de génération du document à retourner soit confiée à la routine *handler* du module Apache::Hello. Remarquons que, dans l'exemple précédent, c'était Apache::Registry lui-même qui se chargeait de cette phase en appelant notre CGI en coulisses.

Après un redémarrage du serveur, le module est accessible à `http://127.0.0.1/hello` :

Figure 4.3-15 : appel d'un module gérant la phase Response de la requête

Ces programmes simples permettent de s'assurer du bon fonctionnement de mod_perl, ce qui est indispensable avant de se lancer dans le développement d'un module plus complexe.

4.3.7. Lecture d'un fichier XML avec Perl

Pour les raisons expliquées plus haut, une syntaxe XML a été retenue pour les fichiers de configuration de *ProxyFilter*. Plutôt que de développer soi-même la logique permettant d'interpréter un fichier XML, il était préférable d'utiliser un *parser* existant, ce serait plus fiable et plus performant.

Le traitement de flux de données étant un domaine pour lequel Perl est particulièrement indiqué, une visite du site du CPAN (*Comprehensive Perl Archive Network*) permet de se rendre compte des dizaines de modules existants pour la lecture ou l'écriture de données XML.

Comme expliqué dans le chapitre précédent, il existe deux catégories de *parsers* XML : ceux travaillant en mode flux, et ceux travaillant en mode arbre. Les premiers sont plus rapides que les seconds, mais leur usage est plus complexe. Comme la syntaxe XML de *ProxyFilter* n'allait pas être compliquée, un *parser* en mode flux était adapté.

Après m'être renseigné et avoir testé plusieurs modules, j'ai retenu le module XML::Parser 2.34 développé par Matt Sergeant, qui repose sur le célèbre Expat, écrit en langage C. L'avantage de faire appel à un programme C externe plutôt que d'utiliser un *parser* entièrement écrit en Perl est un gain de vitesse indéniable, l'inconvénient est qu'il faut impérativement compiler et installer Expat avant d'utiliser XML::Parser.

Ayant déjà utilisé le *parser* SAX (*Simple API for XML Parsing*), lui aussi en mode flux, dans divers programmes Java, l'apprentissage de XML::Parser n'était pas très difficile. Le principe est de configurer le *parser* en lui transmettant des références sur les fonctions (*handlers*) à appeler pour chaque type d'élément rencontré par le *parser* : début de document, balise ouvrante, données CDATA, attributs, balise fermante, fin de document, etc... Lorsque le *parser* appelle un *handler*, il transmet en paramètre d'utiles informations sur l'élément rencontré : le reste n'est que logique dans l'écriture des *handlers*, permettant de mémoriser la position courante dans le document et de construire une structure de données en mémoire à partir des informations contenues dans le fichier XML.

Comme la configuration du module se ferait, dans un premier temps, en éditant les fichiers de configuration XML, il était important de prévenir les erreurs de syntaxe de l'utilisateur. Pour cela, XML offre un outil très pratique appelé une DTD (*Document Type Definition*) et qui permet de définir la "grammaire" d'un document, c'est à dire quels éléments sont autorisés et à quels endroits.

Lorsqu'un document satisfait les règles de base de la syntaxe XML (à chaque balise ouvrante correspond une balise fermante, un document ne comporte qu'un seul élément racine, etc...), on parle d'un document *bien formé*. Si le document respecte la syntaxe définie dans une DTD qui lui est attachée, on parle de document *valide*.

Tous les *parsers*, même les plus élémentaires, sont capable de déterminer si un document bien formé et retournent un message d'erreur si ce n'est pas le cas. En revanche, seuls certains *parsers* sont capable de valider un document relativement à une DTD, et ce n'est malheureusement pas le cas de Expat, ce qui remet en question l'utilité d'avoir écrit une DTD pour décrire la syntaxe de configuration de *ProxyFilter*.

Il existe bien sûr de nombreux validateurs XML sous forme de services Web, mais j'ai préféré joindre à la distribution de *ProxyFilter* un petit programme `Echo10.java` qui utilise le *parser* SAX inclu dans JDK 1.4 pour effectuer une validation du document. Il est recommandé à l'utilisateur de *ProxyFilter* de soumettre les fichiers à ce validateur après chaque modification de la configuration du module, afin de prévenir d'éventuels messages "Server Error" de Apache en cas d'erreur syntaxique dans la configuration du module. À noter que la DTD ne permet pas d'éviter toutes les incohérences possibles dans les fichiers de configuration : les éventuelles erreurs qui ne seraient pas détectées à la validation sont, dans la mesure du possible, vérifiées par *ProxyFilter* lui-même à la lecture de ses fichiers de configuration et signalées dans le *log* le cas échéant.

4.3.8. Définition de la syntaxe de configuration

À partir des fonctionnalités arrêtées du firewall applicatif et de l'expérience acquise avec AppShield, il devenait possible de réfléchir à une syntaxe de configuration XML suffisament souple qui permette à l'utilisateur de définir les règles de filtrage et les paramètres généraux du module.

Pour rappel, voici les éléments que nous aimerions pouvoir vérifier / filtrer :

- Syntaxe globale de l'URL de la requête
- Entêtes HTTP de la requête
- Contenu de la requête
- Entêtes HTTP de la réponse
- Contenu de la réponse

De plus, il est demandé que le module puisse travailler en mode *White list* dans lequel tout est bloqué par défaut sauf ce qui est reconnu comme étant valide (à la manière d'un firewall conventionnel) ou dans un mode *Black list* dans lequel on laisse tout passer par défaut, sauf ce qui est reconnu comme étant suspect (à la manière d'un IDS). Mieux : nous aimerions pouvoir changer ce comportement par défaut en fonction du contexte de la requête, c'est à dire du répertoire concerné de l'application Web.

Beaucoup de solutions de firewalls HTTP emploient un système de règles "à plat", dans lequel toutes les règles de type Deny ou Allow concernent toute l'URL, et AppShield ne fait pas exception. Cela est souhaitable dans bien des cas, mais il en résulte une syntaxe de configuration plutôt lourde s'il faut décrire au firewall toute la structure d'une application Web, d'où les modes d'auto-apprentissage des règles, dont il est prévu de doter *ProxyFilter* à terme.

Avec *ProxyFilter*, nous aimerions offrir le choix à l'utilisateur : des règles globales qui concernent toute l'URL, ainsi que des règles locales qui ne s'appliquent qu'à un répertoire, à un fichier ou à un type de fichier. Pour cela, nous avons pensé à implémenter plusieurs types de règles, chacune correspondant à un élément XML.

Les règles <url>

Ces règles sont globales à l'application et concernent toute l'URI. Elles sont évaluées en priorité par rapport aux autres règles décrites ci-dessous. Ce type de règle permet de rechercher la présence de certains caractères, au moyen d'expressions régulières, dans l'URI complète, c'est à dire la partie de l'URL suivant le nom d'hôte, y compris l'éventuelle chaîne de paramètres transmise à la fin de l'URL. C'est l'emplacement de choix pour bloquer les attaques de type XSS ou limiter la longueur totale des URLs permettant de prévenir les attaques de type *buffer overflows* sur des serveurs ou applications mal protégés.

Les règles <file>

Ces règles, placées dans un élément <directory> ou <webapp> permettent de définir la présence de fichiers localement dans l'application. Au sens de *ProxyFilter*, une requête ne comportant aucun paramètre GET ou POST concerne un fichier. Le nom du fichier peut être exact, ou générique par l'utilisation d'expressions régulières et/ou de *charsets*.

Les règles <script>

Les règles <script> sont conceptuellement identiques aux règles <file>, sauf qu'elles concernent une requête comportant au moins un paramètre GET ou POST. L'élément <script> est un conteneur pour des règles <param>.

Les règles <param>

Les règles <param> prennent place à l'intérieur d'une règle <script>, elles servent à décrire chaque paramètre autorisé du script, son nom, sa valeur en expression régulière, sa méthode (GET ou POST). Les paramètres peuvent être mandatoires (leur présence est exigée pour que la règle <script> soit satisfaire) ou optionnels (leur présence n'est pas requise pour que la règle <script> soit satisfaite).

Les règles <header>

Elle servent à définir quels sont les entêtes HTTP autorisés en entrée (dans la requête provenant du client) et en sortie (dans la réponse provenant du serveur). Chaque entête est défini par son nom et par la plage de valeurs qu'il peut prendre à l'aide d'expression règulière, ou par une valeur exacte. *ProxyFilter* ne permet pas définir des règles de filtrage d'entêtes localement en fonction du contexe (URI) de l'application : elles sont définies globalement pour toute l'application.

Chaque règle doit être contenue dans un élément décrivant le comportement qui lui est associé (*policy*). Ainsi, on utilisera des éléments <allow> pour contenir les règles qui définissent ce qui est autorisé et des éléments <deny> pour contenir les règles qui définissent ce qui est interdit. Un troisième comportement <filter> ne concerne que les règles <header> : il sert à filtrer certains entêtes HTTP sans pour autant bloquer la requête s'ils sont présents.

Les éléments <allow> et <deny>, quand à eux, doivent être contenus dans des éléments de type <directory> ou <webapp> qui décrivent à quel répertoire de l'application ils s'appliquent.

L'élément <webapp>

Cet élément représente la racine de l'application Web, il est aussi l'élément parent du fichier proxyfilter_webapp décrivant la structure de l'application. Il peut contenir des éléments <allow> ou <deny> pour définir des règles s'appliquant à la racine, ou des éléments <directory> pour déclarer des sous-répertoires. Un rôle de l'élément <webapp> est de définir de *default policy* (comportement par défaut) global de l'application, qui sera hérité dans les sous-répertoires sauf s'il est redéfini dans un élément <directory>.

L'élément <directory>

Il permet de déclarer un sous-répertoire de l'application. Les règles <allow> ou <deny> contenues dans un élément <directory> ne s'appliquent qu'à ce répertoire, et éventuellement à ceux qu'il contient par héritage. Un élément <directory> peut contenir d'autres éléments <directory>, permettant ainsi de définir une arborescence de répertoires illimitée.

Pour rendre les choses un peu plus claires, voici un exemple de fichier de configuration :

```
 1  <webapp default="deny" allowindex="true">
 2
 3     <allow>
 4         <file name ="index.html">
 5         <file name ="contact.html">
 6         <file name ="menu.html">
 7     </allow>
 8
 9     <deny>
10         <url value="~.*" minlength="200">
11         <url value="~root\.exe">
12         <url value="~<script>">
13         <url value="~\.\./">
14         <url value="~passwd">
15         <url value="~htaccess">
16         <file name="~^\." inherit="true">
17     </deny>
18
19     <directory name="images">
20        <allow>
21            <file name="~\.jpg$">
22            <file name="~\.jpeg$">
23            <file name="~\.gif$">
24        </allow>
25     </directory>
26
27     <directory name="cgi-bin" allowindex=false>
28        <allow>
29            <script name="mail.cgi">
30                <param name="to" optional="false" method="post">
31                <param name="from" optional="false" method="post">
32                <param name="subject" optional="true" method="post">
33                <param name="text" optional="false" maxlength="300">
34            </script>
35        </allow>
36     </directory>
37  </webapp>
```

Figure 4.4-13 : exemple de fichier de configuration webapp

À la ligne 1, l'élément racine <webapp> indique que le comportement par défaut (*default policy*) de l'application est de tout bloquer en l'absence de règle <allow>, on a donc clairement un monde de fonctionnement *White list*. Il indique également que l'index de répertoire est autorisé à la racine et, par héritage, dans les sous-répertoires. Cela signifie qu'une requête de type GET / ou GET /images/ sera autorisée par le proxy.

Aux lignes 3 à 7, un bloc <allow> déclare la présence de trois fichiers à la racine pour lesquels les requêtes seront autorisées. Ainsi, une requête GET /menu.html sera autorisée par exemple, mais une requête GET /menu.html?toto=3 sera refusée, car les règles <file> correspondent à des fichiers statiques du serveur et ne tolèrent pas la présence de paramètres GET ou POST:

L'élément <deny> situé à la racine de l'application (lignes 9 à 17) est contient des règles <url> globales à l'application. On les utilise pour rechercher la présence de caractères illicites dans toute l'URI, comme des attaques XSS ou des tentatives d'accès à des fichiers invisibles, du système ou *htaccess* par exemple. La règle <url> de la ligne 10 est intéressante : elle n'est vraie que si l'URI a

une longueur de 200 caractères au moins, et ce quelle que soit sa valeur (l'expression régulière "~.*" indique une suite de caractères quelconques). C'est une manière de limiter, pour toute l'application, la longueur maximale des URIs et donc de se protéger d'une attaque *buffer overflow* dans l'URL. La règle <file> de la ligne 16 interdit, pour toute l'application grâce à son attribut *inherit*, les requêtes vers des fichiers dont le nom commence par un point, qui correspondent aux fichiers invisibles sous Unix.

Un premier bloc <directory> (lignes 19 à 25) décrit un répertoire /images/ dans lequel les requêtes à des fichiers JPEG ou GIF sont autorisées. En l'absence d'attribut *default* spécifiant le contraire, un élément <directory> hérite du *default policy* de son parent (ici <webapp>). De même, il hérite de l'attribut *allowindex* de son parent s'il n'est pas redéfini, et donc une requête GET /images/ sera autorisée.

