Génie Logiciel Appliqué

Gestion des fichiers

Yves Duchesne
yves@acceis.fr

- Par nature, lorsqu'un client utilise une **application** web **embarquée** sur un serveur HTTP, il est **cloisonné** en son sein
- Il ne peut pas accéder au **système** qui héberge le serveur de façon **directe**
- Il y a une **rupture** de flux **applicatif** : l'utilisateur s'adresse au **serveur** web, qui lui s'adresse au **système** pour le traitement des requêtes
- Ce cloisonnement est un **mécanisme** essentiel de la **sécurité** du système

- Il y a **plusieurs** types **d'interactions** possibles entre une **application** web et le **système de fichier** sous-jacent
  - Il peut y avoir des accès en **écriture**
    - *Upload* de fichiers
    - Journalisation
    - Etc.
  - Il peut y avoir des accès en **lecture**
    - Ressources
    - Inclusion de fichiers
    - Etc.

- Ces **interactions** sont de nature à **rompre** le **cloisonnement** de l'utilisateur dans le **serveur**
- En effet, si les possibilités **d'accès** au système de fichiers sous-jacent sont trop peu **contrôlées**, il est possible à un attaquant de les **utiliser** pour provoquer des accès **illégitimes**
  - En **écriture**
  - En **lecture**
- Être capable **d'accéder** aux ressources du système de fichier peut **menacer** la **sécurité** du système entier

- Les **accès** au **système de fichiers** souffrent d'un **problème connu** depuis longtemps et qui a déjà donné lieu à des **vulnérabilités**
- Il s'agit de la technique du « ***path traversal*** »
- Les **chemins** de fichier sont représentés sous la forme de **chaînes de caractères** dans lesquels il est possible **d'injecter** des portions de chemin
- En particulier, l'utilisation de **caractères spéciaux** permet de se « promener » dans **l'arborescence** du système de fichiers

- Les **caractères spéciaux disponibles** peuvent **dépendre** du **système** utilisé par le serveur, mais **certains** sont **valides** dans tous
  - **.** désigne le répertoire **courant**
  - **..** désigne le répertoire **parent**
- Sous **UNIX**
  - **/** sépare les **répertoires** dans un **chemin**
  - **~** désigne le **répertoire personnel** (*home*)
- Sous **Windows**
  - **** sépare les **répertoires** dans un **chemin**

- La **façon** dont l'application utilise ce chemin pour accéder à la **ressource** peut même permettre d'utiliser des structures plus **complexes**
- En particulier, **certaines API** supportent les **URL** et sont capables **d'interpréter** ces chemins comme ainsi
- Il est donc **possible** d'adresser des ressources **distantes**, par le biais de protocoles **réseaux**
  - http://
  - ftp://
  - Etc.

- **L'erreur** parfois commise par les **développeurs** est de ne **pas penser** à l'utilisation de ces éléments lors de la **construction** des chemins de fichier
- Si l'utilisateur peut **contrôler** une partie d'un **chemin de fichiers**, il peut alors **injecter** ces éléments pour aller chercher n'importe quel **fichier** sur le système
- C'est cette attaque qui est appelée « ***path traversal*** », car elle consiste à « **traverser** » des **répertoires** pour aller chercher une **ressource**

- **Historiquement**, cette vulnérabilité affectait même les **serveurs HTTP**
- Il était possible d'aller **chercher** des fichiers **sensibles** sur le disque **directement** en utilisant les **URL** que l'on passait au serveur
- On utilisait alors des URL de ce genre
  - `http://site/../../../../../etc/passwd`
- Aujourd'hui ce n'est (heureusement) plus le cas, mais cette vulnérabilité est **récurrente** sur les **contenus applicatifs**

- Lorsqu'un **navigateur** souhaite accéder à une **ressource**, il va envoyer une **requête** au serveur pour l'obtenir
  - Requête **GET**
  - Requête **POST**
- La ressource **demandée** peut être de plusieurs **natures**
  - Il peut s'agir d'une ***servlet***, dont le contenu sera **traité** par un conteneur web qui appellera un **module applicatif** spécifique
  - Il peut s'agir d'un **fichier**

