1. **Cross-site Scripting**
   1. Definition

Le Cross-Site Scripting abrégé XSS est une faille de sécurité des applications utilisant un navigateur Web permettant à un attaquant d’injecter du code malveillant exécutable par ce navigateur. Il s’agit donc d’un cas d’injection de code à un interpréteur qui, à la différence des autres cas d’injection de code qui cible le serveur, cible le navigateur de l’utilisateur.

Cross-Site Scripting (XSS) is a security vulnerability which enables an attacker to place client side scripts (usually JavaScript) into web pages. When other users load affected pages the attackers scripts will run, enabling the attacker to steal cookies and session tokens, change the contents of the web page through DOM manipulation or redirect the browser to another page. XSS vulnerabilities generally occur when an application takes user input and outputs it in a page without validating, encoding or escaping it.

Le code injecté consiste généralement en du script JavaScript exécuté par le navigateur victime qui le prend pour du code légitime provenant de la même origine que le contenu html affiché. Ce script peut ainsi faire de nombreuses choses, allant d’une simple redirection à une usurpation de l’identité d’un utilisateur par vol des cookies de session. Et dans ce dernier cas, si l’utilisateur en question a des droits d’administrateur sur une certaine application, l’attaquant peut compromettre entièrement ladite application.

On note trois types d’attaques XSS :

* Reflected XSS
* Stored XSS
* DOM XSS

However, the possibility of JavaScript being malicious becomes more clear when you consider the following facts:

* JavaScript has access to some of the user's sensitive information, such as cookies.
* JavaScript can send HTTP requests with arbitrary content to arbitrary destinations by using XMLHttpRequest and other mechanisms.
* JavaScript can make arbitrary modifications to the HTML of the current page by using DOM manipulation methods.

Lors d’une attaque par injection SQL, l’attaquant fournit un/des segments malicieux de code SQL en tant qu’input transformant la requête SQL initiale en une autre souvent dangereuse. Les attaques par injection SQL sont utilisées pour dérober des informations sensibles de la base de données.

Par une injection SQL, l’attaquant, selon ses capacités peut obtenir des informations confidentielles ou même arriver à modifier ces informations.

* 1. Causes

Étant donné que les sites Web dynamiques reposent sur l'entrée de l'utilisateur, un utilisateur malveillant peut entrer un script malveillant dans la page en le masquant dans des demandes légitimes.

Le cross site scripting (XSS) affecte essentiellement des sites qui acceptent des données en provenance de l'utilisateur sans en contrôler correctement la conformité

La cause principale d’une injection SQL est le fait qu’une application accepte des données provenant de sources non sures comme l’utilisateur par exemple (input) ou le navigateur de l’utilisateur (cookies ou headers des requêtes http)

* 1. Vecteurs principaux
* http POST
* http GET
* Cookies
  1. Préventions

At a basic level XSS works by tricking your application into inserting a <script> tag into your rendered page, or by inserting an On\* event into an element. Developers should use the following prevention steps to avoid introducing XSS into their application.

1. Never put untrusted data into your HTML input, unless you follow the rest of the steps below. Untrusted data is any data that may be controlled by an attacker, HTML form inputs, query strings, HTTP headers, even data sourced from a database as an attacker may be able to breach your database even if they cannot breach your application.
2. Before putting untrusted data inside an HTML element ensure it's HTML encoded. HTML encoding takes characters such as < and changes them into a safe form like &lt;
3. Before putting untrusted data into an HTML attribute ensure it's HTML attribute encoded. HTML attribute encoding is a superset of HTML encoding and encodes additional characters such as " and '.
4. Before putting untrusted data into JavaScript place the data in an HTML element whose contents you retrieve at runtime. If this isn't possible then ensure the data is JavaScript encoded. JavaScript encoding takes dangerous characters for JavaScript and replaces them with their hex, for example < would be encoded as \u003C.
5. Before putting untrusted data into a URL query string ensure it's URL encoded.