Le second bloc <directory> (lignes 27 à 36) décrit un répertoire `/cgi-bin` contenant un script `mail.cgi` vraisemblablement utilisé pour transmettre le contenu d'un formulaire à une adresse e-mail. Ce script accepte 4 paramètres, tous transmis comme données POST (et non dans l'URL), et dont 3 sont mandatoires (la requête est refusée s'ils ne sont pas présents). Aucune limitation n'est fixée quant à la valeur de ces paramètres, car l'attribut *value* est absent des règles <param>, mais le champ *text* a une longueur limitée à 300 caractères. Cette configuration n'est qu'un exemple et n'est pas vraiment sécuritaire, puisque des attaques XSS ou BOF sont possibles en transmettant des données POST au script mail.cgi si celui-ci est vulnérable (les règles <url> globaless permettent de filtrer le contenu des paramètres transmis à la fin de l'URL, mais pas les données POST).

Pour mieux comprendre les conséquences de cette configuration, voici quelques exemples d'URL autorisées et interdites par le proxy, avec en regard le numéro de la ligne de la règle respective dans le fichier de configuration :

requête	autorisé	règle
`GET / HTTP/1.1`	oui	1
`GET /images/ HTTP/1.1`	oui	1, 19
`GET /cgi-bin/ HTTP/1.1`	non	27
`GET /index.html HTTP/1.1`	oui	4
`GET /index.html?page=3 HTTP/1.1`	non	4, default
`GET /images/img_2030.jpg HTTP/1.1`	oui	21
`GET /images/htaccess.gif HTTP/1.1`	non	15
`GET /images/.invisible.jpeg HTTP/1.1`	non	16

`POST /cgi-bin/mail.cgi HTTP/1.1`
`data:to=toto@exemple.com&from=foo@exemple.com`
`&subject=hello&text=have+a+nice+day`
	oui	27, 30, 31, 32, 33

`GET /cgi-bin/mail.cgi?to=toto@exemple.com`
`&from=foo@exemple.com&subject=hello`
`&text=have+a+nice+day HTTP/1.1`
	non	30

Figure 4.3-17 : URL autorisées et refusées par ProxyFilter

La description arborescente de l'application Web ayant pour élément racine <webapp> sera stockée dans un fichier *proxyfilter_webapp.xml* (nom par défaut, modifiable), dont l'emplacement est indiqué au module au moyen d'une unique directive *ProxyFilterConfig* placée dans le fichier *httpd.conf*. Les autres informations de configuration sont contenues dans des fichiers séparés, ceci

par souci de clareté et en raison d'une limitation du format XML qui n'autorise qu'un seul élément racine par fichier.

Les directives de configuration globales qui s'appliquent à toute l'application sont contenues dans le *proxyfilter_config.xml*, lui aussi validé par une DTD, dont voici un exemple :

```
 1  <config>
 2      <charsetsfile>/etc/proxyfilter_charsets</charsetsfile>
 3      <mappingsfile>/etc/proxyfilter_mappings</mappingsfile>
 4      <webappfile>/etc/proxyfilter_webapp.xml</webappfile>
 5      <logfile>/var/log/httpd/proxyfilter_log</logfile>
 6     <loglevel>info</loglevel>
 7      <http_methods>GET,POST,HEAD</http_methods>
 8     <headers_in default="filter">
 9        <allow>
10            <header name="host" />
11             <header name="connection" value="~^(Keep-Alive|close)$" />
12            <header name="accept" />
13             <header name="referer" maxlength="100" />
14             <header name="if-modified-since" maxlength="50" />
15             <header name="user-agent" maxlength="200" />
16             <header name="content-length" maxlength="20" />
17             <header name="content-type" maxlength="50" />
18            <header name="pragma" />
19        </allow>
20        <filter>
21            <header name="pragma" />
22            <header name="cookie" />
23            <header name="ua-cpu" />
24            <header name="ua-os" />
25        </filter>
26     </headers_in>
27     <headers_out default="allow">
28        <filter>
29            <header name="server" />
30             <header name="powered-by" />
31        </filter>
32     </headers_out>
33  </config>
```

Figure 4.3-18 : exemple de fichier de configuration global de ProxyFilter

L'élément racine du fichier est <config>. Les premières directives définissent l'emplacement sur le disque des divers fichiers de configuration : fichier *charsets*, fichier *mappings*, fichier *webapp* et fichier d'historique. Ce dernier est facultatif : si aucun fichier d'historique n'est indiqué, *ProxyFilter* écrira dans le *ErrorLog* de Apache.

ProxyFilter gère plusieurs niveaux de verbosité au niveau de son historique (ligne 6), dans l'ordre du plus détaillé au moins détaillé : *debug*, *info*, *warning*, *error*. Le niveau *debug*, assez lent, est destiné à être utilisé durant la phase d'établissement des règles car il donne des informations détaillées sur la requête et la façon dont elle est traitée. Le niveau *info* émettra des messages en cas de requête refusée, mais restera siliencieux pour les requêtes acceptées. Le niveau *warning* ne retiendra que les messages d'avertissement en cas d'erreur non-fatale, et le niveau *error* ne générera des messages dans l'historique qu'en cas d'erreur fatale empêchant le bon fonctionnement du module (une erreur syntaxique dans un fichier de configuration, par exemple).

Un filtrage basique des méthodes HTTP s'appliquant à l'ensemble de l'application est également proposé (ligne 7). La plupart des applications Web se contenteront de GET et POST, éventuellement HEAD. *ProxyFilter* n'a pas été testé avec d'autres méthodes, et les extensions particulières à HTTP comme WebDAV et les requêtes PUT ne sont pas supportées. L'idée de cette fonction est surtout, si une application n'utilise à aucun endroit la méthode POST, de permettre de la bloquer au niveau du proxy.

L'élément <headers_in> (lignes 8 à 26) est un conteneur permettant de filtrer les entêtes de la requête en provenant du client. Le *default policy* est à *filter*, ce qui signifie que toute entête qui ne correspond à aucune règle <allow> ou <deny> est filtrée de la requête par défaut. À noter que le comportement par défaut *deny* n'est pas recommandé pour les entêtes entrantes, car les différents navigateurs Web utilisent des entêtes radicalement différentes et la présence d'une entête inconnue provoquerait un refus de la requête.

L'élément <headers_out> (lignes 27 à 32) est identique à <headers_in> sauf qu'il concerne, on le devine, le filtrage des entêtes dans la réponse du serveur.

Les règles <header> sont syntaxiquement identiques aux règles <file> : une entête comporte un nom (insensible à la casse des caractères, comme défini dans la norme HTTP), une plage de valeurs définie de manière exacte ou par une expression régulière, et peut être optionnellement limitée en longueur au moyen des attributs *length*, *minlength* ou *maxlength*.

Dans l'exemple de la figure 4.4-15, toutes les entêtes provenant du serveur sont autorisées, sauf les entêtes *Server* et *Powered-by* qui sont filtrées afin de ne pas trop divulguer d'informations sur les logiciels exploités par le serveur.

Il existe certaines entêtes particulières comme *Cookie*, *SetCookie*, *Accept-Charset*, *Accept-Encoding* et *Accept-Language* qui peuvent être multivaluées. Pour de raisons de cohérence et de simplicité, ce cas ne fait pas l'objet d'un traitement particulier par la syntaxe de configuration de *ProxyFilter* : selon la norme HTTP, une entête multivaluée s'écrit sur une seule ligne et voit ses valeurs séparées par un point-virgule. Il suffit dès lors d'écrire une expression régulière correspondante, certe un peu complexe, pour définir une règle autorisant ce type d'entête :

```
Set-Cookie = php_session=bc6ebf0381c81780bf66e45ec68255d1;
expires=Thu, 11-Dec-03 23:27:21 GMT

<header name="Set-Cookie" value="~^php_session=[0-9a-z]{32}
(;\s?expires=[-:0-9a-zA-Z\s]{10-30})?$" />


X-Meta-Keywords = security, hackers, hacking, XSS attacks, buffer overflow

<header name="X-Meta-Keywords" value="~^([-_a-zA-Z0-9\s]+(,\s?)?)*$"
maxlength="250" />
```

Figure 4.3-19 : entêtes multivaluées et règles <header> correspondantes

Pour plus de détails sur la syntaxe de configuration, on pourra consulter l'exemple complet de configuration plus loin, ainsi que les fichiers DTD décrivant la syntaxe dans les annexes.

4.3.9. Syntaxe de réécriture d'URL

Un *reverse proxy*, même basique, doit offrir des fonctions de réécriture d'URL puisque la requête est redirigée vers un autre serveur (directive *ProxyPass* de mod_proxy). S'il est vraiment minimaliste, il se contente de réécrire l'URL en remplaçant le nom d'hôte par celui du serveur cible. S'il est plus évolué comme se veut *ProxyFilter*, il peut offrir une réécriture d'URL basée sur le préfixe de l'URL source, permettant ainsi de mapper plusieurs serveurs dans un même espace d'URL.

Pour pouvoir fonctionner, *ProxyFilter* a besoin d'au minimum une règle de réécriture d'URL définie. Ces règles sont définies dans le fichier *proxyfilter_mappings* (nom par défaut, modifiable) dont voici un exemple :

```
/                          http://www.exemple.com/                    forward
http://www.exemple.com/    /                                          reverse
/mail                      http://www.exemple.com/cgi-bin/mail.cgi    exact
/images/                   http://pics.exemple.com/                   forward
```

Figure 4.3-20 : syntaxe de réécriture d'URL (mappings)

Chaque ligne du fichier comporte trois colonnes séparées par des tabulations : l'URL source, l'URL cible et le type de règle (*forward*, *reverse* ou *exact*).

Les règles *forward* correspondent à la directive *ProxyPass* de mod_proxy : elles sont utilisées pour réécrire l'URL de la requête. Si un préfixe de la première colonne est trouvé dans l'URL, il est rempacé par sa valeur correspondante dans la seconde colonne. Attention : l'URL source est relative à la racine (c'est une URI au sens de Apache et *ProxyFilter*), alors que l'URL de destination doit être absolue, car elle indique le protocole (HTTP ou HTTPS) et le nom du serveur cible.

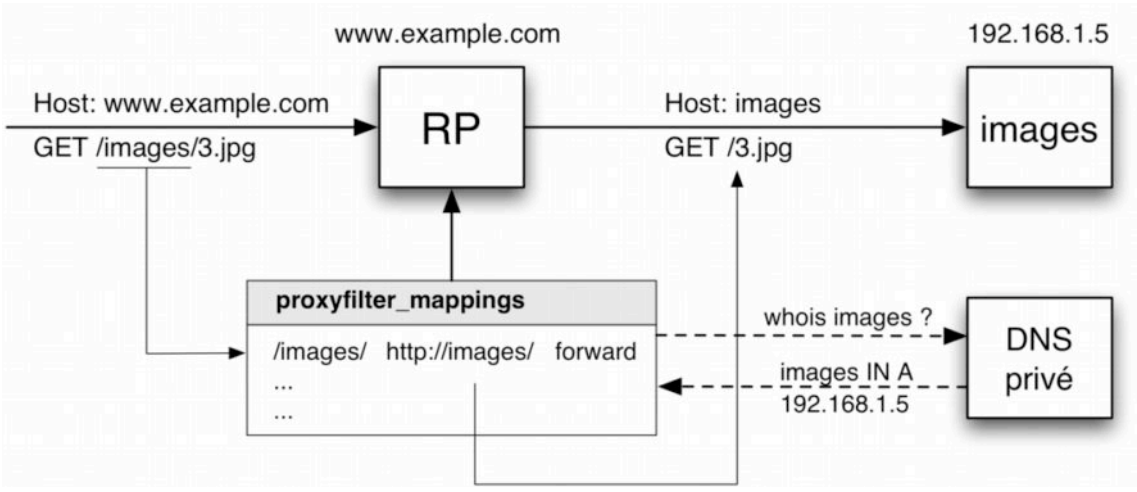


Figure 4.3-21 : principe d'une réécriture d'URL de type "forward"

Les règles *reverse* sont utilisées pour gérer les éventuels entêtes *Location* renvoyés par le serveur au client. Ceux-ci signalent une redirection au client, et sont généralement accompagnés d'un code de statut HTTP en 3xx. Si le serveur, qui n'a pas connaissance de l'URL originale demandée par le client, utilise une adresse absolue contenant une référence sur son propre nom d'hôte dans l'entête *Location*, alors la redirection va échouer car le client va tenter d'accéder directement au serveur Web sans passer par le proxy, ce qui est interdit par un firewall IP. De plus, le nom du serveur cible n'est pas forcément défini dans les DNS publics, car cela peut être un nom de machine à usage interne de l'entreprise ou une simple entrée *host* sur la machine hébergeant le *reverse proxy*.

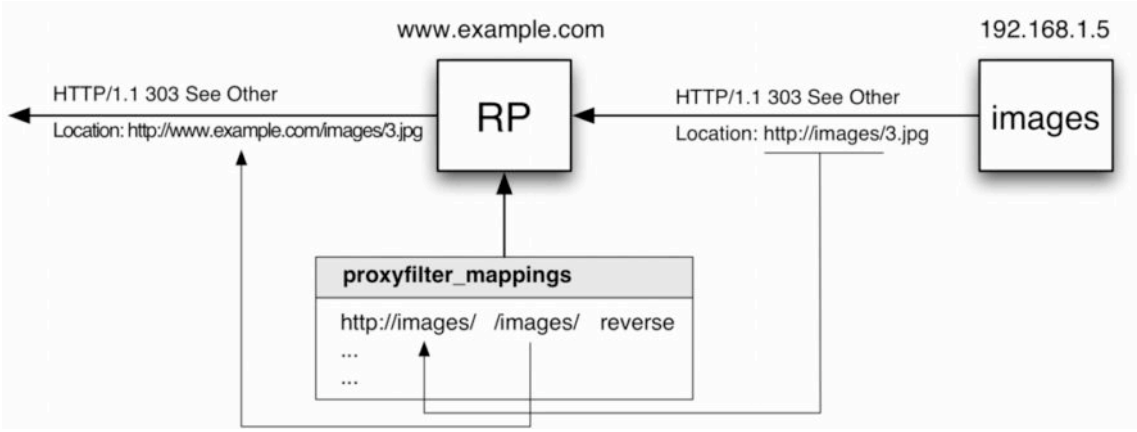


Figure 4.3-22 : principe d'une réécriture d'URL de type "reverse"

Enfin, les règles *exact* sont très semblables aux règles *forward*, sauf que la recherche d'URL source se fait de manière exacte au lieu d'être basée sur un préfixe. Ainsi, avec les règles :

```
/         http://server/                      forward
/mail     http://server/cgi-bin/mail.cgi      exact
```

Une requête `/mail` est effectivement redirigée vers `/cgi-bin/mail.cgi` sur le serveur cible par la seconde règle, mais une requête `/mail/index.html` est redirigée vers `/mail/index.html` sur le serveur cible par la première règle. Remarquons au passage qu'une règle *forward* ou *reverse* se termine toujours par une barre oblique (*trailing slash*) au niveau du préfixe de l'URL source et de destination, alors qu'une règle *exact* est généralement utilisée pour mapper une URL de fichier vers un autre fichier.

