Ymox

Zeste de savoir

Ne pas confondre faille par injection SQL et faille XSS

29 août 2023

Table des matières

Introduction

> – [**Personne 1**] Help ! J'ai besoin d'aide ! Mon code ne fait pas ce que je veux !
>
> *Le début d'une conversation presque fictive au détour d'un forum*

A la base, il y a ceci.

```
$valeur1 = htmlspecialchars($_POST['truc']);
// Ou pire encore
$valeur2 = htmlentities($_GET['chose']);
```

Puis viennent deux variantes possibles, selon la source d'inspiration.

Variante 1 Voici du code dinosaure, mais j'en avais vu encore la semaine où ce contenu avait été initialement publié. On peut encore en trouver dans de vieux tutoriels à succès et donc bien référencés, mais malheureusement plus (mis) à jour.

```
$requete = "INSERT INTO matable (macolonne1, macolonne2)
    VALUES ($valeur1, $valeur2)";
mysql_query($requete);
```

Il y a aussi les variantes avec PDO::exec ou `mysqli_query()`.

Variante 2 Un peu plus réfléchie, mais plus préoccupante à mon avis (ici avec PDO, mais adaptable avec mysqli).

```
$requete = $pdo->prepare(
    'INSERT INTO matable (macolonne1, macolonne2) VALUES
    (:valeur1, :valeur2)'
);
$requete->execute(array(':valeur1' => $valeur1, ':valeur2' =>
    $valeur2));
```

> – [**Personne 2**] Salut ! Ton problème est *là*. Aussi, je te conseille de ne pas utiliser `htmlspecialchars()` ni `htmlentities()` sur des données à envoyer dans la base, elles s'utilisent à l'affichage.
> – [**Personne 1**] Ben si, il faut que j'utilise `htmlspecialchars()`, c'est pour sécuriser mes données !
>
> *Suite d'une conversation presque fictive au détour d'un forum*

Depuis quelques temps, j'assiste à une recrudescence de cas comme celui mis en exemple ci-dessus. Ces discussions et cette portion de code me font penser que quelque part, il y a confusion—voire amalgame—entre les failles par injection SQL et XSS.

L'idée de cette publication est de bien dissocier les deux types et d'éviter que les solutions pour l'un soient utilisées pour l'autre, ou même pour les deux.

1. Une bien mauvaise pratique…

Première chose, qui m'arrache l'œil: l'utilisation de `htmlspecialchars()` ou de `htmlentities()` directement sur des données `$_POST` ou `$_GET`. D'aucuns soutiennent mordicus que c'est pour protéger leur base de données. Je certifie que ce n'est **pas avec l'une ou l'autre de ces fonctions que votre base de données est en sécurité**.

Afin de comprendre pourquoi, commençons par regarder ce que font ces deux fonctions. Pour ça, rendez-vous dans la bonne vieille documentation officielle.

1.0.0.1. `htmlspecialchars`

```
<?php
string htmlspecialchars ( string $string [, int $flags =
    ENT_COMPAT | ENT_HTML401 [, string $encoding =
    ini_get("default_charset") [, bool $double_encode = TRUE
    ]]] )
```

Documentation officielle pour `htmlspecialchars()`

L'explication est sobre: la fonction convertit les caractères spéciaux en entités HTML, et il y a heureusement un tableau un peu plus bas. Ce qu'il faut surtout noter, c'est que les guillemets simples ne sont traités que si on spécifie `ENT_QUOTES` dans le second paramètre, et que cette valeur n'est pas spécifiée par défaut. En clair: **les guillemets simples ne sont pas modifiés quand cette fonction est utilisée avec seulement le premier paramètre de renseigné**, et ce depuis PHP 5.3.29 pour la version la plus vieille que j'aie testée.
A noter que la documentation mentionne aussi les changements apportés au fonctionnement de la fonction. Or, l'historique mentionne un changement au niveau de la valeur par défaut pour le second paramètre que depuis PHP 8.1.0. On peut raisonnablement penser que ce comportement n'avait donc pas changé depuis l'introduction de la fonction dans le noyau de PHP, ce qui remonte quand même à une version 4, dont la première mouture date de mai 2000 environ!