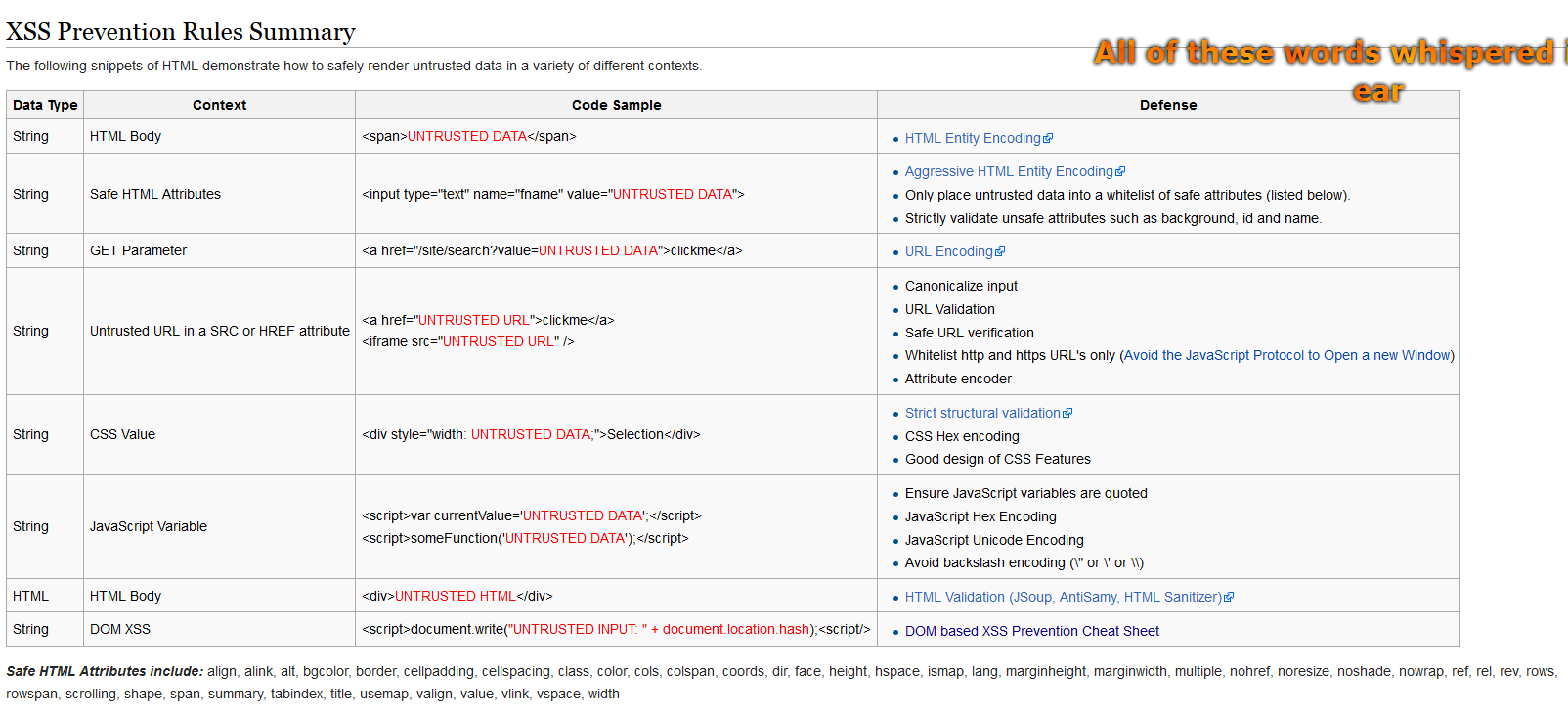
<script>**...NEVER PUT UNTRUSTED DATA HERE...**</script> directly in a script

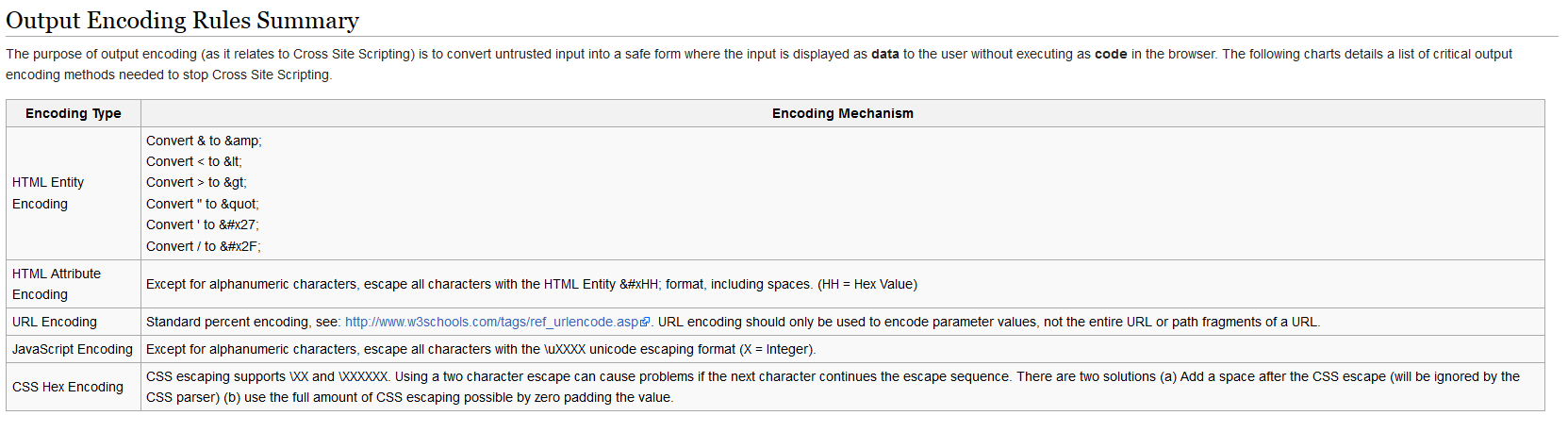
<!--**...NEVER PUT UNTRUSTED DATA HERE...**--> inside an HTML comment