Caractéristique intéressante, les paramètres transmis à la fin de l'URL (méthode GET) sont gérés séparément de la réécriture d'URL. Ainsi, dans l'exemple précédent, une requête à `/mail?to=foo@bar.com` est réécrite en `/cgi-bin/mail.cgi?to=foo@bar.com`.

De même il est possible d'indiquer des paramètres dans l'URL réécrite d'une règle *exact* :

```
/news/2002     http://server/cgi-bin/news.cgi?year=2002   exact
```

Ainsi, une requête :

```
GET  /news/2002?newsid=3
```

est réécrite en :

```
GET  /cgi-bin/news.cgi?year=2002&newsid=3
```

Donc, les paramètres effectifs de la requête sont fusionnés de manière élégante avec ceux définis au niveau d'une règle *exact*. Comme la réécriture d'URL est effectuée avant le filtrage d'URL, les paramètres GET ajoutés à la requête dans une règle *exact* seront également filtrés et devront avoir une règle <param> correspondante pour que la requête soit acceptée. Il n'est pas possible d'ajouter des paramètres POST dans les règles de réécriture d'URL, ceux-ci sont gérés séparément.

4.3.10. Syntaxe de définition des *charsets*

Cette fonctionnalité est inspirée de Sanctum AppShield. Lorsque l'on définit des règles pour certaines parties de l'URL ou des champs de formulaire, il arrive que des suites ou jeux de caractères reviennent régulièrement dans l'application. Ainsi, les caractères autorisés dans une adresse e-mail, un champ de mot de passe ou simplement dans un nom de fichier sont connus et sont des constantes pour toute l'application.

Afin d'éviter de devoir redéfinir dans chaque règle des jeux de caractères complexes, *ProxyFilter* permet de ne les définir qu'une seule fois et de les utiliser ensuite facilement.

Voici un exemple de fichier *proxyfilter_charsets* (nom par défaut, modifiable) :

```
email                [-_\.a-z0-9]+@[-a-z0-9]\.[a-z]{2,3}
filename             [a-zA-Z0-9][-_.a-zA-Z0-9]*
digit                [0-9]
number               [0-9]+
xss                  <script>[^<]+</script>
tel                  [0-9/\+\s]+
```

Figure 4.3-23 : fichier de définition des charsets

Chaque ligne du fichier comporte deux colonnes : la première colonne donne le nom du *charset*, la seconde sa valeur en expression régulière compatible Perl. Lorsque, dans une règle des fichiers *webapp* et *config*, une référence à un *charset* est rencontrée, elle est remplacée par sa valeur.

Une référence à un *charset* est constituée du nom du *charset* précédé et suivi d'un caractère '%' (pourcent). Ce caractère a été choisi car il est peu probable qu'il doive se trouver dans une URL ou un nom de fichier. Le nom du *charset* ne peut contenir que des lettres minuscules et majuscules, des chiffres ainsi que les caractères tiret (-) et soulignement (_).

Par exemple, pour limiter les caractères dans le nom d'un fichier JPG, l'une des formes suivantes conviendra, la première utilisant une expression fixe avec *charset*, la seconde une expression régulière avec *charset* :

```
<file  name="%filename%.jpg"  maxlength="100"  />
<file  name="~^%filename%\.jpg$"  maxlength="100"  />
```

Cette règle est équivalente, sans *charset*, à écrire :

```
<file  name="~^[a-zA-Z0-9][-_.a-zA-Z0-9]*\.jpg$"  maxlength="100"  />
```

De même, pour un champ de formulaire devant contenir un numéro de téléphone européen :

```
<param  name="tel"  value="%tel%"  maxlength="15"  method="post"  />
```

Qui est l'équivalent de :

```
<param  name="tel"  value="~^[0-9/\+\s]+$"  maxlength="15"  method="post"  />
```

À noter que la référence à l'expression régulière est remplacée au caractère près par sa valeur, et donc qu'elle ne devrait pas comporter d'ancres de début (^) ou de fin ($) d'expression, il est préférable de les définir directement dans la règle. De même, on préférera souvent utiliser un opérateur de répétition générique (* ou +) dans un *charset* plutôt qu'une forme limitant le nombre de caractères comme {1-50} puisqu'il est possible de limiter la longueur de l'expression au moyen des attributs *length*, *minlength* ou *maxlength* d'une règle. L'idée des *charsets* est de favoriser au maximum leur réutilisation.

4.3.11. Lecture des fichiers de configuration

Après avoir défini la syntaxe employée par les 4 fichiers de configuration, dont 2 au format XML et 2 au format texte tabulé, la première tâche de programmation effective était de lire ces fichiers en mémoire et de construire une structure de données arborescente permettant de rechercher rapidement une information. Mon idée était que les fichiers de configuration du module soient lus une fois au démarrage du serveur, et qu'ensuite toutes les requêtes soient traitées en utilisant les informations stockées en mémoire.

Alors qu'il existe d'excellents dévermineurs pour la programmation Perl classique, permettant de fixer des points d'arrêt et de connaître l'état des variables à chaque étape du programme, les programmes Perl tournant dans l'environnement de Apache et mod_perl sont beaucoup plus difficiles à dépanner. En effet, ils ne peuvent pas simplement être exécutés à la ligne de commande : ils doivent tourner à l'intérieur d'un serveur Apache dont ils utilisent l'API, et la moindre erreur de programmation peut faire "planter" tout le serveur Apache sous Windows, ou au mieux le processus enfant concerné sous Unix.

Pour ces raisons, *ProxyFilter* a tout d'abord été développé comme un programme Perl classique exécuté depuis la ligne de commande, pour ensuite migrer plus tard vers un module Apache. Cela a considérablement facilité le processus de développement, du moins pour toute la partie de réécriture et de filtrage de l'URL (difficile d'émuler les entêtes HTTP à partir de la ligne de commande).

La technique de programmation utilisée pourraît être qualifiée de "programmation défensive", c'est à dire ajouter peu à peu des fonctionnalités et vérifier à chaque étape que le programme tourne encore, tout en gardant précieusement les versions intermédiaires pour pouvoir y revenir en cas de "pépin".

Comme la partie centrale de l'algorithme qui assure la fonction de reverse proxy, ne pouvait pas fonctionner avant que le programme ne soit migré en module Apache, il était plus approprié de suivre un développement *bottom-up* au lieu de *top-down*, c'est à dire de commencer par écrire des fonctions de détail, les tester individuellement, puis de tout assembler à la fin.

4.3.12. Fonctions d'évaluation des règles

Le langage Perl, très peu typé, ne connaît que 3 types de données : les scalaires, les listes (tableaux) et les *hashs* (tables de hashage). Les *hashs* sont des tableaux indicés au moyen de chaînes de caractères au lieu d'entiers. *ProxyFilter* utilise les *hashs* de manière intensive pour mémoriser les informations de configuration et les règles.

Ainsi, par exemple, l'ensemble des règles <file> est mémorisé dans un seul *hash* qui a la structure suivante :

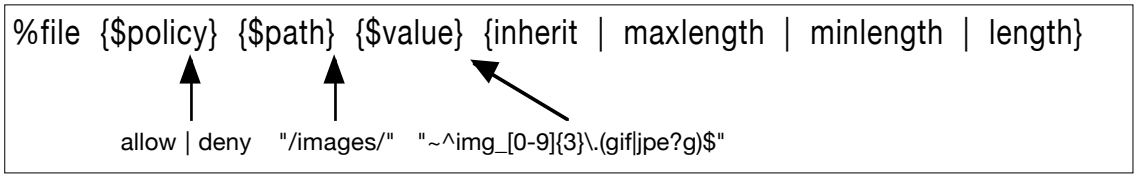


Figure 4.3-24 : structure de données arborescente pour mémoriser les règles <file>

La structure est composée de *hashs* contenus les uns dans les autres, formant ainsi une sorte d'arbre. Le premier niveau `$policy` permet de sélectionner les règles *allow* ou *deny*. `$path` renseigne sur le contexte (répertoire) dans l'application. Le troisième niveau `$value` contient la valeur de la règle proprement dite qui correspond au paramètre *name* de la règle <file>. Enfin, le dernier niveau de l'arbre contient les paramètres de la règle, à savoir si elle doit être héritée par les sous-répertoires (*inherit*) et les bornes de longueur du nom de fichier (**length*), facultatives.

Une fois cette structure de données construite en mémoire à partir des fichiers de configuration, il est aisé pour le programme de rechercher, par exemple, toutes les règles *allow* qui s'appliquent au répertoire `/images/`, et ensuite de les parcourir une à une pour déterminer si l'une d'elle s'applique au nom de fichier de la requête.

Ce travail est effectué par une collection de fonctions spécialisées qui sont :

`matchFile()`	Teste si un nom de fichier satisfait une règle <file>
`matchScript()`	Teste si une requête de script satisfait une règle <script>
`matchUrl()`	Teste si une URL satisfait une règle <url>
`matchParam()`	Teste si un paramètre de script satisfait une règle <param>
`matchHeader()`	Teste si une entête HTTP satisfait une règle <header>

Ces fonctions sont atomiques : elles ne permettent de vérifier qu'une seule règle à la fois. Pour parcourir toutes les règles qui s'appliquent au contexte donné, il faut chacune les appeler plusieurs fois. J'ai donc ensuite défini des fonctions qui, sur la base de l'ensemble des règles d'une catégorie (par exemple toutes les règles <file>) retourne un "verdict". Ce verdict peut valoir "accepter", "refuser" et, dans le cas des règles <url>, il peut également valoir "neutre" pour indiquer que les règles ne permettent pas de se prononcer.

Les fonctions qui retournent un verdict sont :

`fileAllowed()`	Confronte la requête à l'ensemble des règles <file>
`scriptAllowed()`	Confronte la requête à l'ensemble des règles <script>
`urlAllowed()`	Confronte la requête à l'ensemble des règles <url>

Pour obtenir la liste des règles s'appliquant à un répertoire donné, il ne suffit pas de rechercher le répertoire dans la structure de données : il faut également retenir les règles des répertoires parents dont le paramètre *inherit* vaut *true*. Par exemple, si l'on recherche toutes les règles *allow* qui s'appliquent dans /images/, il faut aussi rechercher les règles *allow* dans / qui auraient un attribut *inherit* = *true*. Les règles <url> n'ont pas d'attribut *inherit*, car elles sont de toute façon globales à l'application.

Le travail de recherche des règles qui s'appliquent à un répertoire est réalisé par les fonctions :

`getFiles($pwd,$policy)`	Retourne les règles <files> de type $policy dans le répertoire $pwd
`getScripts($pwd,$policy)`	Retourne les règles <script> de type $policy dans le répertoire $pwd
`getUrls($policy)`	Retourne les règles <url> de type $policy
`getDefault($pwd)`	Retourne le *default policy* pour le répertoire $pwd
`getAllowIndex($pwd)`	Retourne si l'index est autorisé pour le répertoire $pwd

Ces fonctions écrites et testées individuellement en injectant des URLs comme paramètre à la ligne de commande, il restait à implémenter le filtrage des entêtes HTTP en entrée et sortie, ainsi qu'à migrer le code vers un module Apache et écrire le programme principal remplissant la fonction de *reverse proxy*.

4.3.13. Développement d'un reverse proxy simple

Avant même de penser à ajouter des fonctions de contrôle et de filtrage, l'idée était de mettre au point un simple module reverse proxy non-filtrant. Dans ce domaine, il existe sur le site du CPAN quelques exemples de modules (*WebProxy*, *ProxyPass*) dont il était bon de s'inspirer, et un chapitre du livre *Writing Apache Modules With Perl* donne un exemple de code Perl permettant de faire un reverse proxy. Cependant, chacun y va de sa petite cuisine et les implémentations sont assez diverses : il s'agissait de reprendre le meilleur de ces codes sources pour construire ma propre version, qui soit la plus compatible possible avec toutes les subtilités d'implémentation du protocole HTTP par les différents serveurs, navigateurs et applications.