- Dans le cas où le client souhaite **obtenir** un **fichier**, la requête qu'il enverra au serveur pourra être **traitée** de **plusieurs façons**
- Il peut s'agir d'un fichier **purement statique**
  - Dans ce cas, le contenu du fichier est **envoyé** au client **tel qu'il figure sur le système de fichiers**
- Il peut s'agir d'un fichier **dynamique**
  - Dans ce cas, la requête du client provoquera **l'exécution** de portions de **code source** sur le serveur pour générer les **contenus dynamiques**

- Pour traiter des fichiers de façon **dynamiques**, le serveur web charge des **modules**, permettant de **supporter** des langages **supplémentaires**
  - PHP
  - ASP
  - Etc.
- Lorsqu'un utilisateur accède à un **fichier**, le serveur web **détermine** s'il doit le traiter **dynamiquement** en se basant sur son **extension**

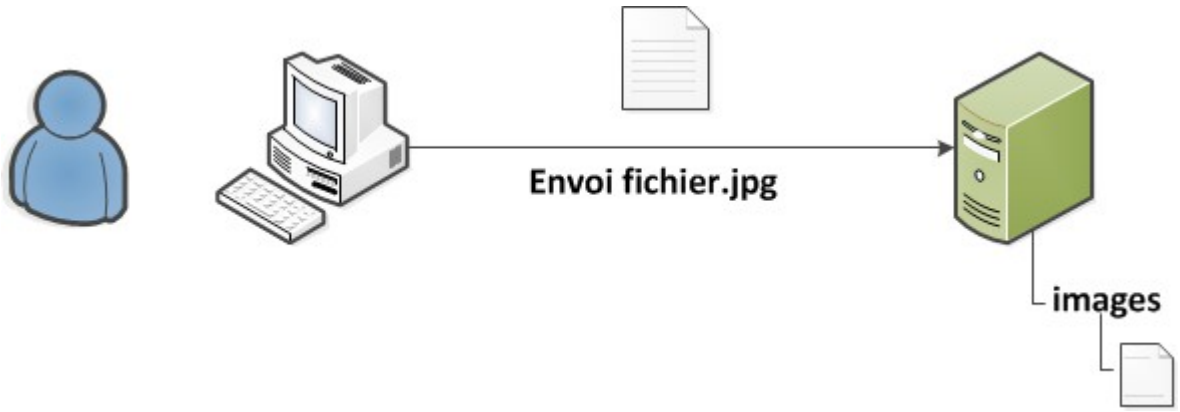
- Les problèmes de **sécurité** liés aux accès en **écriture** sur un **système de fichiers** depuis une application **web** concernent principalement les fonctionnalités d'**envoi de fichier** (*upload*)
- C'est la possibilité de **fournir** un fichier au **serveur** dans le cadre d'une **requête**
  - Champ **`<input>`** de type "file"
- C'est le cas dans les fonctionnalités d'ajout d'image de profils par exemple

Gestion des fichiers

- Si le fichier est **stocké** sur le **disque**, il peut être possible à un client d'y **accéder** par l'intermédiaire de son **navigateur**
- Dans ce cas, le serveur web va **déterminer** comment il doit **traiter** cette requête en se basant sur **l'extension** de ce fichier
- Si le fichier possède une **extension** correspondant à un **module** chargé sur le serveur, il sera **interprété** (exécuté)
  - C'est une **grave menace**

- La perspective qu'un utilisateur puisse **placer** un **fichier** sur le **serveur**, puis **l'exécuter** est **très inquiétante** et constitue une **menace très sérieuse**
- Si un utilisateur peut **exécuter** des portions de **langage** qu'il a lui-même **créées**, il est en effet en mesure de **provoquer** la **compromission** totale du **serveur**
- Il pourra **rompre** le **cloisonnement** du serveur et la **rupture de flux** applicatif en s'adressant directement au **système** sous-jacent

- Un utilisateur **envoie** un fichier et le serveur le **stocke** sur le **système de fichiers**



- Si l'utilisateur fait ensuite une **requête** pour **obtenir** ce fichier. Le serveur va donc le **chercher** sur son **système de fichiers**

- Si le fichier est un fichier **exécutable** (par exemple une page **JSP**), cette requête par l'utilisateur provoquera une **exécution**