<div **...NEVER PUT UNTRUSTED DATA HERE...**=test /> in an attribute name

<**NEVER PUT UNTRUSTED DATA HERE...** href="/test" /> in a tag name

<style>**...NEVER PUT UNTRUSTED DATA HERE...**</style> directly in CSS





**Content Security Policy (CSP)**

The disadvantage of protecting against XSS by using only secure input handling is that even a single lapse of security can compromise your website. A recent web standard called Content Security Policy (CSP) can mitigate this risk.

CSP is used to constrain the browser viewing your page so that it can only use resources downloaded from trusted sources. A *resource* is a script, a stylesheet, an image, or some other type of file referred to by the page. This means that even if an attacker succeeds in injecting malicious content into your website, CSP can prevent it from ever being executed.

#### How to enable CSP

By default, browsers do not enforce CSP. To enable CSP on your website, pages must be served with an additional HTTP header: Content‑Security‑Policy. Any page served with this header will have its security policy respected by the browser loading it, provided that the browser supports CSP.

Since the security policy is sent with every HTTP response, it is possible for a server to set its policy on a page-by-page basis. The same policy can be applied to an entire website by providing the same CSP header in every response.

The value of the Content‑Security‑Policy header is a string defining one or more security policies that will take effect on your website. The syntax of this string will be described next.

The example headers in this section use newlines and indentation for clarity; this should not be present in an actual header.

#### Syntax of CSP

The syntax of a CSP header is as follows:

Content‑Security‑Policy:  
    directive source‑expression, source‑expression, ...;  
    directive ...;  
    ...

This syntax is made up of two elements:

* **Directives** are strings specifying a type of resource, taken from a predefined list.
* **Source expressions** are patterns describing one or more servers that resources can be downloaded from.

For every directive, the given source expressions define which sources can be used to download resources of the respective type.

##### Directives

The directives that can be used in a CSP header are as follows:

* connect‑src
* font‑src
* frame‑src
* img‑src
* media‑src
* object‑src
* script‑src
* style‑src

In addition to these, the special directive default‑src can be used to provide a default value for all directives that have not been included in the header.

##### Source expressions

The syntax of a source expression is as follows:

protocol://host‑name:port‑number

The host name can start with \*., which means that any subdomain of the provided host name will be allowed. Similarly, the port number can be \*, which means that all ports will be allowed. Additionally, the protocol and port number can be omitted. Finally, a protocol can be given by itself, which makes it possible to require that all resources be loaded using HTTPS.

In addition to the syntax above, a source expression can alternatively be one of four keywords with special meaning (quotes included):

'none'

Allows no resources.

'self'

Allows resources from the host that served the page.

'unsafe‑inline'

Allows resources embedded in the page, such as inline <script> elements, <style> elements, and javascript: URLs.

'unsafe‑eval'

Allows the use of the JavaScript eval function.

Note that whenever CSP is used, inline resources and eval are automatically disallowed by default. Using 'unsafe‑inline' and 'unsafe‑eval' is the only way to allow them.