Mon premier module, Apache::PassThru, tient sur quelques lignes. Il s'installe durant la phase *URI translation* et utilise le *handler* de mod_proxy pour gérer la phase de réponse.

```
package Apache::PassThru;
use strict;
use Apache::Constants qw(:common);

my %proxymappings = ("/" => "http://server.local/");

# PerlTransHandler
sub handler {
   my $r = shift;
   return DECLINED if $r->proxyreq;
   my $uri = $r->uri;
   for my $source (keys %proxymappings) {
      next unless $uri =~ s/^$source/$proxymappings{$source}/;
      $r->proxyreq(1);
      $r->uri($uri);
      $r->headers_in->{'User-Agent'} = "Apache::PassThru";
      $r->filename("proxy:$uri");
      $r->handler('proxy-server');
      return OK;
   }
   return DECLINED;
}
1;
__END__
```

Figure 4.3-25 : Module reverse proxy simple

L'avantage d'utiliser mod_proxy est que le code est très compact, car il se contente de reconnaître si une URL source doit être passée au proxy en fonction de préfixe, et règle le *handler* de mod_proxy pour gérer la phase de réponse le cas échéant. Tout le reste du travail (composition et envoi de la requête interne, réception de la réponse, copie des entêtes et du contenu) est géré par mod_proxy lui-même.

Remarquons que ce code n'a pas grand intérêt : il est équivalent aux directives suivantes :

```
<IfModule  mod_proxy.c>
    ProxyRequest  on
    ProxyPass   /   http://server.local/
</IfModule>
```

Le cahier des charges demande à notre module d'utiliser si possible les services de mod_proxy, ce qui semble à priori une bonne idée pour ne pas avoir à réinventer un proxy de toute pièce et pouvoir se concentrer sur l'aspect sécurité. Il y a cependant une limitation à l'utilisation de mod_proxy : notre module ne gère pas la phase de réponse, par conséquent il n'a aucun contrôle sur ce qui est renvoyé par le serveur, ni les entêtes, ni le contenu. Cela est innaceptable puisque nous voulons pouvoir contrôler des paramètres tels que le type MIME du document retourné, filtrer certaines entêtes qui donneraient trop d'informations sur le serveur, vérifier certains mot-clés dans la page retournée, etc...

Il faut donc se tourner vers une autre solution qui nous permette de gérer nous-même la phase de réponse. Après quelques recherches, le module Perl LWP::UserAgent, basé sur libwww, s'est avéré être la solution idéale. Il implémente un véritable client HTTP, HTTPS et FTP, permettant à un programme Perl d'effectuer des requêtes finement paramétrées vers un serveur Web ou FTP, de recevoir et traiter la réponse. Mon choix s'est en particulier porté sur ce module, car un exemple du livre *Programming Apache Modules With Perl* l'utilise pour construire un reverse proxy personnalisé, exactement la solution dont nous avons besoin.

Mon deuxième module, *PassThru2*, fait donc appel à LWP::UserAgent. Le code source, trop long pour être inclu ici, est disponible dans les annexes. Cette version comporte deux *handlers* : l'un pour la phase *URI translation* qui détermine si l'URL doit être passée au reverse proxy, et l'autre pour la phase *Response* qui compose la requête interne, la passe à LWP et reçoit la réponse qu'il renvoie au client.

L'inconvénient d'installer le module durant la phase *URI translation* est de prendre la main sur des modules comme mod_proxy et mod_rewrite avec lesquels nous aimerions être compatible. L'avantage est de pouvoir déterminer très tôt si la requête doit ou ne doit pas être passée au *reverse proxy*. Pour permettre une utilisation conjointe avec mod_proxy, mod_alias et mod_rewrite, les versions suivantes du module se contentent de gérer la phase *Response* et n'interviennent plus durant la phase *URI translation*.

À ce stade, le module n'utilise aucun fichier de configuration et la table de réécriture d'URL est codée en dur. De plus, la réécriture des entêtes *Location* dans la réponse c'est pas gérée, ce qui crée des problèmes dès que l'on essaie d'accéder par le proxy à une application qui utilise des adresses absolues pour signaler qu'un document a changé d'adresse.

Un autre point critique à gérer concerne d'une part les entêtes multivaluées, d'autre part les entêtes présentes à plusieurs exemplaires. En effet, certaines entêtes comme *Cookie* ou *Accept-Language* peuvent comporter plusieurs valeurs, et le risque est de ne copier que la première d'entre-elles dans l'objet requête de LWP si l'on n'y prend pas garde.

Bien que celui ne soit que rarement le cas, la norme HTTP décrit la possibilité d'avoir, dans une requête ou une réponse, plusieurs fois la même entête. C'est le cas par exemple avec l'entête *Proxy-Via* qui permet de savoir par quels proxys a passé la requête : au lieu d'ajouter sa valeur à la suite de l'entête *Proxy-Via* existante, un proxy va avoir tendance à rajouter une nouvelle entête *Proxy-Via*. Même chose pour de multiples cookies qui peuvent soit se présenter sous la forme d'une seule entête multivaluée, soit sous la forme de plusieurs entêtes, une par cookie.

Si l'on se contente d'extraire les entêtes de la requête pour les mémoriser dans une variable *hash* de Perl en utilisant le nom de l'entête comme clé, plusieurs apparitions de cette entête se détruiront mutuellement puisqu'un *hash* ne tolère pas de doublons au niveau des clés. Pour éviter cela, l'artifice est d'utiliser la classe Apache::Table, utilisant des clés insensibles à la casse et pouvant

exister à plusieurs exemplaires. Une autre solution, plus compliquée, aurait consisté à mémoriser les entêtes sous la forme d'un tableau à deux dimensions.

Sur ces derniers points, le code de *PassThru2* est imparfait. Il a donc fallu l'améliorer pour aboutir à une troisième version du module qui a enfin eu le droit de s'appeler *ProxyFilter*, car elle est à l'origine de la version finale du module.

4.3.14. Migration vers un module Apache

Une fois la fonction de *reverse proxy* suffisament élaborée et compatible, il a fallu migrer les fonctions de lecture et d'évaluation des règles précédemment développées sous la forme d'un script Perl autonome vers le module Apache.

Logiquement, les premières fonctions à migrer étaient celles chargées de lire la configuration du module, et notamment les *handlers* appelés par le *parser* XML lorsque les divers éléments sont rencontrés dans le fichier de configuration.

Malheureusement, il s'est avéré que le même code pour le *parser* qui fonctionnait très bien comme script Perl autonome faisait carrément "planter" le serveur Apache ! La bibliothèque XML::Parser était pourtant bien reconnue, une instance de l'objet *parser* était obtenue sans erreur, mais c'est au moment de parcourir le document qu'une erreur se produisait. Dans le *Error-Log* de Apache, on observait le message suivant à répétion, alors que le serveur tentait de relancer à chaque seconde un processus enfant qui plantait aussitôt :

```
[notice] child pid 1127 exit signal Bus error (10)
```

Face à ce comportement inexpliquable, je n'avais pas de solution. Après avoir essayé de simplifier le module au maximum en ne laissant que le code utile au *parser*, cela ne fonctionnait guère mieux. Le comportement était le même avec un script CGI tournant dans Apache::Registry et tentant d'utiliser le *parser*, mais cela fonctionnait si le *parser* était appelé depuis un simple CGI exécuté par mod_cgi, et donc ne faisant pas appel à mod_perl.

Après plusieurs **jours** de demande d'aide dans les forums, recherche dans les documentations de Apache, mod_perl, Perl et Expat, échanges de e-mails avec les développeurs des modules respectifs, toujours aucune solution n'était trouvée à ce problème qui bloquait complétement la progression de mon développement.

C'est alors que mon ami Marc Liyanage, ingénieur en informatique chez futureLAB AG, me suggère d'essayer avec un autre *parser* comme XML::LibXML ne faisant pas appel à Expat, car **Apache lui-même utilise Expat**.

Quelques tests avec cet autre *parser* confirment que Marc a vu juste : il est alors possible de parcourir un fichier XML depuis un module Perl, mais je n'ai pas envie de modifier les quelques 800 lignes de mon code qui sont chargées d'interpréter les fichiers XML pour les adapter à XML::LibXML, car celui-ci est un *parser* de type DOM, alors que mon algorithme était prévu dès le départ pour un *parser* en mode flux.

Renseignement pris, il semble qu'il soit possible à la compilation de Apache de demander de ne pas inclure le support de Expat, et donc que Apache ne tente pas d'utiliser cette bibliothèque créant un conflit d'accès avec XML::Parser. Soit-dit en passant, la raison pour laquelle Apache fait appel à un *parser* XML alors que sa syntaxe de configuration n'a rien de XML et qu'il

n'intègre pas de support pour SOAP m'est inconnue, et donc le support de Expat dans Apache ne me semblait pas indispensable.

Puisque la version 1.3.27 de Apache livrée précompilée avec Mac OS X ne me permettait pas d'accéder à Expat depuis un module Perl, j'ai donc entrepris de compiler ma propre version de Apache, ce qui n'était pas une sinécure. Du même coup, la version 5.6 de Perl que j'utilisais, elle aussi livrée avec le système Mac OS X, s'est révélée incompatible avec mon installation de Apache, et mod_perl également qu'il a fallu recompiler (heureusement que tous ces logiciels sont *open source*).

La documentation officielle de mod_perl conseille, pour plus de performance et de compatilbilité, de compiler mod_perl directement dans l'exécutable du serveur Apache, au lieu de le compiler comme un objet dynamique partagé (*DSO, Dynamic Shared Object*). Comme l'exécutable du Apache 1.3.27 dont je disposais à l'origine était compilé avec le strict minimum de modules inclus et que tous les modules annexés étaient sous forme de DSO, c'était pour moi l'occasion d'inclure la dernière révision 1.3.29 de mod_perl et d'autres modules comme mod_ssl, mod_proxy, mod_rewrite et mod_cgi à ma propre version de Apache, profitant ainsi d'une optimisation bienvenue, mais perdant en souplesse dans le cas où je n'aurais plus eu besoin de ces modules.

Après 5 jours de recherche, tests et recompilations en tous genres, XML::Parser a enfin daigné fonctionner lorsqu'il était appelé depuis un module Perl ! Cet obstacle surmonté, la suite de l'importation des fonctions *allowed(), match*() et get*() s'est effectuée sans trop de problèmes.

On retiendra cependant une difficulté liée aux variables globales : comme, sous Unix, Apache est une application multiprocessus (et non multithreads comme sous Windows), chaque processus enfant dispose de son propre espace de mémoire. Ainsi, une variable globales définie dans un module Perl pour Apache n'est pas vraiment globale, il en existera une copie pour chaque processus enfant, et l'utilisation de variables globales est fortement déconseillée, elle peut amener à des comportements inattendus. Je l'ai constaté avec *ProxyFilter* en essayant de mémoriser la configuration dans des variables `%config`, `%file`, `%script`, `%url`, `%allowindex` et `%default` globales dans l'espoir que leur contenu soit le même pour les différents processus enfants, mais le contenu des *hashs* se modifiait comme par magie d'une requête à l'autre et ne correspondait plus du tout à celui des fichiers de configuration !

Une solution consiste à tourner Apache en mode *single thread* par la commande `httpd -X` au lieu de l'utilitaire `apachectl`, mais ce mode n'est destiné qu'au déverminage et n'est pas utilisable en production car il limite trop les performances du serveur.

Une autre solution, plus raisonnable, est de ne simplement pas utiliser de variables globales, c'est à dire que toutes les variables doivent être déclarées, ou du moins initialisées dans le *handler* à chaque requête. Dès lors, il n'est plus possible de lire les fichiers de configuration une seule fois au démarrage du serveur, il faut relire la configuration sur le disque à chaque requête, ce qui pénalise un peu les performances, c'est pourtant la solution qui a été retenue pour la version actuelle de *ProxyFilter*, dans l'attente de trouver mieux.

4.3.15. Dernière main au programme principal

Le programme principal du module est en fait le *Response handler*. C'est de là que sont invoquées toutes les autres fonctions, à commencer par la procédure d'initialisation qui lit les fichiers de configuration. Les divers tests sur la requête se font dans un ordre qui me semble logique : on commence par vérifier les entêtes, puis on réécrit l'URL, on vérifie que l'URL réécrite corresponde à un répertoire valide de l'application Web, on vérifie que le fichier soit autorisé dans ce répertoire, et si la requête comporte des paramètres GET ou POST, on recherche une règle <script> et des règles <param> correspondantes. À chacune de ces étapes de validation, la requête peut être refusée provoquant l'affichage d'un message "403 Forbidden" et ce n'est qu'une fois toutes les étapes de validation passées avec succès que l'on entreprend de composer un objet requête destiné à être soumis à LWP pour qu'il le transmette au serveur. Lorsque la réponse du serveur revient (ou ne revient pas, c'est selon), les entêtes sont contrôlées et filtrées et la réponse est renvoyée au client.

La mise au point du programme principal était également l'occasion de soigner la fonction d'historique : jusqu'à présent, les messages émis dans le *log* étaient surtout une façon de déverminer le module en l'absence d'autre moyen, ils deviennent maintenant des outils d'établissement des règles et d'analyse des requêtes refusées pour l'utilisateur final, avec plusieurs niveaux de priorité *debug*, *info*, *warning* et *error*. Les messages d'historique peuvent être écrits dans le *ErrorLog* de Apache ou dans un fichier séparé (recommandé).

4.3.16. Test de fonctionnement

Pour tester le bon fonctionnement du module, j'ai utilisé une application d'exemple *securitystore.ch*, développée en PHP dans le cadre du travail de semestre dans le but de faire la démonstration de quelques vulnérabilités courantes. L'avantage d'utiliser cette application est que, tout en restant simple, elle utilise un ensemble de fonctionnalités assez large du protocole HTTP (session au moyen de cookies, paramètres GET et POST, formulaires, redirections externes). De plus, comme elle est conçue vulnérable au départ, cela me permet de vérifier directement si la protection offerte par *ProxyFilter* est efficace.