Si depuis PHP 8.1 il y a bien eu un changement qui convertit les guillemets, l'utilisation de `htmlspecialchars()` présentée en introduction **reste une mauvaise utilisation**

1.0.0.2. `htmlentities`

```php
<?php
string htmlentities ( string $string [, int $flags = ENT_COMPAT
    | ENT_HTML401 [, string $encoding =
    ini_get("default_charset") [, bool $double_encode = TRUE
    ]]] )
```

Documentation officielle pour `htmlentities()`

L'explication tout aussi simple que pour la fonction précédente: celle-ci convertit tous les caractères possibles en entités HTML.

Ce qu'il faut surtout noter, c'est qu'ici aussi **les guillemets simples n'étaient pas modifiés quand cette fonction était utilisée avec un seul argument**. Depuis PHP 8.1.0, la même modification a été apportée, mais là aussi, cette fonction n'est pas à utiliser de la même manière que présenté au début de cet article.

En fait, en comparant les deux méthodes, on peut voir qu'elles ont exactement la même signature[1]. Seul leur fonctionnement interne change un peu, en cela que la seconde va convertir plus de choses que la première.

1.0.1. Pourquoi ces deux fonctions existent-elles ?

C'est en bonne partie dû à la manière dont fonctionne le langage HTML. Il utilise des balises délimitées par des `<` et des `>`, avec ce qu'on appelle des attributs dont la valeur peut contenir des espaces, et du coup ces valeurs d'attribut doivent être encadrées par des guillemets.

1. On parle de signature pour désigner la liste ordonnée des arguments, avec leurs types

```
<input class="tata yoyo" type="hidden" name="chapeau">
<!-- |      ||||  |         |                              |
     |      |   |         |                             Fin de
   balise
     |      |     └──────── Délimiteurs de valeur d'attribut
     |      |
     |    Attribut
     |
   Début de balise
-->
```

Listing 1 – Un exemple de balise HTML (il aurait fallu un grand—comprennent qui pourront)

Maintenant, PHP est utilisé pour traiter dynamiquement des données. Comment éviter que des balises viennent poser problème quand elles sont reprises d'une source externe au code? Hé bien pour le HTML, on change la manière de représenter les caractères "dangereux" en les remplaçant par un groupe de caractères, les fameuses «entités HTML». D'où `htmlspecialchars()`, dont l'utilisation permet d'éviter que du code HTML non souhaité soit pris en compte par le navigateur .

La "petite sœur" `htmlentities()` a été créé du fait des problèmes de représentation des caractères accentués (entre autres) qui, dans le standard initial (ASCII) , n'existaient pas. On a du coup utilisé des entités HTML pour représenter ces "nouveaux" caractères, et fourni une autre fonction qui non seulement ferait le même travail que sa "grande sœur", mais gérerait les caractères n'étant pas dans le standard aussi.

Pour le coup, ces deux méthodes n'ont donc pas de lien avec les bases de données, mais bien avec le HTML uniquement. Si `htmlspecialchars()` est encore recommandée quand on affiche des valeurs qu'on ne sait pas sûres (**protection contre les failles XSS**), `htmlentities()` n'est plus couramment nécessaire, dans la mesure ou les caractères exotiques ne posent plus de problème de nos jours.

? Quel est le rapport avec les injections SQL, alors?

1.1. Comparaison HTML et SQL

Une des premières choses qui vient à l'esprit quand on parle des injections SQL, c'est le souci avec les chaînes de caractères et les guillemets. C'est une des possibilités les plus simples à expliquer, mais ce n'est néanmoins pas la seule, loin de là .

Avant de préciser cela, j'aimerais rappeler quelques éléments de syntaxe du SQL.

```sql
'Entre deux guillemets, il y a une chaîne de caractères'; -- ce
    qui suit deux tirets est un commentaire SQL
-- (jusqu'à la fin de la ligne)
SELECT * /* ceci est aussi un commentaire, mais plus circonscrit
    */ FROM matable;
-- Et un point-virgule ; délimite deux instructions/requêtes SQL
```

