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 | 2 |
| 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 |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| 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 |

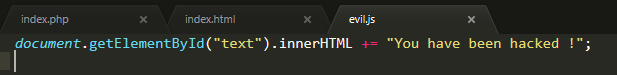
## RULE #0 - Never Insert Untrusted Data Except in Allowed Locations

[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

Ici on exécute du code venu d’une autre origine que nous. Ainsi, on ne peut garantir qu’il n’y est pas du code malveillant. Dans ce cas-là, cela change l’affichage du site.





Pour se protéger de ceci, on peut mettre appeler le script depuis un iframe. Ainsi la page ne sera plus modifiée.

Une image contenant texte

Description générée automatiquement

## 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)

Ici, c’est lors de l’envoie de données dans un input que se situe la faille.

Une image contenant texte

Description générée automatiquement

Ici, si l’on met du code JS dans l’input, il sera exécuté.

Une image contenant texte

Description générée automatiquement

Pour s’en protéger, il faut encoder les données :

Une image contenant texte

Description générée automatiquement

## 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)

Le problème ici concerne encore l’encodage des données non sûres, cette fois dans un attribut href.

Une image contenant texte

Description générée automatiquement

Ici, le script pourrait très bien remplacer le contenu de href par  ce qui exécutera la fonction attack. Une façon très simple de contrer ceci est de remplacer les guillemets par leur version encodé, c’est-à-dire &#x22. Ainsi le code ne sera pas exécuté mais le texte sera affiché en brut.

Une image contenant texte, personne, capture d’écran

Description générée automatiquement

## 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.