Pour injecter des requêtes dans le *reverse proxy*, j'ai utilisé d'une part des navigateurs Web courants (Apple Safari 1.0 et Internet Explorer 5), d'autre part l'outil `curl` en ligne de commande, qui permet de paramétrer assez précisément la forme de la requête, d'effectuer des requêtes HEAD, de simuler un empoisonnement de cookies ou de paramètres, et d'afficher le document retourné avec ses entêtes. Pour les connaisseurs, `curl` est globalement équivalent à `wget`.

4.3.17. Test d'intrusion

Lorsque le module est devenu suffisament stable pour pouvoir supporter des requêtes quelque peu malicieuses, il est devenu possible de le soumettre à un véritable scanner de vulnérabilités.

Face au silence de Sanctum suite à mes demandes d'une licence d'évaluation de leur produit *AppScan*, j'ai voulu me tourner vers *N-Stalker* sur le conseil de M. Maret, mais il est malheureusement devenu un produit commercial et n'est de toute façon disponible que pour la plateforme Windows.

Mon choix s'est porté sur un scanner spécialisé dans les vulnérabilités du Web (bien qu'il teste également les autres ports et les services tels que MySQL) du nom de *Nikto*, entièrement écrit en

Perl et gratuit. *Nikto* dispose d'une base de données régulièrement mise à jour qui contient les vulnérabilités connues les principaux serveurs Web ou d'application. Il se base sur l'entête *Server* pour cibler ses requêtes en fonction du produit. Comme *ProxyFilter* inhibe l'entête *Server*, il est néanmoins possible d'indiquer manuellement à *Nikto* le logiciel serveur utilisé.

Le résultat du test de vulnérabilité de *securitystore.ch* réalisé par *Nikto*, avec et sans *ProxyFilter*, est présenté dans les annexes.

4.4. Fonctionnement

Suite au chapitre précédent expliquant dans le détail le processus de développement, mais qui peut paraître confu au lecteur qui souhaite juste à comprendre comment fonctionne *ProxyFilter* et comment le configurer et l'utiliser, nous faisons ci-après la syntèse de son fonctionnement, même si certaines informations peuvent sembler redondantes par rapport au chapitre qui précède.

Le travail de *ProxyFilter* peut être décomposé en 4 étapes : réception et scrutation de la requête externe provenant du client, composition et envoi de la requête interne au serveur cible, réception et scrutation de la réponse du serveur cible, composition et envoi de la réponse au client.

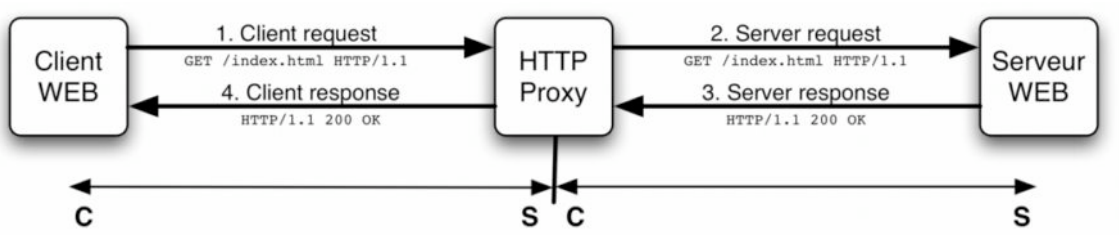


Figure 4.4-1 : fonctionnement en 4 étapes de ProxyFilter

ProxyFilter remplit la fonction de *Response handler* au sein de Apache, il laisse le soin à Apache ou à d'autres modules de gérer les étapes préliminaires de la requête telles que la lecture des entêtes, l'authentification, le contrôle d'accès.

Le *handler* de *ProxyFilter* est appelé par Apache durant la phase de réponse de chaque cycle. Son algorithme, de façon simplifiée, peut être résumé ainsi :

```
initialisation du module
filtrage en fonction de la méthode HTTP (GET, POST, etc...)
lecture et filtrage des entêtes de la requête
réécriture de l'URL de la requête pour pointer vers le serveur cible
lecture des paramètres GET et POST
filtrage global de l'URL (règles <url>)
filtrage en fonction du répertoire (path) de la requête (éléments <directory>)
si la requête comporte au moins un paramètre GET ou POST alors
    filtrage en fonction du nom de fichier et des paramètres (règles <script> et <param>)
sinon
    filtrage en fonction du nom de fichier (règles <file>)
fin de si
création de la requête HTTP interne (copie de l'URL et des entêtes)
instance d'un agent HTTP
envoi de la requête au serveur cible
réception de la réponse du serveur cible
filtrage des entêtes de la réponse (y.c. type MIME)
si la réponse comporte un contenu alors
    copie le contenu de la réponse
sinon
    copie le contenu de la réponse
fin de si
retourne la réponse au client
```

Figure 4.4-2: algorithme global de ProxyFilter

À chaque étape de filtrage du *handler*, le programme est susceptible d'interrompre la requête en retournant au client, selon les cas, un message "Forbidden" ou "Server Error". Le premier signifie que la requête est refusée parce qu'elle ne correspond pas aux critères de l'application. Le second indique qu'une erreur de configuration est survenue, par exemple si la syntaxe de configuration est erronée ou si un fichier de configuration n'a pu être ouvert.

4.4.1. Initialisation

Au début de chaque requête, le module lit ses fichiers de configuration en mémoire dans une structure de donnée adaptée à une consultation rapide par les diverses routines. Nous utilisons des éléments de types *hash* (tables de hashage) encapsulés les uns dans les autres pour mémoriser ces données. L'algorithme général de la procédure d'initialisation est le suivant :

obtenir le chemin d'accès au fichier de configuration principal
crée une instance du parseur XML
lire le fichier de configuration principal (`proxyfilter_config`)
lire le fichier de description de l'application (`proxyfilter_webapp`)
lire le fichier contenant les *charsets* (`proxyfilter_charsets`)
lire le fichier contenant les *mappings* (`proxyfilter_mappings`)

Figure 4.4-3 : algorithme d'initialisation de ProxyFilter

Un module Apache, qu'il soit écrit en Perl ou en C, peut définir des directives de configuration personnalisées à placer dans le fichier de configuration httpd.conf de Apache. Lorsqu'il démarre, le serveur vérifie la présence et la syntaxe de ces directives "par module" et génère une erreur si celle-ci est erronée. Comme *ProxyFilter* est un module qui nécessite beaucoup d'informations pour sa configuration, il était plus logique de les réunir dans des fichiers externes que dans le fichier *http.conf*, c'est pourquoi *ProxyFilter* se contente d'une seule directive *ProxyFilterConfig* à placer dans le fichier *httpd.conf* indiquant l'emplacement du fichier de configuration principal. L'emplacement des autres fichiers de configuration est indiqué au moyen de directives situées dans le fichier de configuration principal.

Voici une brève description des divers fichiers de configuration employés par *ProxyFilter* :

`proxyfilter_config`	Il s'agit du fichier de configuration principal dont la syntaxe est dérivée de XML. L'emplacement des autres fichiers de configuration, le filtrage des entêtes HTTP ainsi que les indications qui s'appliquent globalement à toute l'application y sont définis.
`proxyfilter_webapp`	Fichier décrivant en XML la structure hiérarchique de l'application Web, c'est à dire pour chaque répertoire les différents fichiers, les scripts et leurs paramètres, etc...
`proxyfilter_mappings`	Fichier en simple texte tabulé décrivant les règles de réécriture d'URL, c'est à dire pour chaque URL source, l'URL de destination. L'application ne peut pas fonctionner si au minimum une règle n'a pas été définie, car c'est ce fichier qui définit l'adresse du serveur cible.
`proxyfilter_charsets`	Ce fichier décrit les *charsets*, qui sont des ensembles de caractères pouvant être rencontrés dans les champs de formulaire ou les paramètres de script et qui sont définis une fois pour toute dans ce fichier pour pouvoir être réutilisés facilement dans les règles.

4.4.2. Filtrage de la méthode HTTP

Lorsqu'une requête arrive, *ProxyFilter* commence par vérifier que la méthode HTTP (GET, POST, HEAD, PUT, etc...) soit valide pour l'application. L'utilisateur peut définir les méthodes autorisées pour toute l'application au moyen de l'élément <http_methods> placé directement dans l'élément racine <config> du fichier *proxyfilter_config.xml*. Les méthodes doivent être séparées par des virgules, elles sont insensibles à la casse. Par défaut aucune méthode n'est autorisée.

```
<config>
    ...
     <http_methods>GET,POST,HEAD</http_methods>
    ...
</config>
```

Dans la plupart des cas, on se contentera des méthodes GET et POST, éventuellement HEAD. Les autres méthodes n'ont **pas** été validées pour être utilisées avec *ProxyFilter*.

4.4.3. Filtrage des entêtes de la requête

L'étape suivante est de vérifier les entêtes de la requête. Celles-ci sont définies de manière globale pour l'application dans le fichier proxyfilter_config.xml à l'aide du conteneur <headers_in> et de règles <header>. L'attribut *default* est obligatoire pour <headers_in>, il peut valoir *allow* (les entêtes sont acceptées par défaut), *deny* (les entêtes sont refusées par défaut) ou *filter* (les entêtes sont filtrées par défaut).

```
<config>
    ...
     <headers_in default="filter">
        <allow>
            <header name="Accept" maxlength="250" />
             <header name="Referer" value="~^https?://" maxlength="150" />
             <header name="Connection" value="~^(keep-alive|close)$" />
             <header name="Content-Length" value="%number%" maxlength="4" />
        </allow>
        <deny><
            <header name="Cookie" />
        </deny>
     </headers_in>
    ...
</config>
```

Dans l'exemple ci-dessus, à l'exception de *Accept*, *Referer*, *Connection* et *Content-Length*, toutes les entêtes de la requête sont filtrées (elles ne parviennent pas au serveur). Si une entête *Cookie* est présente, la requête est refusée (probablement que l'on sait que l'application n'utilise pas de cookies, et donc la présence d'une telle entête est assimilée à une tentative de *hacking*).

Dans une règle <header>, seul l'attribut *name* est obligatoire : il doit définir exactement le nom de l'entête, les expressions régulières ou *charsets* ne sont pas autorisé ici. L'attribut *value* définit la valeur de l'entête qui peut être exacte ou définie par une expression régulière ou une référence de *charset*. Pour indiquer la présence d'une expression régulière, on la préfixe d'un caractère tilde (~). Le nom et la valeur de l'entête sont insensibles à la casse, comme définit dans la norme HTTP. Si l'attribut *value* n'est pas présent, l'entête peut prendre n'importe quelle valeur.

Les attributs *maxlength*, *minlength* et *length* sont toujours facultatifs. Ils servent à borner le nombre de caractères autorisés dans la valeur d'une entête. La présence de *length* (longueur exacte) rend caduque *maxlength* et *minlength*.

Dans une règle <header> de type *deny*, seul le nom compte, les autres attributs sont ignorés. Cela signifie qu'il n'est pas possible, au moyen d'une seule règle, de filtrer par exemple toutes les entêtes dont la valeur aurait plus de 100 caractères. Sur ce point là, les règles <header> sont plus limitées que les règles <url>, <file> ou <script> pour lesquelles une règle *deny* peut définir autant d'attributs qu'une règle *allow*.

4.4.4. Réécriture de l'URL

L'opération suivante est de réécrire l'URL pour la faire pointer vers le bon serveur cible et le bon répertoire. Le fichier proxyfilter_mappings permet de définir les règles de réécriture d'URL. Pour que *ProxyFilter* puisse fonctionner, au moins une règle doit être définie, et toute requête pour laquelle il n'existe pas de *mapping* sera déclinée par *ProxyFilter* et prise en charge par le *Response handler* par défaut de Apache qui recherchera un fichier sur le disque du proxy.

```
/                    http://server.local/                      forward
http://server.local/ /                                         reverse
/webmail/            http://owa.local/                         forward
http://owa.local/    /webmail/                                 reverse
/stats               http://server.local/cgi-bin/stats.cgi     exact
/news                http://server.local/index.asp?page=news   exact
```

Le fichier comprend 3 colonnes : préfixe de l'URL source, préfixe de l'URL cible et type de règle (*exact*, *forward* ou *reverse*).

Une règle *forward* est utilisée pour remplacer un préfixe de l'URL de la requête par un autre préfixe, c'est l'équivalent de la directive *ProxyPass* de mod_proxy.

Une règle *reverse* est utilisée pour réécrire les entêtes *Location* utilisant des adresses absolues lorsque le serveur renvoie le client à un autre document au moyen d'une redirection externe, c'est l'équivalent de la directive *ProxyPassReverse* de mod_proxy. Généralement, on écrira une règle *reverse* pour chaque règle *forward*, mais cela n'est pas nécessaire si aucune redirection n'est générée par le serveur ou si la redirection est faite en utilisant des adresses relatives.

Une règle *exact* permet de réécrire l'URL de la requête en se basant sur une correspondance exacte de l'URL source au lieu de basée sur un préfixe. Ainsi, avec l'exemple de configuration ci-dessus, une requête `/stats` sera réécrite en `http://server.local/cgi-bin/stats.cgi` par la 5ème règle, mais `/stats/index.cgi` sera réécrite en `http://server.local/stats/index.cgi` par la 1ère règle, car la 5ème n'est pas satisfaite.

Dans de nombreux cas, plusieurs règles de réécriture s'appliquent. Par exemple, une requête `/webmail/index.php` pourrait être réécrite `http://server.local/webmail/index.php` au lieu de `http://owa.local/index.php` comme souhaité. Dans ce cas, **c'est la règle avec le préfixe qui correspond le plus long qui est retenue**. Ce comportement est valable aussi bien pour les règles *forward* que *reverse*, qui sont basées sur une recherche par préfixe.