#### An example policy

Content‑Security‑Policy:  
    script‑src 'self' scripts.example.com;  
    media‑src 'none';  
    img‑src \*;  
    default‑src 'self' http://\*.example.com

* The most important way of preventing XSS attacks is to perform secure input handling.
  + Most of the time, encoding should be performed whenever user input is included in a page.
  + In some cases, encoding has to be replaced by or complemented with validation.
  + Secure input handling has to take into account which context of a page the user input is inserted into.
  + To prevent all types of XSS attacks, secure input handling has to be performed in both client-side and server-side code.
* Content Security Policy provides an additional layer of defense for when secure input handling fails.
* Eliminate dangerous insertion points

There are some locations within the application page where it is just too inherently

dangerous to insert user-supplied input, and developers should look for

an alternative means of implementing the desired functionality.

Inserting user-controllable data directly into existing script code should be

avoided wherever possible. This applies to code within <script> tags, and also

code within event handlers. When applications attempt to do this safely, it is

frequently possible to bypass their defensive fi lters. And once an attacker has

taken control of the context of the data he controls, he typically needs to perform

minimal work to inject arbitrary script commands and therefore perform

malicious actions.

Where a tag attribute may take a URL as its value, applications should generally

avoid embedding user input, because various techniques may be used to

introduce script code, including the use of scripting pseudo-protocols.

A further pitfall to avoid is situations where an attacker can manipulate the

character set of the application’s response, either by injecting into a relevant

directive or because the application uses a request parameter to specify the

preferred character set. In this situation, input and output fi lters that are well

designed in other respects may fail because the attacker’s input is encoded

in an unusual form that the fi lters do not recognize as potentially malicious.

Wherever possible, the application should explicitly specify an encoding type

in its response headers, disallow any means of modifying this, and ensure that

its XSS fi lters are compatible with it. For example:

Content-Type: text/html; charset=ISO-8859-1

* + 1. Input validation

Une première défense est de valider les données fournies à l’application en mettant assez de contrainte sur les inputs, préciser éventuellement la longueur maximale, la longueur minimale, le type. Ces données peuvent provenir de champs de formulaire, de cookies, d’APIS externe ou encore d’un serveur proxy. Il faut se dire que toute donnée fournie à l’application est dangereuse. Pour s’assurer qu’un certain format est respecté, on peut utiliser les regex.

La validation des données saisies est faite pour s’assurer que seules les données correctement formées sont acceptées par le système, ce qui empêche la persistence des données mal formées et éventuellement dangereuses pour la base de données. La validation des entrées doit se faire le plus tôt possible. Cela permet de détecter les données malicieuses avant même qu’elles ne parviennent à l’application. Cette validation doit être faite à la fois côté client et côté serveur pour se protéger pour assurer le maximum de sécurité.

Plusieurs validations doivent être effectuées :

* Validation de la source des inputs : il faut aussi valider la source de l’input. Si l’input est attendu d’une requête GET par exemple, ne pas accepter un input provenant d’une requête POST.
* validation des types des données : s’assurer que les données saisies sont du type requis par l’application : on peut faire une conversion vers le type requis pour s’assurer que c’est le cas (Integer.parseInt() par exemple)
* validation de la taille des données saisies : préciser des valeurs minimales et maximales pour les données numériques et les dates, longueurs minimale et maximale pour les chaines de caractères. Peut être fait avec les attributs min et max, minLength et maxLength de l’élément input côté client et avec l’utilisation de regex côté serveur.
* validation des formats des données : la meilleure façon de faire cette validation est l’utilisation des regex. De même, il faut privilégier le white listing au black listing : n’accepter que les données respectant un certain format. Utilisation des Regex
* sanitisation : il s’agit de rendre les données saisies potentiellement dangereux pour les interpréteurs SQL non interprétables par un processus d’échappement. Pour ce faire, il est recommandé d’utiliser MySQL Codec ou Oracle Codec.

En résumé, pour toute donnée provenant de l’utilisateur, il faut (éventuellement):

* définir l’ensemble des valeurs acceptées avec l’aide de regex
* caster la (les) donnée(s) pour s’assurer qu’elles sont du type demandé
* définir les longueurs maximale et minimale
* sanitiser
* définir un regex de validation
* définir la source des données attendues
  + 1. Requêtes préparées

L’utilisation de requêtes préparées est la forme de protection la plus effective contre les injections SQL. En effet les requêtes préparées sont pré compilées par le SGBD c’est-à-dire que le SGBD sait déjà ce que doit faire cette requête et peuvent contenir des paramètres qui leur seront fournis lors de l’exécution. Lorsque ces paramètres sont reçus, ils sont placés comme valeurs dans la requête précompilée et ne sont jamais interprétés en tant que code SQL. Ainsi, la pré-compilation, (une compilation faite dès le début et une seule fois) couplée à la prise en compte de paramètres au moment de l'exécution, permet d'améliorer considérablement les performances de requêtes paramétrées destinées à être exécutées plusieurs fois et protègent contre les injections SQL.

