# SECURITE PROGRAMMES

Eric BOUDIN

Florian NAUD

Soulaiman ZABOURDINE

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ATTAQUE | LIEN | Programme vulnérable? | Exploitation | Défense | Page |
| XSS RULE 0 | [Cross\_Site\_Scripting\_Prevention\_Cheat\_Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#rule-0-never-insert-untrusted-data-except-in-allowed-locations) | OUI | OUI | OUI | 2 |
| XSS RULE 1 | [Cross\_Site\_Scripting\_Prevention\_Cheat\_Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#rule-1-html-encode-before-inserting-untrusted-data-into-html-element-content) | OUI | OUI | OUI | 3 |
| XSS RULE 2 | [Cross Site Scripting Prevention Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#rule-2-attribute-encode-before-inserting-untrusted-data-into-html-common-attributes) | OUI | OUI | OUI | 4 |
| XSS RULE 3 | [Cross Site Scripting Prevention Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#rule-2-attribute-encode-before-inserting-untrusted-data-into-html-common-attributes) |  |  |  | 5 |
| XSS RULE 4 | [Cross Site Scripting Prevention Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#rule-2-attribute-encode-before-inserting-untrusted-data-into-html-common-attributes) |  |  |  | 6 |
| XSS RULE 5 | [Cross Site Scripting Prevention Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#rule-2-attribute-encode-before-inserting-untrusted-data-into-html-common-attributes) | OUI | OUI | OUI | 7 |
| XSS RULE 6 | [Cross Site Scripting Prevention Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#rule-2-attribute-encode-before-inserting-untrusted-data-into-html-common-attributes) |  |  |  | 8 |
| XSS RULE 7 | [Cross Site Scripting Prevention Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#rule-2-attribute-encode-before-inserting-untrusted-data-into-html-common-attributes) |  |  |  | 9 |
| XSS RULE 8 | [Cross Site Scripting Prevention Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#rule-2-attribute-encode-before-inserting-untrusted-data-into-html-common-attributes) |  |  |  | 10 |
| DOM XSS | [DOM\_based\_XSS\_Prevention\_Cheat\_Sheet](https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html) | OUI | OUI | OUI | 11 |
| Session ID | [Session\_Management\_Cheat\_Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html) |  |  |  | 12 |
| Session Management | [Session\_Management\_Cheat\_Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html) |  |  |  | 13 |
| Session Cookie | [Session\_Management\_Cheat\_Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html) | OUI | OUI | OUI | 14 |
| XML ENTITY | [XML\_External\_Entity\_Prevention\_Cheat\_Sheet](https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html) | OUI | OUI | OUI | 15 |
| SQL INJECTION | [SQL\_Injection\_Prevention\_Cheat\_Sheet](https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html) | OUI | OUI | OUI | 16 |
| Password Storage | [Password\_Storage\_Cheat\_Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html) | OUI | OUI | OUI | 17 |

## XSS RULE #0 - Never Insert Untrusted Data Except in Allowed Locations

[Cross\_Site\_Scripting\_Prevention\_Cheat\_Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html" \l "rule-0-never-insert-untrusted-data-except-in-allowed-locations)

Une image contenant texte

Description générée automatiquement

L’utilisateur peut se contenter d’entrer de quoi fermer la balise script suivi de l’appel à un script malicieux. Exemple: </div><script>alert("YOU HAVE BEEN HACKED")</script>

Pour s’en protéger, il faut encoder $content avec htmlspecialchars.

## XSS RULE #1 - HTML Encode Before Inserting Untrusted Data into HTML Element Content

[Cross\_Site\_Scripting\_Prevention\_Cheat\_Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#rule-1-html-encode-before-inserting-untrusted-data-into-html-element-content)

Une image contenant texte

Description générée automatiquement

Le principe est identique a la règle 0. Même attaque, même défense.

## XSS RULE #2 - Attribute Encode Before Inserting Untrusted Data into HTML Common Attributes

[Cross Site Scripting Prevention Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#rule-2-attribute-encode-before-inserting-untrusted-data-into-html-common-attributes)

Une image contenant texte

Description générée automatiquement

L’utilisateur peut se contenter d’entrer de quoi fermer la balise script suivi de l’appel à un script malicieux. Exemple: </div><script>alert("YOU HAVE BEEN HACKED")</script>

Pour s’en protéger, il faut encoder $content avec htmlspecialchars.