- Il est également possible d'utiliser ce mécanisme pour mener des **attaques** un peu **différentes**
- Il est en effet possible **d'abuser** de ces fonctionnalités pour que le fichier envoyé **écrase** et **remplace** un fichier déjà présent sur le disque
- Dans ce cas on va cibler des fichiers **sensibles**
  - Fichiers **`.htpasswd/.htaccess`**
  - Fichiers de **configuration** du **serveur**
  - Fichiers **exécutables** du **service**
  - Fichiers de **configuration** du **système**

- Un **autre danger** inhérent à ces **fonctionnalités** d'envoi de fichiers est le **déni de service**
- La technique est **simple** : En utilisant une fonctionnalité d'envoi de fichiers, un utilisateur fournit un fichier de **très grande** taille au serveur
- Le serveur le **copie** sur le **disque** au fur et à mesure
- Le processus **continue** jusqu'à ce que tout **l'espace** disque soit **consommé**
- Le service ne peut plus **fonctionner**

- Il est donc **nécessaire** de faire preuve de **beaucoup** de **prudence** lorsqu'on implémente ces **fonctionnalités**
- Il est **possible** de mettre en place plusieurs **mesures** pour se **protéger** de cette menace, en particulier
  - Utiliser **préférentiellement** un stockage en **base de données**
  - **Renommer** les fichiers envoyés par l'utilisateur
  - **Filtrer** les fichiers par extension
  - **Limiter** la **taille** des fichiers acceptés

- Les accès en **écriture** sont donc **très dangereux** et doivent faire l'objet d'une attention très **particulière**
- Les accès en **lecture** sont également de nature à permettre à un attaquant de **compromettre** le serveur
- Les méthodes utilisées sont **différentes**, car dans l'absolu, il n'est pas possible de **modifier** le système de fichiers du **serveur**

- Il existe **plusieurs** cas de figure d'accès en lecture à un système de fichiers
- **Premier** cas de figure : le serveur va **lire** une ressource spécifiée par l'utilisateur et lui en **renvoie** le contenu
- **Deuxième** cas de figure : le serveur va utiliser un paramètre fourni par l'utilisateur pour **l'inclure** à la page de réponse
  - Mécanisme de « **page générique** » qu'il n'est pas rare de rencontrer

- Dans le cas de figure où le serveur nous renvoie le contenu **brut** d'un **fichier** sans l'interpréter, il faut s'appuyer sur cette fonctionnalité pour **lire** des fichiers **intéressants**
- Il y a plusieurs types de fichiers à cibler
  - Fichiers **système** (**`/etc/shadow`** par exemple)
  - Fichiers de **configuration**, du système ou du service HTTP
  - Fichier de **journalisation** (logs) qui contiennent toujours des informations intéressantes
  - Fichiers de **code source**

- **L'exploitation** de ce genre de **mécanisme** mal implémenté consiste donc à **rassembler** le maximum **d'informations** sur le **système** et les **services**
- On va cherche à **récolter** des éléments de plusieurs **natures**
  - **Identifiants** de connexion (mots de passe, pas exemple)
  - **Topologie** du **système de fichiers** (emplacement du service HTTP, par exemple)
  - **Topologie** du **réseau** (plan d'adressage, par exemple)

- Une autre **possibilité** pour utiliser des **fichiers locaux** et d'utiliser **l'inclusion** dans une **page**
- Cas de figure **classique** : une **page générique** contenant les éléments **statiques** de toutes les pages de l'application, qui inclue d'autres **fichiers** pour modifier le contenu **principal**

```
<html>
    <head></head>
    <body>
        <div>HEADER</div>
        <div>
            <jsp:include file="jsp/accueil.jsp" />
        </div>
        <div>FOOTER</div>
    </body>
</html>
```

- Il arrive **régulièrement** que cette **inclusion** soit **dynamique** afin d'avoir une seule page **générique** dont le contenu soit chargé à l'aide de **paramètres** fournis par **l'utilisateur**
- Le cas **typique** étant la présence d'un **paramètre** « `page` » par exemple, au sein des URL de l'application pour spécifier le contenu
- Il arrive **régulièrement** que, pour simplifier l'implémentation, la **valeur** de ces **paramètres** soit le nom du **fichier** à inclure

```
<html>
    <head></head>
    <body>
        <div>HEADER</div>
        <div>
            <!-- On lit la valeur du paramètre « page » -->
            <%! String page = request.getParameter("page"); %>

            <!-- On utilise ce paramètre pour inclure une page -->
            <jsp:include file="jsp/<%= page %>jsp" />
        </div>
        <div>FOOTER</div>
    </body>
</html>
```

- La navigation se fera par le biais d'URL du type
  - **http://site/?page=accueil**
  - **http://site/?page=login**
  - Etc.

