ProxyFilter

Manuel d'utilisation

Auteur : Sylvain Tissot
Version : 0.1
Date : 3 janvier 2004

http://proxyfilter.sourceforge.net

Table des matières

ProxyFilter : manuel d'utilisation

1. Introduction

1.1. Qu'est-ce que ProxyFilter ?

ProxyFilter est un firewall applicatif HTTP basé sur Apache et Perl, destiné protéger les applications Web d'attaques courantes de niveau applicatif telles que la manipulation de paramètres, l'injection de commandes et données SQL, les attaques par *cross site scripting* (XSS) etc...

Pour plus d'informations sur ProxyFilter et la sécurité des applications Web en général, visiter le site à l'adresse :

```
http://proxyfilter.sourceforge.net
```

1.2. Contenu de ce document

Ce document admet que ProxyFilter et tous les composants requis ont été installés conformément aux instructions présentes dans le manuel d'installation. Nous nous contenterons de décrire la syntaxe des fichiers de configuration, et quels sont les tests et traitement que ProxyFilter effectue à chaque requête conformément à cette syntaxe de configuration.

2. Algorithme général

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.

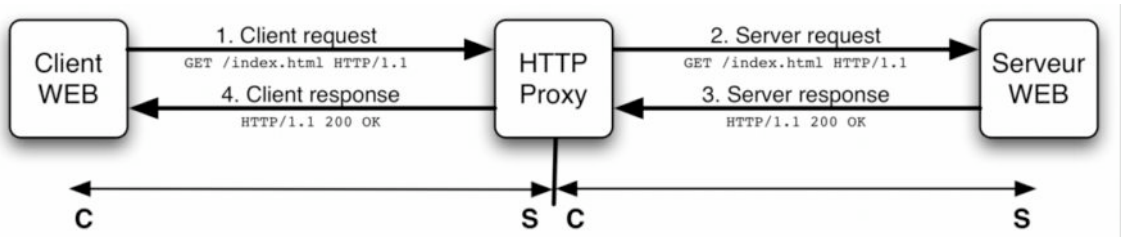

Figure 1: fonctionnement en 4 étapes de ProxyFilter

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 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 2: algorithme général 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.

3. 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 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 er-

reur 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 `httpd.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. Filtrage de la méthode HTTP

Lorsqu'une requête arrive, ProxyFilter commence par vérifier que la méthode HTTP (GET, POST ou HEAD) 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.

5. 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*.

6. 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.

7. 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 `proxyfilter_webapp.xml` (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 URLs spéciales pour lesquelles il n'existe pas même de bloc `<directory>` valide, 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 dépassement de tampon.

```
<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 prendre 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 la comparaison ne s'effectue pas sur le *scheme* et le nom d'hôte, uniquement sur le *path* et les éventuels paramètres transmis par GET.

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* ou *deny*, la fonction chargée d'évaluer les règles `<url>` retourne un statut neutre qui signifie que les règles `<file>` et `<script>` doivent être évaluées pour décider si la requête est acceptée ou non.

8. Vérification du répertoire de la requête

Lorsque le test de règles URL retourne un statut 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.

9. 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'utiliser 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" />
```

10. 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).

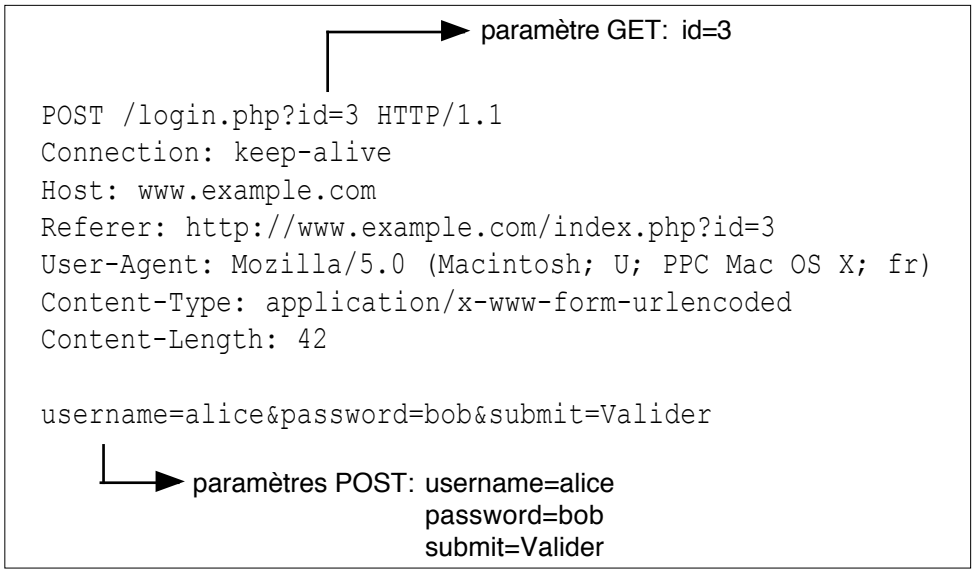

Figure 4: différence entre les paramètres GET et POST selon ProxyFilter

Pour reconnaître les paramètres GET et POST de façon rapide et fiable, ProxyFilter fait appel au module 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. Elle est utilisée par défaut pour les formulaires HTML dans lesquels l'attribut `enctype` de la balise `<form>` n'est pas renseigné.

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 ou de Libapreq.

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, 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 attributs qu'un élément `<file>` à la différence qu'il sert de conteneur à une ou plusieurs règles `<param>`., dont les attributs mandatoires 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 `<deny>`. Par exemple, pour bloquer pour toute l'application les scripts CGI avec au moins un paramètre :

```
<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 requêtes 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 inexacte, bien sûr). Par défaut, `optional` vaut `false` ce qui signifie que les paramètres sont facultatifs.

```
<allow>
    <script name ="show.cgi" inherit="false">
       <param name="id" value="~^[0-9]+$" method="both" optional="false" />
       <param name="sort" value="~^(asc|desc)$" method="both" optional="true" />
    </script>
</allow>
```

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.

11. 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).

12. 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, mais cela est prévu dans une évolution future du logiciel, par l'utilisation de modules tels que HTML::Parser.

13. Exemple de configuration

Nous présentons ici un exemple complet de configuration qui est adapté à l'application d'exemple *securitystore.ch* développée en PHP/MySQL dans le cadre du travail de semestre qui a précédé le développement de ProxyFilter, et qui présente volontairement bon nombre de vulnérabilités dans un but démonstratif. Les sources de cette application peuvent être téléchargées sur le site de ProxyFilter.

13.1. 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 5 : exemple de fichier webapp complet

13.2. 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 6 : exemple de fichier config complet

13.3. 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 7 : exemple de fichier mappings complet

13.4. 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 des lettres
# a-z, A-Z, des chiffres 0-9. Le nom du charset dans ce fichier ne doit pas comporter de
# 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 se présente dans la deuxième
# colonne, sans modification.
#
# Les lignes précédées d'un symbole dièse (#) sont des commentaires, elles sont
# ignorées par le programme.
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 8 : exemple de fichier charsets complet