## XSS RULE #3 - JavaScript Encode Before Inserting Untrusted Data into JavaScript Data Values

[Cross Site Scripting Prevention Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#rule-3-javascript-encode-before-inserting-untrusted-data-into-javascript-data-values)

Une image contenant texte

Description générée automatiquement

Ici on veut afficher ce que rentre l’utilisateur. Comme d’habitude, on peut bloquer le code malicieux avec htmlspecialchars. Mais ce n’est pas suffisant. Il faut également quoter car l’utilisateur pourrait par exemple demander d’afficher document.cookie, qui reste identique même après encodage.

## XSS RULE #4 - CSS Encode And Strictly Validate Before Inserting Untrusted Data into HTML Style Property Values

[Cross Site Scripting Prevention Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#rule-4-css-encode-and-strictly-validate-before-inserting-untrusted-data-into-html-style-property-values)

Ici, il faut protéger le code situer dans l’attribut href. Il est possible de faire ça en php avec la fonction htmlspecialchars.

Une image contenant texte

Description générée automatiquement

Il suffit de ne pas ajouter cette méthode pour rendre le programme vulnérable. Pour attaquer, on peut utiliser ce qui est en commentaire : " onmouseover='alert(1)'

## XSS RULE #5 URL Encode Before Inserting Untrusted Data into HTML URL Parameter Values

[Cross\_Site\_Scripting\_Prevention\_Cheat\_Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#rule-5-url-encode-before-inserting-untrusted-data-into-html-url-parameter-values)

Ici, il faut protéger le code situer dans l’attribut href. Il est possible de faire ça en php avec la fonction htmlspecialchars.

Une image contenant texte

Description générée automatiquement

Il suffit de ne pas ajouter cette méthode pour rendre le programme vulnérable. Pour attaquer, on peut utiliser ce qui est en commentaire : " onmouseover='alert(1)'

## XSS RULE #6 - Sanitize HTML Markup with a Library Designed for the Job

[Cross\_Site\_Scripting\_Prevention\_Cheat\_Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#rule-6-sanitize-html-markup-with-a-library-designed-for-the-job)

## DOM XSS

[DOM](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#rule-1-html-encode-before-inserting-untrusted-data-into-html-element-content) Based XSS Prevention Cheat Sheet

Ici la vulnérabilité se situe sur l’utilisation dangereuse de la méthode ‘eval’ qui permet d’évaluer une fonction mathématique sous la forme d’une String, mais peut également être utiliser pour exécuter du code javascript. Dans l’exemple ci-dessous, nous avons une calculatrice qui cherche la fonction dans l’URL afin de l’exécuter <http://localhost:63342/DOM/Dom.html?ijt=1sb2f90cj?value=10+10>).

Text

Description automatically generated

L’attaquant peut profiter de cette faille en remplaçant ‘10+10’ par du code malicieux. Par exemple, ici en remplaçant cette la chaine de caractères ‘10+10’ par ‘alert(document.cookie)’ l’attaquant peut avoir accès au cookie, et bien d’autres ressources sensibles.

Graphical user interface, text, application

Description automatically generated

Pour éviter ce genre de faille dans le site, il faut à tout prix éviter l’utilisation d’une méthode telle qu’eval dans son site.

## SESSION ID

## SESSION MANAGEMENT IMPLEMENTATION

## SESSION COOKIE

L’id d’une session doit être protégé afin qu’un utilisateur malveillant ne puisse pas s’approprié la session d’une autre personne.

De base avec les fonctions php, l’id de la session est stocké dans les cookies. Cependant, si un utilisateur malveillant arrive à récupérer les cookies de l’utilisateur à cause par exemple d’une faille XSS. Alors sa session sera compromise.

Une image contenant texte

Description générée automatiquement

Pour résoudre ce problème, on peut sécuriser le cookie possédant le session id en lui donnant un temps d’expiration ainsi qu’en appliquant les option **secure** et **httponly**.