Listing 2 – Condensé de syntaxe SQL

Maintenant que la syntaxe a été brièvement présentée, il est facile de remarquer que **la syntaxe du SQL n'a pas de similitude avec le HTML**, si ce n'est l'utilisation des guillemets pour délimiter les chaînes de caractères. Comme il peut y avoir des guillemets légitimes dans celles-ci, il faut là aussi faire en sorte que ça ne détruise pas les requêtes. Mais le processus pour éviter cela n'est pas le même qu'en HTML.
Le plus ennuyeux, c'est que contrairement au HTML, il y a plusieurs SQL: MySQL en a fait une variante largement utilisée, mais PostgreSQL et Microsoft ont les leurs. Et qui dit plusieurs SQL, dit plusieurs syntaxes.
Si celles-ci ont des bases communes, les différences sont cependant rapidement problématiques quand vient le moment de sécuriser tout cela. Donc le SQL, suivant la base de données que vous utilisez, devra être sécurisé de manière adaptée. Les guillemets devront être doublés ou précédés du caractère `\` selon le moteur de bases de données…
La comparaison entre HTML et SQL ne tient pas la route, on le sent dès le début. Néanmoins, c'est un peu ce qui est fait quand on mélange les deux types de failles!
Prenons un autre exemple dont j'ai entendu parler par ici. Vous achetez un super outil de bricolage à l'étranger. Et quand vous souhaitez l'utiliser chez vous, vous constatez que la prise n'est pas compatible. Impossible de changer l'article, et vous en avez vraiment besoin… Utiliser un adaptateur ne vous satisferait pas, parce que devoir acheter un adaptateur juste pour cet outil, c'est inutile… Et puis les adaptateurs ne tiendraient pas avec ce que demande l'outil.
C'est là que vous vous dites que vos talents d'électricien vous serviront: vous allez dénuder la prise pour mettre une fiche de chez vous. Un peu de *bricolage*, et voilà de quoi brancher votre appareil directement sur la prise murale!
Mais pour ça, vous avez dû aller modifier les fils électriques. Là où auparavant vous aviez un branchement avec raccord sécurisé , même si inutilisable, vous avez désormais dû faire ce qu'on peut voir dans ce tutoriel pour avoir, au final, quelque chose qui fonctionne, mais qui est **moins sécurisé** que la version initiale. Pourquoi ne pas avoir acheté régional, du coup?
Pour en revenir au sujet de l'article: si `htmlspecialchars()` et `htmlentities()` ont une incidence sur les injections SQL, ce n'est qu'une coïncidence qui empêche une des possibilités. Autrement, **`htmlspecialchars()` et `htmlentities()` ne servent à rien pour «sécuriser une requête»**, vu que cette sécurisation est dépendante de la base de données.

? Mais enfin, si j'utilise une de ces fonctions sur les données à enregistrer, je n'ai plus besoin de le faire à l'affichage! Qu'est-ce que tu tentes de nous faire gober, hein?

2. En quoi une telle utilisation de ces fonctions est un problème

Jusqu'à présent, vous vous êtes cantonné aux sites internet "simples", qui utilisent du HTML, et cet article vous fait ricaner. Voici cependant plusieurs cas précis où des problèmes surviennent.

2.1. En cas d'utilisation des données pour autre chose que le site

Imaginons qu'un beau jour, vous voudrez générer d'autres choses que du HTML depuis vos bases de données, disons que vous devez créer une API pour une application mobile native. Pas de bol, ces applications ne sont pas des pages web, donc les "sécurisations" que vous avez enregistrées doivent être "levées" pour que les mobiles puissent utiliser correctement les données. Deux choix s'offrent à vous: utiliser `html_entities_decode()` sur votre serveur pour envoyer

des données au bon format, ou effectuer ce traitement dans l'application même, donc chez les clients.
Dans le premier cas, votre serveur devra supporter la charge, et si l'application a du succès, ça risque de coûter cher en ressources, comme vous "sécurisez" ***et*** "dé-sécurisez" sur la même machine. Deux opérations qui s'annulent, mais qui doivent être faites.
Dans l'autre cas, même si les performances des mobiles s'améliorent avec le temps, le traitement dans l'application aura un impact.
Dans les deux cas, ça restera un point d'optimisation normalement facile mais complètement bloqué, tout ça parce que vos données en base sont trop liées à l'utilisation initiale en HTML. On voulait «*gagner du temps*» au départ, on voulait «*ne pas oublier de le faire à l'affichage*», ou «*faire d'une pierre deux coups*», voire même «*faire pro*»—et là, c'est totalement raté.

i

A ce dernier propos, je vous conseille d'aller fouiner dans le code d'Eloquent (qui gère les bases de données dans le framework Laravel), ou de Doctrine (qui est le composant choisi pour s'occuper des bases de données avec Symfony), ou encore de Propel 2 (qui est une troisième solution effectuant le même genre de travail, initialement proposée avec Symfony 1), ou de toute autre couche d'abstraction de base de données ou même d'ORM qui se base sur un gestionnaire de bases de données SQL—et qui soit encore maintenu.