- Dans ce cas, la **situation** et les **possibilités** sont **différentes**
- Le **contenu** du fichier ne va pas être **inclus** tel quel avant d'être envoyé au serveur
- Il va préalablement être **interprété** par le moteur du serveur
- Cette **interprétation** va donc exposer le serveur à un risque **plus grand** qu'une simple fuite de données
  - Il peut y avoir **exécution de code**

- Dans ce cas, il est intéressant pour un **attaquant d'obtenir** cette exécution de code
- Pour y parvenir il faut réussir **deux choses**
  - 1. Provoquer **l'inscription** d'un **code malveillant** dans un fichier du serveur
  - 2. Utiliser la vulnérabilité **d'inclusion** de fichier pour provoquer son **exécution**
- Ces deux éléments peuvent être **difficiles** à réussir **conjointement**, cette exploitation n'est **pas triviale**

- La **partie** la plus **difficile** est de provoquer **l'inscription** du **code malveillant** dans un fichier du serveur
- On se trouve dans le cas de figure où le serveur présente une **vulnérabilité** permettant l'accès en **lecture** à des fichiers du système, mais **aucune** permettant l'accès en **écriture**
- Il n'est donc **pas possible** d'utiliser une **attaque** pour y parvenir, il faut utiliser un **mécanisme légitime** du serveur

- Le **mécanisme** tout **désigné** pour obtenir un tel effet est le **mécanisme** de **journalisation** du serveur HTTP
- En effet, le serveur HTTP va **inscrire** les détails de chaque **requête** dans un fichier de **journalisation**
  - `/var/log/apache2/access.log` par exemple
- C'est le comportement par **défaut** du serveur et cette **fonctionnalité** sera presque **toujours présente**

- Voilà par exemple le format d'une **requête HTTP** telle qu'elle est **enregistrée** dans le fichier **`access.log`** de **Apache**

192.168.0.20 - - [18/Oct/2015:19:00:58 +0200] "**GET /js/acceis.js HTTP/1.1**" 200 1857 "**http://192.168.0.20/**" "**Mozilla/5.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0 Iceweasel/38.2.1**"

- Certaines **parties** de ces **enregistrements** sont **contrôlées** par **l'utilisateur**
  - C'est un **comportement normal** pour un journal des **accès utilisateurs**
- On peut donc placer du **code malveillant** dans ces **champs**

- Il y a **manifestement** plusieurs champs possible pour une telle **injection**
  - L'URL
  - Le champ « **Referer** » de l'en-tête HTTP
  - Le champ «**User-Agent**» de l'en-tête HTTP
- Ces **champs** sont **faciles** à manipuler par le client qui effectue la requête
- Parmi eux, le plus **trivial** à **modifier** est l'URL, qu'il suffit de **modifier** dans la **barre d'adresse** d'un **navigateur**, mais pose des **problèmes d'encodage** des caractères

- Un champ **utile** dans ce genre de **situation** est le champ « **User-Agent** »
  - Il ne porte aucune **sémantique** et est souvent **neutre** pour le **traitement** des requêtes
- Exemple de **requête** avec un champ « User-Agent » **modifié** :

```
$ GET -H "User-Agent: <?php phpinfo(); >?" "http://192.168.0.20/"
```

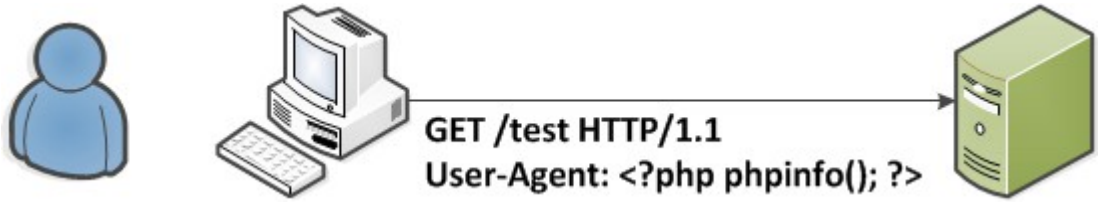
- Cette **requête** a provoqué **l'inscription** de la ligne suivante dans le **journal** d'Apache

```
192.168.0.20 - - [18/Oct/2015:19:16:06 +0200] "GET / HTTP/1.1" 200
6852 "-" "<?php phpinfo(); >?"
```