Une image contenant texte

Description générée automatiquement

## XML EXTERNAL ENTITY PREVENTION

Afin de démontrer cette vulnérabilité, nous allons nous aider d’un programme PHP ‘welcome.php’ qui tourne sur un serveur, et qui permet tout simplement d’afficher une valeur (invite->prenom) grâce à un fichier XML fourni par un utilisateur.

Text

Description automatically generated

Nous allons également utiliser le programme ‘request.php’ PHP ci-dessous afin de simuler les appels d’un utilisateur vers notre serveur :

Text

Description automatically generated

Le résultat généré par ‘request.php’ sera donc la suivante :

Graphical user interface, text, application

Description automatically generated

Le fait de ne pas contrôler l’XML fourni, en tolérant notamment les entités externes soumettent le serveur à un danger important car un hacker pourrait l’utiliser pour accéder aux données du serveur. Pour cela, il doit tout simplement s’appuyer sur un ‘External Document Type Definition’ ou External DTD.

Vous retrouvez ci-dessous un exemple de cette vulnérabilité, ou l’hacker pourrait accéder à un fichier ’passwd’ en modifiant le code de ‘request.php’. Vous retrouverez également le nouveau résultat de la requête.

Text

Description automatically generatedGraphical user interface, text

Description automatically generated

Pour se protéger de cette vulnérabilité, il suffit de bannir les entités externes, en ajoutant la ligne suivante au programme PHP du serveur.

A screenshot of a computer

Description automatically generated with medium confidence

## SQL INJECTION 1

Nos données peuvent être vulnérable si on ne contrôle pas ce qu’injecte les utilisateurs de nos applications. Notre requête sql renvoie la valeur d’un utilisateur donné.



Ici le problème c’est que l’utilisateur peut rentrer n’importe quoi dans le script comme

 (Ce qui donnera la requête sql 2)

On peut donc voir qu’il récupère toutes les données enregistrées dans la table.

Pour nous protéger de cela on peut préparer une requête sql.



Ici même en tentant de remettre notre faille, cela ne fonctionnera pas car cela considérera la faille comme un paramètre.

Une image contenant texte

Description générée automatiquement

## Password Storage

Un mot de passe mal stocké dans une base de données peut amener à une faille de sécurité autant pour l’application que pour le client qui à la mauvaise habitude d’utiliser le même mot de passe (ou un mot de passe très proche) pour chacun de ses comptes.

On ne stock pas un mot de passe en dur dans une base de données !

Une image contenant texte

Description générée automatiquement



Ici notre application a stocké en dur le mot de passe d’Eric. Il suffit qu’une personne arrive à récupérer soit depuis le script soit depuis la BDD ce mot de passe et le compte d’Eric est compromis.

On a deux solutions pour protéger le stockage d’un mot de passe. La première est l’encodage. Cette méthode est cependant déconseillé pour des informations très importante comme un mot de passe car elles peuvent être décodé grâce à la clé d’encodage. L’autre méthode est le hachage. Cette méthode est très conseillé car il est quasiment impossible de retourner la valeur haché à son état d’origine.

Pour rajouter encore plus de sécurité au hachage on a 2 moyens. Le premier est de salé, c’est-à-dire rajouté un élément connu à notre mot de passe puis on le hache. Le second moyen est de poivré, c’est-à-dire d’encoder notre hash avec une clé garder par notre logiciel.



Notre mot de passe est haché par une fonction php qui s’occupe aussi de salé avec une valeur aléatoire. On la stock ensuite dans notre base de données.

Une image contenant texte

Description générée automatiquement

Ensuite on utilisera une autre méthode pour comparer ce que l’utilisateur donne comme mot de passe ($password) avec le mot de passe haché stocké sur la base de données ($res).

Une image contenant texte

Description générée automatiquement

Ici, on a fait un petit test pour montrer les résultat des fonctions. On a d’abord utilisé 2 mot de passe différents, 1 bon et 1 mauvais pour pouvoir les comparer. Ensuite sur la ligne 3 on a haché le bon mot de passe et sur la ligne 4 on récupère le mot de passe stocké sur la base de données. On peut voir qu’avec le même mot de passe les deux hache sont différents. Enfin on utilise la fonction pour vérifier si le mot de passe est correcte ou non et on constate qu’il a pu retrouver que le premier mot de passe était le bon mot de passe.