A parameterized query is a query in which placeholders are used for parameters and the parameter values are supplied at execution time. The most important reason to use parameterized queries is to avoid SQL injection attacks.

* + 1. Défenses supplémentaires
       1. Réduire la surface d’attaques en donnant le moins de privilèges

Utiliser plusieurs utilisateurs de Base de données dans l’application. Par exemple pour la connexion, on a juste besoin d’un privilège de lecture de la base de données, on peut créer un utilisateur avec juste ce privilège et utiliser l’*entity manager\** avec la persistence unit utilisant une datasource avec cet utilisateur dans l’Ejb pour gérer la connexion. Pour l’inscription par contre on a besoin de pouvoir lire et insérer dans la base, on crée un utilisateur avec ces privilèges et ainsi de suite. Séparer les privilèges en plusieurs utlisateurs.

* + - 1. Crypter toutes les données sensibles

En matière de sécurité, il faut toujours considérer les pires des cas : vol d’informations, suppression, modification et de données. De ce fait, il est important de

* crypter toutes les données sensibles présentes en base de données. Ce cryptage doit aussi être fait sur les données enregistrées dans des fichiers ou encore sous forme de cookies au niveau du navigateur de l’utilisateur. Il faut crypter aussi les données sensibles en transit en utlisant un algorithme de cryptage avec sel.
* Logger toutes les requêtes arrivant à la BD : prévoir un fichier pour chaque type de requête (Create, Read, Update, Delete) et faire un audit périodique de ces fichiers.
* Prévoir au moins une base de données de réplication et faire périodiquement la copie des données présentes en base de données ainsi que leur conservation dans des endroits sécurisés de préférence physique.
  + - 1. Se méfier des headers http

Les en-têtes HTTP sont envoyés au début des requêtes HTTP et des réponses HTTP. L’ application Web doit s'assurer qu'elle ne base aucune décision de sécurité sur les informations contenues dans les en-têtes HTTP car il est facile pour un attaquant de manipuler l'en-tête. Par exemple, le champ referer dans l'en-tête contient l'URL de la page Web d'où provient la demande. Ne prenez aucune décision de sécurité basée sur la valeur du champ référant, par exemple, pour vérifier si la **requête** provient d'une page générée par l'application Web, car le champ est facilement falsifié.

import java.io.IOException;

import java.security.MessageDigest;

import java.security.NoSuchAlgorithmException;

import java.security.SecureRandom;

import java.util.ArrayList;

import java.util.List;

import javax.servlet.Filter;

import javax.servlet.FilterChain;

import javax.servlet.FilterConfig;

import javax.servlet.ServletException;

import javax.servlet.ServletRequest;

import javax.servlet.ServletResponse;

import javax.servlet.annotation.WebFilter;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.apache.commons.codec.binary.Hex;

/\*\*

\* Sample filter implementation to define a set of Content Security Policies.<br/>

\*

\* This implementation has a dependency on Commons Codec API.<br/>

\*

\* This filter set CSP policies using all HTTP headers defined into W3C specification.<br/>

\* <br/>

\* This implementation is oriented to be easily understandable and easily adapted.<br/>

\*

\*/

@WebFilter("/\*")

public class CSPPoliciesApplier implements Filter {

/\*\* Configuration member to specify if web app use web fonts \*/

public static final boolean APP\_USE\_WEBFONTS = false;

/\*\* Configuration member to specify if web app use videos or audios \*/

public static final boolean APP\_USE\_AUDIOS\_OR\_VIDEOS = false;

/\*\* Configuration member to specify if filter must add CSP directive used by Mozilla (Firefox) \*/

public static final boolean INCLUDE\_MOZILLA\_CSP\_DIRECTIVES = true;

/\*\* Filter configuration \*/

@SuppressWarnings("unused")

private FilterConfig filterConfig = null;

/\*\* List CSP HTTP Headers \*/

private List<String> cspHeaders = new ArrayList<String>();

/\*\* Collection of CSP polcies that will be applied \*/

private String policies = null;

/\*\* Used for Script Nonce \*/

private SecureRandom prng = null;

/\*\*

\* Used to prepare (one time for all) set of CSP policies that will be applied on each HTTP response.