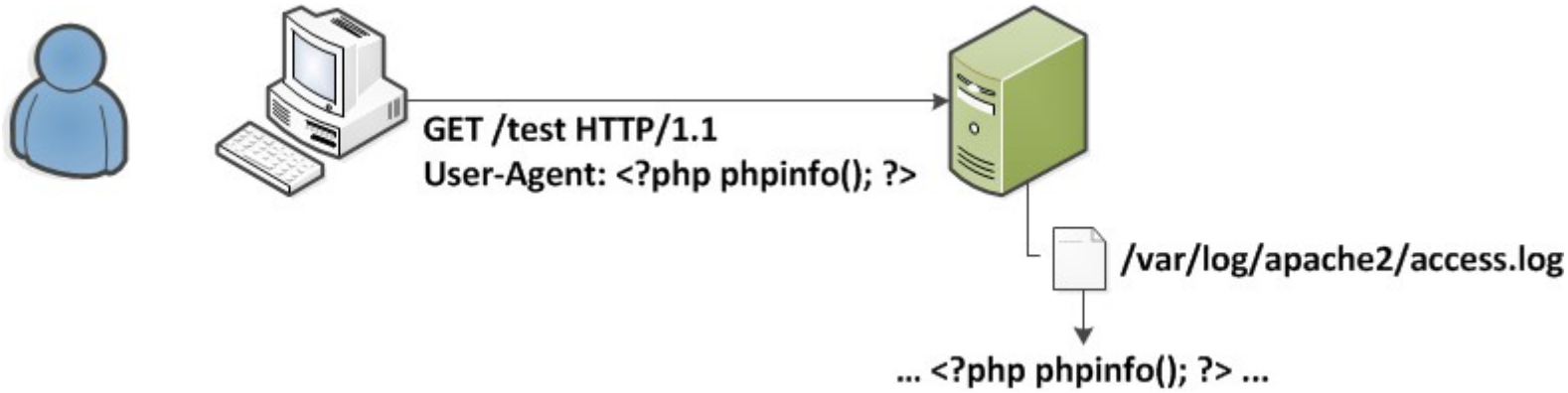
- Il est donc possible **d'inscrire** une **portion** de code **exécutable** (ici : du PHP) dans un **fichier** du serveur HTTP
- Il faut maintenant y **accéder** en utilisant la **vulnérabilité d'inclusion** de **fichier**, qui permet d'aller **inclure** un fichier sur le disque
- Pour ce faire, il faut **forger** une URL qui ira **chercher** ce fichier pour **l'inclure** dans la page de réponse
  - `http://site/?page=../../../var/log/apache2/access.log`

- **L'appel** à cette URL va provoquer chez le serveur une **inclusion** dans la page du fichier **`/var/log/apache2/access.log`**
- Le fichier **complet** sera **intégré**, y compris la ligne contenant **l'extrait de code** PHP inséré précédemment
- Une fois **inclus**, il sera **interprété** par le module PHP du serveur web
- **L'attaquant** aura donc réussi à obtenir une **exécution de code** à l'aide d'une simple inclusion de fichiers