4.4.5. Filtrage global de l'URL

À ce stade, les règles <url> sont évaluées en priorité. L'idée de ces règles est d'offrir, en cas de besoin, une souplesse que les règles <file> et <script> ne peuvent pas assurer. Ainsi, les règles <url> sont globales à toute l'application, elles ne dépendent pas du répertoire dans lequel est placé la règle dans le fichier *webapp* (il est d'ailleurs conseillé de placer les règles <url> au premier niveau à l'intérieur de l'élément <webapp>).

Cela permet d'autoriser certaines <url> spéciales pour lesquelles il n'existe pas même de bloc <directory> valides, car si une URL retourne un verdict "allow", la requête est autorisée sans même évaluer les règles suivantes. De même, une règle <url> de type *deny* peut être utilisée pour interdir certaines chaînes de caractères sur toute l'URL, paramètres GET compris, comme les attaques du ver Nimda ou pour limiter la longueur maximale des URLs afin de prévenir les attaques par de type *buffer overflow*.

```
<webapp default="deny" allowindex="true">
    <allow>
        <url value="~^/pub/(\?(M|S|D|N)=(A|D))?$" />
    </allow>
    <deny>
        <url value="~.*" minlength="300" />
        <url value="<script>" />
        <url value="&lt;script&gt;" />
        <url value="&#60;script&#62;" />
    </deny>
    ...
</webapp>
```

L'attribut *value* peut valoir une URL exacte, mais on utilisera plus souvent une expression régulière (préfixée du caractère tilde ~). Au sens de Apache, on devrait plutôt parler d'une URI, car elle ne comprend pas le protocole et le nom d'hôte, uniquement le chemin d'accès et les éventuels paramètres.

Dans l'exemple de configuration ci-dessus, la règle *allow* permet de supporter l'index du répertoire /pub généré par mod_autoindex de Apache, ce qu'il n'aurait pas été possible de faire au moyen d'une règle <script> placée dans un élément <directory> car mod_autoindex utilise des requêtes du genre :

```
GET /pub/?N=A
```

Au sens de *ProxyFilter*, le nom de fichier est vide, et donc il ne serait pas possible de définir une règle <script> correspondante. Ainsi, une application Web qui utilise une syntaxe inhabituelle dans les URLs pourra tout de même être supportée par *ProxyFilrer*.

Les règles <url> de type *deny* sont utiles pour bloquer des attaques de type XSS via des paramètres GET, *Directory Traversal*, Nimda, *Buffer Overflow*. Dans l'exemple ci-dessus, la première règle *deny* limite à 300 caractères la longueur de toutes les URLs de l'application, les 3 règles suivantes bloquent diverses formes d'attaques XSS.

Insistons sur le fait que, si une requête dont les entêtes sont valides satisfait une règle <url> de type *allow*, elle est immédiatement acceptée sans que les règles <file> ou <script> ne soient évaluées. De même, si la requête satisfait une règle <url> de type *deny*, elle sera refusée sans pouvoir être "repêchée" par une règle <file> ou <script>. De part leur niveau de priorité élevé, les règles <url> sont donc à manier avec précaution.

Si la requête ne satisfait aucune règle <url> de type *allow* et aucune règle <url> de type *deny*, la fonction chargée d'évaluer les règles <url> retourne un verdict "neutre" qui signifie que les règles <file> et <script> seront consultées pour décider si la requête est acceptée ou non.

4.4.6. Vérification du répertoire de la requête

Lorsque le test de règles URL retourne un verdict "neutre", *ProxyFilter* extrait le *path* de la requête, c'est à dire le chemin d'accès moins le nom de fichier. Exemple :

requête: GET /forum/images/avatar.gif
path: /forum/images/
fichier: avatar.gif

Le module vérifie qu'il existe dans le fichier *proxyfilter_webapp* les éléments <directory> correspondant au *path* de la requête, sans quoi celle-ci est refusée. Par exemple, pour la requête ci-dessus, il vérifie que la structure suivante soit définie :

```
<webapp default="deny" allowindex="true">
   ...
   <directory name="forum">
      ...
      <directory name="images">
         ...
      </directory>
   </directory>
</webapp>
```

Pour séparer le nom de fichier du *path*, *ProxyFilter* recherche la première barre oblique (*slash*) à partir de la fin du chemin d'accès. Il est donc évident qu'un nom de fichier n'a pas le droit de contenir une barre oblique (mais ce caractère est permis dans la chaîne de paramètres GET).

Comme l'élément <webapp>, les éléments <directory> peuvent prendre des attributs optionels *default* et *allowindex*. Si ces attribut ne sont pas présents, leur valeur est héritée du répertoire de niveau supérieur.

4.4.7. Vérification du nom de fichier

Si le *path* de la requête correspond à un répertoire valide de l'application, *ProxyFilter* recherche une règle <file> ou <script> qui corresponde au nom de fichier de la requête.

La différence entre les règles <file> et <script> est qu'une requête comprenant **au moins** un paramètre GET ou POST doit satisfaire une règle <script> alors qu'une requête qui ne comprend aucun paramètre doit satisfaire une règle <file>.

Reprenons l'exemple la requête ci-avant et ajoutons à la structure de données une règle <file> :

```
<webapp default="deny" allowindex="true">
   ...
   <directory name="forum">
      ...
      <directory name="images">
          <allow><file name="avatar.gif" /></allow>
      </directory>
   </directory>
</webapp>
```

Nous utilisons ici la règle <file> sous sa forme exacte. Si le dossier comporte des dizaines d'images GIF, il est un peu long de définir une règle <file> pour chacune. Dans ce cas, on peut utiliser une expression régulière pour l'attribut *name* en la préfixant par un caractère tilde :

```
<file name="~\.gif$" />
```

Si l'on souhaite directement autoriser tous les fichiers GIF dans le dossier *forum* et sa descendance, on peut utiliser l'attribut *inherit* propre aux règles <file> et <script>, qui vaut *false* par défaut s'il n'est pas précisé :

```
<webapp default="deny" allowindex="true">
   ...
   <directory name="forum">
      <allow><file name="~\.gif$" inherit="true" /></allow>
      <directory name="images">
         ...
      </directory>
   </directory>
</webapp>
```

Enfin, il est évidemment possible d'utilisers des *charsets* dans l'attribut *name* d'une règle <file> ou <script> et de borner sa longueur au moyen des attributs *length*, *minlength* et *maxlength* :

```
<file name="%filename%" maxlength="50" />
<file name="img_%number%.gif" length="11" />
<file name="~^%filename%\.(gif|jpe?g)$" minlength="5" maxlength="30" />
```

4.4.8. Vérification des paramètres de scripts

Si la requête contient des paramètres GET ou POST, une vérification supplémentaire a lieu qui est celle des règles <param>, contenues dans la règle <script>. La vérification des règles n'a bien sûr lieu que si la requête satisfait la règle <script> elle-même.

Attention : par abus de langage, le fait qu'un paramètre soit de type GET ou POST n'a rien à voir avec la méthode HTTP indiquée sur la première ligne de la requête. En effet, le protocole HTTP prévoit qu'une requête GET puisse comporter un contenu (données POST selon *ProxyFilter*) et une requête POST peut tout à fait contenir des paramètres accolés à la fin de l'URL (paramètres GET selon *ProxyFilter*).

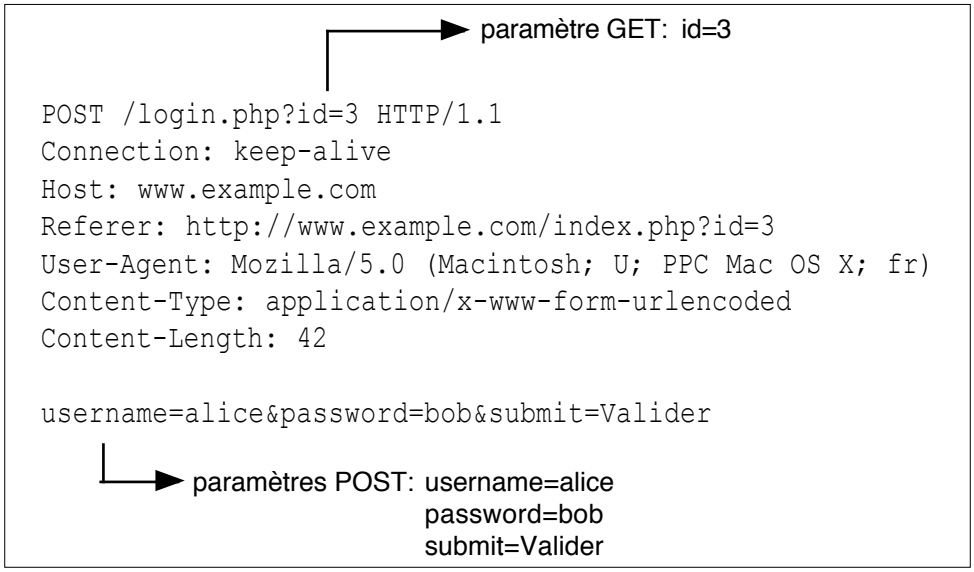


Figure 4.4-4: différence entre les paramètres GET et POST selon ProxyFilter

Pour reconnaître les paramètres GET et POST de façon rapide et fiable, *ProxyFilter* utilise les services de Apache::Request qui a l'avantage de supporter les paramètres multivalués comme ceux qui sont renvoyés par des formulaires comportant des champs à sélection multiple. Pour les données POST **seul l'encodage *application/x-www-form-urlencoded*** supporté. Celui-ci correspond à la méthode d'encodage décrite dans la norme HTML du W3C et qui est aujourd'hui utilisée dans 90% des cas. Pour les formulaires HTML, elle est utilisée par défaut si l'attribut *enctype* de la balise <form> ne définit pas un autre encodage.

Il existe cependant une autre méthode plus moderne pour encoder les données POST qui est *multipart/form-data* décrit par la RFC 1867 et qui est utilisée pour permettre l'envoi de fichiers par POST à partir d'un élément de formulaire <input> de type *file*. Cette méthode n'est pas reconnue par *ProxyFilter*, elle le sera peut-être dans une version future moyennant l'usage de module Perl externe comme la dernière version (instable) de CGI::Request.

Si l'encodage utilisé dans l'entité de la requête (indiqué par l'entête *Content-Type*) n'est pas *application/x-www-form-urlencoded*, alors *ProxyFilter* se contente d'en copier le contenu sans chercher à en reconnaître et filtrer les paramètres. Ce comportement n'est certes pas très sécuritaire, il pourra être amélioré dans une version future du module.

Pour la requête illustrée par la figure 4.4-4, une configuration possible serait :

```
<webapp default="deny" allowindex="true">
    <allow>
        <script name="login.php">
            <param name="id" value="%digit%" method="GET" />
            <param name="username" value="%alphanum%" method="POST" />
            <param name="password" value="%alphanum%" method="POST" />
            <param name="submit" value="Valider" method="POST" />
        </script>
    </allow>
    ...
</webapp>
```

L'élément <script> possède les mêmes attibuts qu'un élément <file>, mais il contient une ou plusieurs règles <param>, dont les attributs obligatoires sont *name*, *value* et *method*, et dont les attributs facultatifs sont *length*, *minlength* et *maxlength*.

Une règle <script> de type *deny* ne devrait pas contenir d'éléments <param> : ils seront ignorés car seul le nom du script (attribut *name*) est pris en compte, comme avec une règle <file>. Par exemple : pour bloquer tous les scripts CGI avec paramètres de l'application :

```
<webapp default="allow" allowindex="true">
    ...
    <deny>
        <script name="~\.cgi$" inherit="true" />
    </deny>
    ...
</webapp>
```

Dans cet exemple, nous avons volontairement choisi un *default policy* valant *allow* au niveau de la racine de l'application, car sinon la règle *deny* n'aurait pas été d'une grande utilité. Comme les règles <script> ne sont évaluées que si la requête contient au moins un paramètre GET ou POST, cette configuration ne bloquera pas les requête à des fichiers .cgi ne comportant aucun paramètre : pour cela, il faut ajouter une règle <file> à la règle <script> :

```
<webapp default="allow" allowindex="true">
    ...
    <deny>
        <file name="~\.cgi$" inherit="true" />
        <script name="~\.cgi$" inherit="true" />
    </deny>
    ...
</webapp>
```

En utilisant l'attribut *optional*, il est possible d'indiquer qu'un paramètre est mandatoire. Dans ce cas, la règle <script> est fausse si le paramètre n'est pas présent (où si sa valeur est inexact, bien sûr). Par défaut, *optional* vaut *false*.

```
<script>
</script>
```

Il ne suffit pas de déclarer une règle <script> dont tous les paramètres sont optionnels pour qu'une requête ne comportant aucune paramètre soit acceptée : il faut pour cela impérativement déclarer une règle <file>.

```
<script name="mail.aspx">
    <param name="to" value="%email%" optional="false" method="POST" />
    <param name="from" value="%email%" optional="false" method="POST" />
    <param name="subject" value="%iso8859%" optional="true" method="POST" />
    <param name="body" value="%iso8859%" optional="false" method="POST" />
</script>
```

L'attribut *method* de <param> est obligatoire : il peut valoir GET (paramètre transmis à la fin de l'URL), POST (paramètre transmis dans l'entité de la requête) ou BOTH (le paramètre peut être transmis par les deux moyens).

Attention : Toutes les règles <file>, <script> et <param> sont évaluées de façon insensible à la casse des caractères, ceci afin d'être compatible avec des serveurs Web tournant sur des environnements non-Unix comme Mac OS 9 et Windows. Il est possible que ce comportement réduise quelque peu le niveau de sécurité avec un serveur Unix, une version future de *ProxyFilter* devrait permettre de choisir le comportement désiré relativement à la casse des caractères.