2.2. En cas de réception depuis une autre interface que votre code

Vous constatez une petite erreur dans certaines informations HTML en base de données, erreurs que vous souhaitez modifier rapidement. Comme vous avez vu cela alors que vous vous baladiez avec phpMyAdmin, vous effectuez la modification par ce biais. Si pour une application mobile cela ne pose pas nécessairement de problème, vous remarquez que pour les pages où ces modifications sont affichées, ça ne passe pas. Hé oui, phpMyAdmin n'utilise pas `htmlspecialchars()` sur les données qu'il fait insérer en base… Et pourtant, c'est un outil aussi largement utilisé que ceux listés précédemment, et bien plus vieux!

2.3. En cas de formulaire de recherches

Le cas de `htmlentities()`, lui, pose un problème plus vaste. Si vous l'utilisez pour enregistrer vos données et que vous voulez soudainement y effectuer des recherches, vous allez perdre beaucoup de pertinence. En effet, beaucoup n'utilisent plus les lettres accentuées dans les champs de recherche. Seulement, les données dans lesquelles on recherche *ont* les accents. Comment faire comprendre qu'une recherche sur `ecrire` doit faire sortir `&eacute;crire`? On me dira qu'on peut enlever les accents avant d'utiliser `htmlentities()`. Je répondrai que s'il n'y avait pas les accents ici, on aurait durement condamné ce manquement à l'orthographe du français, sans parler de l'équipe de validation qui aurait refusé la publication.
Quant à proposer d'utiliser `html_entities_decode()`, c'est le même problème que mentionné pour l'utilisation des informations en base sur un autre support qu'un site web. D'autant plus que vous ne décoderiez pas uniquement ce qui sera affiché, mais aussi tout ce dans quoi il faut chercher…

!

Je ne dis pas qu'il faut absolument effectuer les recherches avec les accents, seulement que c'est un non-sens de devoir les enlever pour faire fonctionner une recherche qui ne

2. *En quoi une telle utilisation de ces fonctions est un problème*

retournerait pas les mêmes résultats s'ils sont ou non présents.

2.4. En cas de classement par ordre alphabétique

Remplacer des caractères par d'autres a une influence sur le tri. Prenons une liste de mots que l'on souhaite afficher par ordre alphabétique dans une page web:

- assurance
- croître
- énerver
- assurément
- tester
- sur
- enerver
- croissance

Il est facile de déterminer le résultat du tri.

1. assurance
2. assurément
3. croissance
4. croître
5. enerver
6. énerver
7. sur
8. tester

Maintenant, appliquons `htmlentities()` sur la liste initiale, et affichons le résultat du tri..

1. énerver
2. assurément
3. assurance
4. croître
5. croissance
6. enerver
7. sur
8. tester

Donc `e` est avant `a` du fait des lignes 1 et 2 insi que 2 et 3. Alors pourquoi «enerver» est à la ligne 6 et non avant la 2? De manière similaire, pourquoi est-ce que `t` est avant `s` aux lignes 4 et 5, mais pas aux lignes 7 et 8? Il est facile de comprendre que ce sont les accents qui posent problème, et si on va regarder la version "brute" de la dernière liste, on a ceci:

```
1 <ol>
2 <li>&eacute;nerver</li>
3 <li>assur&eacute;ment</li>
4 <li>assurance</li>
```

2. En quoi une telle utilisation de ces fonctions est un problème

```
<li>cro&icirc;tre</li>
<li>croissance</li>
<li>enerver</li>
<li>sur</li>
<li>tester</li>
</ol>
```