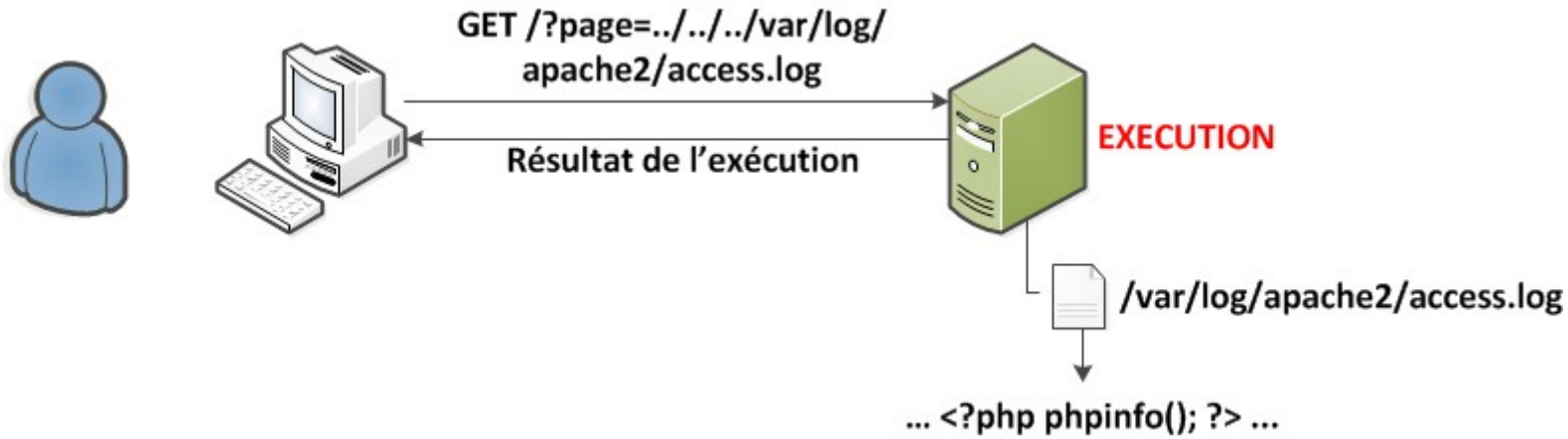
- **L'attaquant** commence par faire une **requête** contenant **l'extrait de code** malveillant



- Cette requête est **journalisée** dans le fichier **`/var/log/apache2/access.log`**
  - **L'extrait de code** fait partie du message journalisé



- Enfin, l'attaquant utilise la **vulnérabilité d'inclusion de fichier** pour **inclure** l'extrait de code journalisé et provoquer son **exécution**



- Lorsqu'on tente **d'inclure** un fichier de **journalisation** contenu dans le répertoire **`/var/log`**, on est **directement** concerné par les problématiques de **droits d'accès**
- Par **défaut**, les versions **récentes** d'Apache positionnent des droits **très restrictifs** sur le dossier **`/var/log/apache2`**
- Cela **empêche** d'inclure le contenu du fichier, donc l'accès sera **refusé** au serveur lorsqu'il essaiera de le **lire**

- Droits d'accès par défaut sur un système Debian

```
$ ls -al /var/log/|grep apache2
drwxr-x---  2 root    adm      4096 oct.  18 07:35 apache2
```

- Le répertoire est **totalement interdit** d'accès aux utilisateurs qui ne sont pas **`root`** et l'accès est **interdit** en **écriture** aux membres du groupe **`adm`**

```
$ ls -al /var/log/apache2
ls: impossible d'ouvrir le répertoire /var/log/apache2:
Permission non accordée
```

- C'est un exemple de l'intérêt d'exécuter les services avec un utilisateur **non-privilégié**

- La **vulnérabilité** d'inclusion de fichier peut utiliser des mécanismes **d'inclusion** ou **d'ouverture** de fichier qui supportent également les URL **complètes**
  - C'est le cas de la fonction **`include()`** de PHP, même si l'accès à une URL externe est **désactivé** par défaut
- Dans ce cas il est **facile** et **immédiat d'exploiter** cette vulnérabilité pour obtenir une **compromission directe** du serveur
  - L'attaquant passe l'URL d'un fichier **exécutable externe**, hébergé sur un de ses serveurs

- Pour se **protéger** de ces **vulnérabilités**, les **développeurs** ont souvent de **bonnes idées**
- On retrouve très **souvent** les **mêmes idées** dans des **applications différentes**
- Une **bonne partie** de ces idées sont de **fausses bonnes idées**, qui ne résistent pas à **l'astuce** d'un **attaquant motivé**
- Il est utile de **connaître** ces **mécanismes** de défense couramment **rencontrés** afin de savoir les **contourner**

- La **première** des mauvaises idées est d'essayer de « **cloisonner** » un attaquant au sein d'un répertoire précis en **préfixant** le **chemin** de fichier utilisé
- Exemple :

```
String page = request.getParameter("page");
File file = new File("./pages/jsp/" + page);
```

- Le développeur a tendance à penser que cela va **empêcher** l'utilisateur **d'accéder** à d'autres répertoires, alors que l'utilisation de **../** va **très simplement** lui permettre de s'en **émanciper**