4.4.9. Envoi de la requête interne

Lorsque la requête a passé tous les tests avec succès, *ProxyFilter* compose un objet requête à transmettre à LWP. Il copie l'URL réécrite, les paramètres et les entêtes filtrés de la requête d'origine.

L'URL réécrite, de part la syntaxe imposée dans le fichier *proxyfilter_mappings* doit avoir une forme absolue, avec protocole, nom d'hôte, et éventuellement numéro de port.

Si l'URL réécrite commence par "`https://`", *ProxyFilter* ouvrira une connexion SSL sur le port 443 du serveur Web (par défaut). Si l'URL réécrite commence par "`http://`", la connexion se fera en HTTP passant en clair sur le port 80 du serveur Web (par défaut).

ProxyFilter ne vérifie pas l'identité du certificat SSL du serveur : l'idée était surtout d'offrir une confidentialité des données dans le cas où le *reverse proxy* et le serveur Web sont séparés par un réseau qui n'est pas de toute confiance. Dans la plupart des cas, on utilisera SSL entre le client et le *reverse proxy* (auquel cas c'est mod_ssl de Apache sera utilisé), et on fera passer les données en clair entre le *reverse proxy* et le serveur Web, afin de ne pas charger davantage le serveur.

Il n'est toutefois pas exclu qu'une version future de *ProxyFilter* permette d'authentifier le serveur au moyen de son certificat, car LWP::UserAgent le permet. Il suffirait pour cela d'indiquer dans le fichier de configuration l'emplacement du répertoire contenant les certificats racines.

Si nécessaire, LWP effectue une résolution DNS du nom d'hôte de l'URL réécrite pour déterminer l'adresse IP du serveur. L'entête *Host* est également renseignée avec le nom d'hôte de l'URL réécrite : toute entête *Host* existante dans la requête du client est remplacée. Si l'on ne dispose pas ou ne souhaite pas utiliser un serveur DNS privé, il possible de résoudre le nom du serveur Web en utilisant une entrée *host* au niveau du proxy (fichier `/etc/hosts` sous Unix et configurer le fichier `resolv.conf` en conséquence).

4.4.10. Traitement de la réponse

Lorsque LWP reçoit la réponse du serveur Web, il appelle un *handler* de *ProxyFilter* qui prend en charge ces données. Le *handler* peut être appelé plusieurs fois dans la même réponse, chaque appel correspondant à quelques Ko de données reçues. Cette technique permet à *ProxyFilter* de lire et filtrer les entêtes avant même d'avoir reçu tout le document, et de commencer d'envoyer les données au client avant d'avoir reçu entièrement le document du serveur. Cette technique optimise drastiquement le temps de réponse du *reverse proxy* pour le client, particulièrement pour les gros fichiers.

Les entêtes HTTP de la réponse sont filtrées selon la même technique employée pour les entêtes de la requête, à la différence près que les règles sont contenues dans l'élément <headers_out> :

```
<config>
   ...
    <headers_out default="deny">
      <allow>
           <header name="connection" value="~^(keep-alive|close)$" />
           <header name="content-type" value="~^(text/html|image/.*)$" />
           <header name="content-length" value="%number%" maxlength="8" />
           <header name="last-modified" value="%longdate%" />
          <header name="transfer-encoding" />
           <header name="set-cookie" value="~^SESS_ID=[a-z0-9]{64}.*$" />
      </allow>
      <filter>
          <header name="server" />
          <header name="x-powered-by" />
      </filter>
    </headers_out>
    ...
</config>
```

L'intérêt de filtrer les entêtes de la réponse est moins évident que de filtrer les entêtes de la requête, car les attaques ne proviennent pas du serveur Web (sauf dans le cas particulier d'un cheval de Troie, mais ce genre d'attaque n'emploiera certainement pas le port 80 et devrait être bloqué au niveau IP).

Pourtant, certains entêtes tels que *Server* et *X-Powered-By* sont intéressantes à filtrer car elles donnent trop d'informations sur le serveur. De plus, le simple fait de pouvoir contrôler le type MIME des documents retournés (*Content-Type*) permet d'éviter que, à la suite d'une erreur de configuration du serveur, le code source de certains scripts soit dévoilé, car ceux-ci retourneront probablement un type *text/plain* au lieu de *text/html*.

La version actuelle de *ProxyFilter* ne permet pas de filtrer le contenu du document retourné (voir les explications à ce sujet dans le chapitre "améliorations futures").

4.5. Exemple de configuration

Nous présentons ici un exemple complet de configuration qui est adapté à l'application d'exemple *securitystore.ch* développée dans le cadre du travail de semestre et disponible sur le CD-Rom et le site Web de *ProxyFilter*. Cette application et la procédure d'installation sont décrites en annexe.

4.5.1. Fichier proxyfilter_webapp.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE webapp SYSTEM "proxyfilter_webapp.dtd">
<!-- racine de l'application Web -->
<webapp default="deny" allowindex="true">

   <allow>
      <file name="index.php" />
      <file name="logout.php" />
      <file name="styles.css" />
      <file name="login.php" />
      <script name="login.php">
         <param name="username" value="%alphanum%" method="post" optional="false" />
         <param name="userpass" value="%textarea%" method="post" optional="false" />
         <param name="submit" value="Valider" method="post" optional="false" />
      </script>
      <script name="details.php">
         <param name="bid" value="~^[0-9]$" method="get" optional="false" />
      </script>
      <script name="comment.php">
         <param name="bid" value="%number%" maxlength="2" method="both" optional="false" />
         <param name="rating" value="~^[0-5]$" method="post" />
         <param name="commentText" value="~^%textarea%$" maxlength="500" method="post" />
         <param name="commentName" value="%alphanum%" maxlength="30" method="post" />
         <param name="submit" value="Envoyer" method="post" />
      </script>
   </allow>

   <deny>
      <file name="~^\..*" inherit="true" />
      <url value="~%xss%" />
      <url value="~root\.exe" />
   </deny>

   <directory name="images" default="deny" allowindex="false">
      <allow>
         <file name="~^[0-9]\.jpg$" />
         <file name="~^.*\.gif$" maxlength="60" />
      </allow>

      <directory name="rank">
         <allow>
            <file name="~^rank[0-5]\.gif$" />
         </allow>
      </directory>
   </directory>

</webapp>
```

Figure 4.5-1: exemple de fichier webapp complet

4.5.2. Fichier proxyfilter_config.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE config SYSTEM "proxyfilter_config.dtd">
<!-- fichier de configuration general de ProxyFilter -->
<config>
   <charsetsfile>/httpd/conf/proxyfilter_charsets</charsetsfile>
   <mappingsfile>/httpd/conf/proxyfilter_mappings</mappingsfile>
   <webappfile>/httpd/conf/proxyfilter_webapp.xml</webappfile>
   <logfile>/var/log/httpd/proxyfilter_log</logfile>
   <loglevel>debug</loglevel>
   <headers_in default="deny">
      <allow>
         <header name="host" value="~^[-_a-zA-Z0-9\.@]*$" maxlength="50" />
         <header name="connection" value="~.*" maxlength="50" />
         <header name="accept" value="~.*" maxlength="500" />
         <header name="referer" value="~.*" maxlength="200" />
         <header name="if-modified-since" value="~.*" maxlength="50" />
         <header name="if-unmodified-since" value="~.*" maxlength="50" />
         <header name="If-None-Match" value="~.*" maxlength="50" />
         <header name="range" value="~.*" maxlength="50" />
         <header name="user-agent" value="~.*" maxlength="200" />
         <header name="cookie" value="~securitystore_session=%alphanum%" maxlength="150" />
         <header name="content-length" value="~.*" maxlength="20" />
         <header name="content-type" value="~.*" maxlength="50" />
         <header name="accept-language" maxlength="700" />
      </allow>
      <deny />
      <filter>
         <header name="pragma" />
         <header name="extension" />
         <header name="ua-cpu" />
         <header name="ua-os" />
      </filter>
   </headers_in>
   <headers_out default="filter">
      <allow>
         <header name="date" />
         <header name="connection" />
         <header name="content-type" />
         <header name="content-length" />
         <header name="etag" />
         <header name="accept-ranges" />
         <header name="client-transfer-encoding" />
         <header name="last-modified" />
         <header name="link" />
         <header name="title" />
         <header name="transfer-encoding" />
         <header name="set-cookie" />
      </allow>
      <filter>
         <header name="server" />
      </filter>
   </headers_out>
</config>
```

Figure 4.5-2: exemple de fichier config complet

4.5.3. Fichier proxyfilter_mappings

```
# Fichier proxyfilter_mappings
#
# Ce fichier contient les règles de réécriture d'URL de ProxyFilter
#
# Chaque règle occupe une ligne et contient 3 colonnes séparées par une tabulation.
# Il existe 3 types de règles :
#
# - les règles "forward" réécrivent l'URL de la requête en se basant sur un préfixe
# - les règles "exact" réécrivent l'URL de la requête en remplaçant exactement
#   l'URL par une autre
# - les règles "reverse" servent à réécrire les entêtes Location lors de redirections
#   externes et les URL absolues dans les documents HTML retournés au client
#   (liens hypertextes, images, etc...)
#
# La première colonne définit l'URL source, la seconde l'URL de destination, la troisième
# le type de règle. Les lignes précédées d'un symbole dièse (#) sont des commentaires,
# elles sont ignorées par le programme.
#
/                           http://www.securitystore.ch/            forward
http://www.securitystore.ch/    /                                   reverse
http://securitystore.ch/        /                                   reverse
/index                      http://www.securitystore.ch/index.php   exact
```

Figure 4.5-3: exemple de fichier mappings complet

4.5.4. Fichier proxyfilter_charsets

```
# Fichier proxyfilter_charsets
#
# Ce fichier contient les équivalences des noms des "charsets" en expressions régulières
# compatibles Perl. La première colonne contient le nom du charset (composé de lettre a-z,
# A-Z, des chiffres 0-9. Le nom du charset dans ce fichier ne doit pas comporter le
# caractère "%" utilisé par la suite dans les expressions simples pour identifier une référence
# de charset. La deuxième colonne contient une expression régulière compatible Perl, sans les ancres
# de début et de fin (^ et $). La référence du charset sera remplacée par l'expression régulière
# telle qu'elle est dans la deuxième colonne, sans modification.
#
# Les lignes précédées d'un symbole dièse (#) sont des commentaires, elles sont ignorées par le pro-
gramme.
#
filename        [a-zA-Z0-9][-_.a-zA-Z0-9]{0,255}
textarea        [-a-zA-Z0-9éàê',:;!?_()\[\]\s]*
xss             <script>.*</script>
digit           [0-9]
number          [0-9]{1,20}
alphanum        [a-zA-Z0-9]*
url             https?|ftp)://([-a-z0-9]{1,50}\.)?[-a-z0-9]{2,50}\.[a-z]{2,4}(/.*)?
```

Figure 4.5-4: exemple de fichier charsets complet

4.6. Améliorations futures

Il n'était pas possible dans le temps imparti pour ce travail de diplôme et en comptant l'apprentissage du langage Perl et des diverses technologies annexes, d'obtenir un produit fini. Tout au plus, *ProxyFilter* dans son état à fin décembre 2003 peut être qualifié de version *alpha*. Il subsiste des bogues et certaines fonctionnalités du cahier des charges n'ont pas pu être implémentées. Il y a également certaines fonctionnalités qui ne sont pas mentionnées dans le cahier des charges et qu'il serait intéressant d'offrir. Dans ce chapitre, nous proposons quelques améliorations futures de *ProxyFilter*.

4.6.1. Optimisation des performances

Dans la mesure où j'ai écrit ce module en étant néophite en Perl au départ, la syntaxe n'est pas toujours très propre. En particulier, il faudrait davantage décomposer le programme principal en sous-programmes, éliminer les quelques variables globales qui subsistent, et optimise le code afin d'améliorer les performances, de telle sorte que *ProxyFilter* lui-même ne soit pas plus vulnérable au attaques par déni de service que le serveur qu'il protège !

L'amélioration la plus bienvenue serait d'éviter de lire les fichiers de configuration à **chaque requête** comme actuellement, mais de les lire **au démarrage du serveur uniquement**. Cela signifierait qu'il faudrait redémarrer Apache pour que les modifications soient prises en compte.

Le problème consiste à mémoriser la configuration dans des variables globales, alors que chaque processus enfant de Apache dispose, sous Unix du moins, de son propre espace mémoire. Cela serait possible en utilisant la module IPC::Shareable qui offre précisément des variables globales aux modules Perl pour Apache avec gestion des accès concurrent en terme d'exclusion mutuelle, et le livre *Writing Apache Modules With Perl And C* propose même un exemple d'utilisation de ce module pour réaliser un compteur global en mémoire de chargement d'une page (sans utiliser de fichier sur le disque ou de base de donnée).

4.6.2. Mode d'autoapprentissage

Il faut reconnaître que de configurer *ProxyFilter* pour une application Web complexe comprenant des dizaines de pages dynamique est un long travail. L'idée est donc, comme le fait AppShield, de proposer un mode dans lequel le *reverse proxy* laisse tout passer (*PassThru*) mais observe le trafic et génère dynamiquement des règles. Après cette période d'apprentissage (durant laquelle on assure qu'il n'y ait pas de tentative d'attaque), on peut basculer le module en mode filtrant, après avoir éventuellement affiné manuellement certaines règles.

À priori, cette fonction revient à proposer une version XML du log qui écrirait les règles à la place de l'administrateur, mais tout n'est pas si évident. En effet, le programme devrait faire beaucoup d'hypothèse : comment, sur la base de la valeur d'une entête, déterminer quelle est sa plage de valeurs valide ? Il faudrait un historique important et pouvoir extrapoler des expressions régulières et bornes de longueur, ce qui n'est pas évident.