On constate aisément que ce sont les caractères spéciaux dûs aux entités HTML qui causent ces aberrations dans le tri. Expliquez cela aux nombreux utilisateurs de votre site.
Pour ceux qui pensent qu'utiliser `htmlspecialchars()` uniquement ne pose aucun problème dans ce cas, voici de quoi vous prouver le contraire.

```php
<?php
$greaterThan = '>';
$lessThan = '<';
var_dump($lessThan < $greaterThan); // boolean true
$greaterThan = htmlspecialchars($greaterThan);
$lessThan = htmlspecialchars($lessThan);
var_dump($lessThan < $greaterThan); // boolean false
```

2.5. Au niveau de l'espace de stockage et de la bande passante

Dernier point que j'aimerais soulever: si tous les caractères qui ne sont pas dans le standard ASCII sont convertis en entités HTML, il faut bien se rendre compte que ça prend plus d'espace de stockage. Vu que ces données doivent être envoyées, ça demande soit plus de bande passante, soit plus de temps pour que le client reçoive ce dont il a besoin, et dans tous les cas plus de RAM chez le client.
Imaginons un site avec une gigantesque base de données contenant des informations dans une multitude de langues, Wikipedia ou Facebook. Si notre alphabet pose peu de soucis parce que les lettres qui le composent sont en grande partie dans l'ASCII, ce n'est pas le cas des caractères chinois, japonais, tamouls ou encore sanscrits. Ils ont pratiquement tous une représentation sous forme d'entité HTML, mais une entité HTML représente plus de caractères qu'un caractère de cette langue en lui-même. Un exemple: devient `&#x6C49;&#x5B57;`, on passe de 2 à 16 caractères, une augmentation d'un facteur 8 dans ce cas[1], donc autant de données à stocker sur le serveur, à envoyer, et à mettre dans la RAM de l'appareil cible. De là à dire que pour gérer des données en chinois il faut multiplier les ressources par 8 par rapport à l'anglais, il n'y a qu'un pas. Quant aux smileys, n'en parlons pas: ils sont presque tous représentés sous la forme `&#x1F###;`, donc un facteur 9.
Et je ne mentionnerai plus les soucis avec les recherches.
Au final, on peut raccourcir tout ça en disant que **le contenu d'une base de données ne doit pas être lié à une utilisation**. Avoir des entités HTML est déjà une forme de liaison à une utilisation, pour un site web en HTML en l'occurrence. Dès qu'il faut ajouter une étape de correction pour utiliser les valeurs, il y a très probablement un souci.

1. L'exemple donné est volontairement simplifié. Plus d'informations dans cette partie et la suivante du tutoriel sur les encodages.

> Je parle bien ici de *correction*, dans notre cas de devoir utiliser `html_entities_decode()` pour défaire quelque chose qu'on a fait et qui devient gênant.

Autant le dire tout net: dans un environnement professionnel, choisir d'utiliser `htmlspecialchars()` avant d'envoyer en base des données serait *passible d'un blâme*. Quant à faire de même avec `htmlentities()`, cela pourrait tout à fait **coûter une place de travail**.

3. Comment s'utilisent ces fonctions

Qu'on soit d'accord, il ne faut pas complètement mettre de côté `htmlspecialchars()`, la sécurité que cette fonction apporte est *réelle* et *nécessaire*. Seulement, c'est au moment où les données sont utilisées dans du HTML qu'elles peuvent poser problème, donc *quand on les affiche*. En conséquence, **`htmlspecialchars()` s'utilise après `echo`, `<?=` ou encore `print`**. Pas besoin de "pré-traiter" et de s'encombrer de tous les problèmes mentionnés plus haut. Quant à `htmlentities()`, cette fonction n'a plus vraiment de raison d'être utilisée de nos jours, sauf dans des cas très particuliers qui mettront bien plus à l'épreuve votre carrière de développeur web avec PHP. 🍊
Et pour sécuriser la requête, il faudrait dans l'idéal utiliser les requêtes préparées. C'est une technique qui permet d'écrire les requêtes sans mettre les valeurs (et uniquement les valeurs, pas les noms de champs ou de tables) définitives, de la faire reconnaître et pré-valider syntaxiquement par le moteur de bases de données, puis de l'utiliser en lui fournissant les données définitives. L'avantage étant que c'est bien le moteur de base de données qui va sécuriser ces informations, et donc sait exactement comment le faire correctement. Le principe est expliqué dans ce tutoriel qui, même s'il est pour MySQL, fonctionne pour les autres SQL. Les outils d'accès aux bases de données supportent très largement cette fonctionnalité, permettant d'éviter une syntaxe SQL qui, là aussi, dépend du moteur de bases de données utilisé.
Dans certains cas, il n'est pas possible d'utiliser une requête préparée. Les outils d'accès aux bases de données ont des fonctions ou des méthodes qui permettent de sécuriser "manuellement" les données, entre autres la méthode PDO::quote et la fonction `mysqli_real_escape_string()` pour PHP.
Ces méthodes de sécurisation ainsi que plusieurs points de vérification supplémentaires sont présentées plus en détail dans ce tutoriel sur les failles par injection SQL, qui expliquera plus à fond les risques encourus.