\*

\* @see javax.servlet.Filter#init(javax.servlet.FilterConfig)

\*/

@Override

public void init(FilterConfig fConfig) throws ServletException {

// Get filter configuration

this.filterConfig = fConfig;

// Init secure random

try {

this.prng = SecureRandom.getInstance("SHA1PRNG");

}

catch (NoSuchAlgorithmException e) {

throw new ServletException(e);

}

// Define list of CSP HTTP Headers

this.cspHeaders.add("Content-Security-Policy");

this.cspHeaders.add("X-Content-Security-Policy");

this.cspHeaders.add("X-WebKit-CSP");

// Define CSP policies

// Loading policies for Frame and Sandboxing will be dynamically defined : We need to know if context use Frame

List<String> cspPolicies = new ArrayList<String>();

String originLocationRef = "'self'";

// --Disable default source in order to avoid browser fallback loading using 'default-src' locations

cspPolicies.add("default-src 'none'");

// --Define loading policies for Scripts

cspPolicies.add("script-src " + originLocationRef + " 'unsafe-inline' 'unsafe-eval'");

if (INCLUDE\_MOZILLA\_CSP\_DIRECTIVES) {

cspPolicies.add("options inline-script eval-script");

cspPolicies.add("xhr-src 'self'");

}

// --Define loading policies for Plugins

cspPolicies.add("object-src " + originLocationRef);

// --Define loading policies for Styles (CSS)

cspPolicies.add("style-src " + originLocationRef);

// --Define loading policies for Images

cspPolicies.add("img-src " + originLocationRef);

// --Define loading policies for Form

cspPolicies.add("form-action " + originLocationRef);

// --Define loading policies for Audios/Videos

if (APP\_USE\_AUDIOS\_OR\_VIDEOS) {

cspPolicies.add("media-src " + originLocationRef);

}

// --Define loading policies for Fonts

if (APP\_USE\_WEBFONTS) {

cspPolicies.add("font-src " + originLocationRef);

}

// --Define loading policies for Connection

cspPolicies.add("connect-src " + originLocationRef);

// --Define loading policies for Plugins Types

cspPolicies.add("plugin-types application/pdf application/x-shockwave-flash");

// --Define browser XSS filtering feature running mode

cspPolicies.add("reflected-xss block");

// Target formating

this.policies = cspPolicies.toString().replaceAll("(\\[|\\])", "").replaceAll(",", ";").trim();

}

/\*\*

\* Add CSP policies on each HTTP response.

\*

\* @see javax.servlet.Filter#doFilter(javax.servlet.ServletRequest, javax.servlet.ServletResponse, javax.servlet.FilterChain)

\*/

@Override

public void doFilter(ServletRequest request, ServletResponse response, FilterChain fchain) throws IOException, ServletException {

HttpServletRequest httpRequest = ((HttpServletRequest) request);

HttpServletResponse httpResponse = ((HttpServletResponse) response);

/\* Step 1 : Detect if target resource is a Frame \*/

// Customize here according to your context...

boolean isFrame = true;

/\* Step 2 : Add CSP policies to HTTP response \*/

StringBuilder policiesBuffer = new StringBuilder(this.policies);

// If resource is a frame add Frame/Sandbox CSP policy

if (isFrame) {

// Frame + Sandbox : Here sandbox allow nothing, customize sandbox options depending on your app....

policiesBuffer.append(";").append("frame-src 'self';sandbox");

if (INCLUDE\_MOZILLA\_CSP\_DIRECTIVES) {

policiesBuffer.append(";").append("frame-ancestors 'self'");

}

}

// Add Script Nonce CSP Policy

// --Generate a random number

String randomNum = new Integer(this.prng.nextInt()).toString();

// --Get its digest

MessageDigest sha;

try {

sha = MessageDigest.getInstance("SHA-1");

}

catch (NoSuchAlgorithmException e) {

throw new ServletException(e);

}

byte[] digest = sha.digest(randomNum.getBytes());

// --Encode it into HEXA

String scriptNonce = Hex.encodeHexString(digest);

policiesBuffer.append(";").append("script-nonce ").append(scriptNonce);

// --Made available script nonce in view app layer

httpRequest.setAttribute("CSP\_SCRIPT\_NONCE", scriptNonce);

// Add policies to all HTTP headers

for (String header : this.cspHeaders) {

httpResponse.setHeader(header, policiesBuffer.toString());

}

/\* Step 3 : Let request continue chain filter \*/

fchain.doFilter(request, response);

}

/\*\*

\* {@inheritDoc}

\*

\* @see javax.servlet.Filter#destroy()

\*/

@Override

public void destroy() {

// Not used

}

}