4.6.3. Filtrage du contenu de la réponse

En l'état actuel, *ProxyFilter* vérifiée et filtre les entêtes de la réponse, mais pas le contenu du document retourné qu'il se contente de recopier pour l'envoyer au client. Ce filtrage n'aurait de sens que pour les documents textes et HTML, car que peut-on filtrer dans des fichiers binaires comme le GIF ou le JPEG ?

Reconnaître et filtrer certains mot-clés dans le document retourné serait assez simple en utilisant des expressions régulières de substitution. Ainsi, si l'on sait que l'application n'emploie pas d'applets, on pourrait filtrer les balises <applet> et si l'application n'utilise pas de plug-in ou d'ActiveX les balises <object> correspondantes, voir les balises <script> dans les documents qui ne sont pas censés en comporter pour éviter les attaques XSS par contamination directe de la base de données.

Une autre idée, proposée par M. Maret, est de réécrire tous les liens hypertextes qui contiennent le nom du serveur cible. Exemple :

proxyfilter_mappings

```
/site/                  http://server.local/   forward
http://server.local/    /site/                 reverse
```

document renvoyé par le serveur

```
<html>
<head> ... </head>
<body>
   ...
   <img src="http://server.local/image3.jpg">
   <form action="/cgi-bin/mail.cgi" method="post">
      ...
   </form>
   ...
</body>
</html>
```

document renvoyé au client

```
<html>
<head> ... </head>
<body>
   ...
   <img src="/site/image3.jpg">
   <form action="/site/cgi-bin/mail.cgi" method="post">
      ...
   </form>
   ...
</body>
</html>
```

Figure 4.6.1 : exemple de réécriture d'URL dans le document retourné

On remarque que cette fonctionnalité est utile non seulement lorsque la page utilise des adresses absolues en `http://` contenant le nom local du serveur, mais aussi lorsque les adresses sont relatives à la racine de l'application Web et que celle-ci n'est pas *mappée* à la racine dans le contexte d'URL publique du *reverse proxy*. Il s'agit donc d'une fonctionnalité urgente à proposer, mais qui pose pas mal de difficultés.

Le langage HTML n'est pas très rigoureux : la casse des caractères dans les balises et attributs est laissée bon vouloir de l'auteur de la page, les guillemets encadrant les valeurs des attributs ne sont pas obligatoires. Sur ce point là, XML et XHTML font nettement mieux. Par conséquent, parcourir une page HTML de façon rapide et fiable nécessite des modules spécialisés, surtout s'il s'agit d'aller modifier le contenu de la page.

Pour le langage Perl, il existe le module HTML::Parser qu'il était prévu d'étudier et d'utiliser pour la version actuelle de *ProxyFilter*, mais le temps a manqué. Même en supposant que l'on parvienne à réécrire les liens dans la page, cela se fera incontestablement au prix d'une baisse de performances : en l'état actuel, on utilise les appels multiples du *handler* par LWP pour commencer de retourner le document au client avant même que le proxy ne l'aie entièrement reçu du serveur (lorsque les entêtes ont été vérifiées). Cette technique est nettement plus efficace, surtout pour des documents de taille importante ou si le lien entre le proxy et le serveur n'est pas très rapide, mais elle ne permet pas de modifier le contenu d'un document HTML, car les *parser* nécessitent le document en entier pour travailler. Même la recherche d'un mot-clé est critique en recevant le

document par *chunks* de la sorte, car un mot-clé présent dans le document pourrait être coupé en deux parties à la réception.

Il est indéniable que, si *ProxyFilter* disposait de cette capacité de réécrire les URL présentes dans le document, il bénéficierait d'une avance sur ses concurrents. Mais ne vaut-il pas mieux sensibiliser les programmeurs Web afin qu'ils écrivents des applications *proxy-compliants* ?

4.6.4. Interface graphique de configuration

Même si la syntaxe de configuration XML est relativement claire, il est concevable qu'elle rebute certains utilisateurs de système non-Unix qui ne sont pas habitués à configurer de la sorte un logiciel. Il serait donc souhaitable, à terme, de disposer d'une interface Web permettant de créer de nouvelles règles et modifier les règles existantes. Cette interface prendrait sans doute la forme d'un script CGI écrit en Perl et tournant sur un port sécurisé par SSL du serveur *proxy*, et qui éditerait par-dessous les fichiers de configuration (en utilisant XML::Parser, bien sûr). On pourrait également proposer de redémarrer le serveur Apache à distance sans devoir établir une connexion SSH, mais il existe déjà des interfaces comme *Webmin* pour cela.

4.6.5. Firewall *stateful*

En l'état actuel, *ProxyFilter* est un firewall que l'on pourrait qualifier de *stateless*, car chaque requête est filtrée individuellement. AppShield est plutôt *stateful* : il emploie un cookie pour tracer les actions de l'utilisateur tout au long de sa session, comme permet de le faire mod_usertrack sous Apache. Cela permet d'améliorer le niveau de sécurité, et par exemple de ne pas permettre l'accès direct à certaines pages internes du site si l'utilisateur n'est pas auparavant passé par un portail d'entrée, ou de mémoriser les liens, cookies et champs de formulaire de chaque page retournée pour les vérifier à la requête suivante de l'utilisateur. Ce sont ces fonctionnalités avancées qui font la différence entre un produit libre comme *ProxyFilter* et un produit commercial à plusieurs milliers de francs comme AppShield.

4.6.6. Mise à jour dynamique de la Black list

Des scanners de vulnérabilités comme Nessus sont constamment mis à jour avec les dernières vulnérabilités connues. Il serait intéressant de pouvoir importer le fichier décrivant les requêtes permettant d'exploiter ces vulnérabilités pour générer dynamiquement des règles *deny*, sans que l'utilisateur aie à les saisir de lui-même.

4.6.7. Plugin de compatibilité pour le serveur Web

Lorsqu'on utilise un *reverse proxy* pour protéger une application Web, toutes les requêtes semblent venir d'une seule et même adresse : celle du proxy. Ainsi, les programmes d'analyse des historiques du serveur Web comme Analog ou AWStats qui utilise l'adresse IP du client pour dresser des statistiques de provenance ne peuvent plus travailler correctement.

Pour que l'adresse IP du client distant soit tout de même propagée au serveur, la solution employée par Sanctum AppShield est d'ajouter au niveau du proxy à chaque requête une entête "CLIENT_IP". Cette information peut ensuite être écrite dans l'historique du serveur Web, moyennant un plugin de compatibilité à installer sur le serveur (livré avec AppShield pour Apache et IIS). L'idée serait de développer un plugin similaire pour *ProxyFilter*, qui prendrait la forme d'un module Perl pour Apache gérant la phase d'historique (*LogHandler*) de la requête.

5. Conclusions

La sécurité sur Internet peut être vue comme plusieurs couches qui reposent l'une sur l'autre :

- **La sécurité au niveau réseau** est assurée par les firewalls qui sont aujourd'hui des produits bien maitrisés.
- **La protection contre les vers et virus** passe par l'installation de logiciels anti-virus relativement efficaces s'ils sont mis à jour régulièrement et par la sensibilisation des utilisateurs aux problèmes de sécurité.
- **Les vulnérabilités des systèmes d'exploitation et logiciels serveurs** peuvent être contrées par l'application immédiate des *patchs* de sécurité qui sont généralement mis à disposition peu après la découverte d'une vulnérabilité, Internet et le monde *open source* aidant à créer une communauté de personnes compétentes en matière de sécurité (*white hats*).
- **La sécurité des applications et services Web** est plus critique, car elle concerne des vulnérabilités dans du code "développé maison" dans des langages de haut niveau tels que Java, PHP, Perl, ASP et qui n'est pas largement diffusé. Contre ces vulnérabilités, seuls les firewalls applicatifs (proxys), un domaine relativement nouveau, sont efficaces.

On constate que, des deux stratégies utilisées par les proxys pour filtrer le trafic, aucune n'est vraiment la panacée. Le filtrage exclusif (*Black list*) nécessite une bonne connaissance des dernières vulnérabilités découvertes pour être efficace. Le filtrage inclusif (*White list*) offre un plus haut niveau de sécurité mais nécessite une configuration lourde qui doit être actualisée lorsque l'application évolue. La meilleure solution est vraisemblablement un compromis entre ces extrêmes, comme tente de le faire AppShield avec sa génération dynamique de règles en fonction du document retourné et ProxyFilter en permettant de redéfinir le comportement par défaut à différents niveaux de l'application.

ProxyFilter, le firewall HTTP développé dans le cadre de ce travail, n'est qu'une ébauche qui ne peut bien sûr pas se mesurer avec la solution commerciale et établie depuis 4 ans qu'est AppShield, mais il s'agit d'un logiciel *open source* dont la publication sur Internet devrait inciter d'autres développeurs à s'intéresser au problème et améliorer *ProxyFilter* ou à proposer leur propre solution.

Ce travail de diplôme a été très enrichissant, car il m'a permis d'étudier de manière détaillée le protocole HTTP, de mieux comprendre la façon dont Apache traite une requête et d'apprendre le langage Perl. L'étude et le déploiement d'un produit de première classe en matière de sécurité comme AppShield était en outre très formatrice.

Je tiens à remercier MM. Sylvain Maret et Stefano Ventura pour leurs conseils et pour m'avoir suivi durant ce travail, M. Gérald Litzistorf pour avoir eu la patience de me lire et d'évaluer mon travail, M. Marc Lyanage de futureLAB pour m'avoir dépanné au sujet de Perl et XML, ainsi que les auteurs des merveilleux ouvrages qui m'ont servi de référence.

Yverdon, le 18 décembre 2003 Sylvain Tissot

6. Références

6.1. Ouvrages papier

- *Learning Perl, 3rd Edition*, O'Reilly, Randal L. Schwartz & Tom Phoenix
- *Writing Apache Modules with Perl and C*, O'Reilly, Licoln Stein & Doug MacEachern
  http://www.modperl.com
- *Professional Apache Security,* Wrox Press Ltd, Tony Mobility, Kapil Sharma
- *Sécurité des applications,* Travail de diplôme 2002, Michael Zanetta, EIG

6.2. Adresses Internet

- *AppShield edges InterDo in battle of Port 80 filters*, NetworkWorldFusion
  http://www.nwfusion.com/reviews/2003/0818rev2.html
- *Review: AppShield 4.0*, eWeek
  http://www.eweek.com/article2/0,4149,1110427,00.asp
- *Comprehensive Perl Archive Network*
  http://www.cpan.org
- *libwww-perl*
  http://lwp.linpro.no/lwp/
- *mod_perl*
  http://perl.apache.org

7. Lexique des termes et abréviations

Apache	Serveur Web *open source* issu des serveurs du NCSA et CERN, aujourd'hui projet autonome regroupant des centaines de développeurs. Rapide, stable, gratuit, extensible, multiplateforme, tels sont les termes qui peuvent définir le serveur Apache
AppScan	Scanner de vulnérabilités de la société Sanctum, spécialisé dans les applications Web
AppShield	Firewall applicatif HTTP développé par Sanctum depuis 1999, actuellement en version 4.0
Black List	Stratégie de filtrage consistant à bloquer les attaques reconnues et à laisser passer tout le reste
Cookie	Informations mémorisées par le serveur Web sur la machine du client et qui pourront être relues ultérieurement. On utilise fréquemment un cookie pour mémoriser une clé de session
démon	Logiciel sans interface graphique tournant en tâche de fond de manière permanente. Tous les serveurs (Apache, IIS, Postfix) sont des démons
DOM	*Document Object Model*, technique pour parcourir un document XML retournant un arbre
Expat	*Parser XML* en mode flux écrit en langage C, très rapide et accessible au monde Perl au travers du module XML::Parser
HTTP	*Hypertext Transfer Protocol*, protocole utilisé sur le World Wide Web, fonctionnant généralement sur le port TCP 80
HTTPS	Protocole HTTP sécurisé par SSL, fonctionnant généralement sur le port TCP 443
LWP	Bibliothèque permettant l'accès au World Wide Web et au protocole HTTP depuis un programme Perl
parser	Programme effectuant le parcours d'un document texte, action de parcourir un document
Perl	À l'origine abréviation pour Practical Extraction And Report Language, langage de script inventé par Larry Wall au milieu des années 1980 et qui s'est sans cesse enrichi depuis. Simple à utiliser, difficile à apprendre, très peu typé
Proxy Web	Passerelle au niveau applicatif servant d'intermédiaire pour atteindre des serveurs Web externes depuis un réseau local, dans le but de réaliser un cache performance, filtrer le trafic, etc...
Reverse proxy	Passerelle au niveau applicatif située devant un serveur Web dans le but de réaliser du load balancing, protéger l'application, réaliser un cache performance, etc...
SAX	*Simple API for XML Parsing*, technique en mode flux pour parcourir un document XML
SSL	*Secure Socket Layer*, technique de cryptographie et d'authentification PKI utilisée pour assurer le confidentialité d'une communication sur TCP, généralement utilisé pour le Web (HTTPS)
URL	Universal Ressource Locator, adresse permettant d'identifier une ressource sur le Web
White List	Stratégie de filtrage consistant à laisser passer ce qui est connu et à bloquer tout le reste
XML	*Extensible Markup Language*, langage à balises dérivé de SGML permettant de décrire une structure de donnée
XSS	*Cross Site Scripting*, attaque de l'utilisateur d'une application Web vulnérable visant essentiellement à s'emparer de sa clé de session stockée sous la forme d'un cookie, en injectant du code mobile (Javascript, VBScript) dans la page Web retournée au client