- Il existe une **variante** de cette mauvaise idée, qui consiste à tenter **d'empêcher** l'utilisateur **d'atteindre** des fichiers de son choix en **suffixant** le chemin par une **extension** précise
- Exemple :

```
String img = request.getParameter("image");
File file = new File("img/" + img + ".jpg");
```

- On **peut** penser, en lisant cet extrait de code source qu'il ne sera **pas possible** d'ouvrir le fichier **`/etc/passwd`** à l'aide de ce code source
  - Il **existe** pourtant un **moyen** de le **contourner**

- Une **technique** de **contournement** utile dans ce genre de cas est l'utilisation d'un « ***null-byte*** »
- Il s'agit **d'inclure** des **caractères nuls** dans une **chaîne** de caractère pour provoquer des erreurs lors de son **traitement**
- Les chaînes de caractère sont traditionnellement représentées comme des **suites** de caractères terminées par un **caractère nul** (de valeur 0)
- Un caractère nul **inséré** en milieu de chaîne peut donc « **tronquer** » cette chaîne en **deux parties**

- Dans notre cas, nous allons **insérer** ce caractère nul en **fin** de chaîne de façon à **gêner** la **concaténation** de **l'extension**
- Le caractère nul peut être **inséré** dans une **URL** en utilisant la notation **%00**
- Nous **passons** donc au **serveur** une chaîne se **terminant** en réalité par **deux zéros**
  - Le zéro **légitime** et le zéro **rajouté**
- Lors de la **concaténation**, le zéro **rajouté** va forcer à **terminer** la chaîne **avant l'extension**

- Par exemple, si nous passons la chaîne « yves%00 » au serveur

y	v	e	s	0	0

- Lors de la **concaténation**, un autre **tableau** de caractère va lui être **ajouté**

y	v	e	s	0

+

.	j	p	g	0

y	v	e	s	0	.	j	p	g	0

- Malgré tout, la présence du **caractère nul** au milieu va **réduire** la chaîne et annuler l'extension

y	v	e	s	0

- Cette technique possède aussi **l'avantage** de pouvoir être **interprétée différemment** par les **modules** de l'application
- En particulier lors d'un **filtrage**
- On peut en effet tirer **profit** du fait que le **filtre** chargé d'empêcher une attaque et le **traitement** métier utilisent des **mécanismes différents**
- **L'objectif** étant de passer une chaîne qui **fonctionnera** dans le **module** métier mais sera **tronquée** au moment du **filtrage**

- Cela permet de contourner une autre mauvaise idée : les filtres sur l'extension
- Exemple

```
String page = request.getParameter("page");
If (page.endsWith(".jpg")) {
     File file = new File("images/" + page);
     …
}
```

- Dans ce cas, ce sont deux classes différentes qui se chargent de chaque traitement
  - La classe **`String`** pour le filtre
  - La classe **`File`** pour l'ouverture de fichier

- En fonction des implémentations et de la chaîne que l'on fournit, on peut réussir à obtenir un résultat où la chaîne passe le filtre et permet d'atteindre correctement le fichier
- En l'occurrence on pourrait essayer d'obtenir un tel résultat en passant la chaîne suivante : **`../../etc/passwd%00.jpg`**
- En fonction des implémentations, cette chaîne peut être intéprétée de deux façons différentes
  - **`../../etc/passwd`** ne se termine pas par **.jpg**
  - **`../../etc/passwd%00.jpg`** se termine par **.jpg**

- Lorsqu'on exploite une vulnérabilité d'inclusion de fichier local en injectant une portion de code source dans le fichier **`/var/log/apache2/access.log`** on peut être confronté à des filtres applicatifs
  - Le serveur embarque parfois des protections visant à supprimer les éléments dangereux, comme les balises PHP
  - Il peut également procéder à des encodages pour les besoins métiers, qui empêcheront l'exécution de la charge malveillante

- Dans ce cas, il est possible d'utiliser le mécanisme d'authentification HTTP
  - En-tête **`Authorization`**
- L'intérêt d'utiliser cet en-tête est qu'il est encodé en Base 64 lorsqu'il est transmis au serveur par le client
- Les attaques transmises par l'intermédiaire de cet en-tête ne pourront donc pas être détectées par un mécanisme de filtrage