3.1. Hé, et en quoi la « variante 2 » est plus préoccupante ?

Oups, j'ai failli oublier! 😇
En fait, si vous avez lu tout ce qu'il y a ci-dessus, je pense que vous pouvez imaginer pourquoi. C'est que d'une part, on "sécurise" avec `htmlspecialchars()`, et d'autre part, on effectue une requête préparée, qui sécurise aussi. Mais avec tout l'argumentaire ci-dessus, vous savez clairement que ce sont deux sécurisations distinctes, et l'une d'entre elles n'est pas bien placée. Seulement avant cela, est-ce qu'il était clair que c'était deux choses distinctes? Et surtout en quoi? C'est précisément ***là*** que c'est plus préoccupant: en les utilisant ainsi, on finit par ne plus faire la distinction. Mais faire les deux en ne pensant qu'aux failles XSS ou qu'à celles par injection SQL montre aussi que la distinction entre les deux n'est pas claire. Sinon, pourquoi utiliser deux méthodes? Pourquoi *devoir* utiliser deux méthodes pour la même chose, depuis le

temps que ces failles existent? Logiquement, c'est parce que ce sont bien deux cas d'utilisations différents.

Conclusion

Pour conclure, voici une suite possible de la discussion hypothétique qui a ouvert cette publication.

> —[**Personne 2**] Sécuriser quoi exactement ? Si c'est ta requête, ce n'est pas la bonne fonction.
> —[**Personne 1**] Ben j'ai testé et ça marche.
> —[**Personne 2**] D'accord, tu peux me dire ce que fait la fonction `mysqli_real_escape_string()` ?
> —[**Personne 1**] Elle sécurise les données pour pouvoir les utiliser dans une requête.
> —[**Personne 2**] Tu arrives à me dire ça, mais tu utilises quand-même une méthode qui fait autre chose, et tu me dis en plus que cette autre chose est ce qu'il te faut ?!
>
> *Suite d'une conversation presque fictive au détour d'un forum*

J'espère que vous aurez remarqué la contradiction contenue dans cette discussion imaginaire ! 🍊
Le point le plus surprenant là est que finalement, l'erreur est d'utiliser des outils pour deux langages différents comme s'ils étaient comparables. On l'a bien vu, ce n'est pas le cas.

En conséquence :

- `htmlspecialchars()` s'utilise **à l'affichage dans du HTML** (après `echo`, `print`, `<?=`), comme l'indique la partie `html` dans le nom de la fonction, afin de protéger **contre les failles XSS** ;
- les requêtes préparées permettent de sécuriser une très grande majorité des requêtes **contre les injections SQL**. Pour quelques cas plus gênants, il y a en PHP des méthodes comme PDO::quote et `mysqli_real_escape_string()`.

Donc si vous vous trouvez dans un des cas suivants :

- vous trouvez un tutoriel qui pratique l'utilisation de `htmlentities()` ou `htmlspecialchars()` sur des données qui sont envoyées en base ;
- quelqu'un vous parle très sérieusement d'utiliser une de ces deux fonctions pour sécuriser vos requêtes ;

Je me permettrai de vous remettre en tête cette réplique culte du premier film d'une trilogie sur un anneau :

4. Fuyez, pauvres fous !

Et dans le second cas, si vous ne vous sentez pas de force à lui faire comprendre la différence entre les failles par injection SQL et XSS, proposez-lui de lire sinon cette publication, au moins celles ci-dessous :

- le tutoriel sur les injections SQL
- l'article sur les failles XSS

Si vous souhaitez aller plus loin, renseignez-vous sur le principe *filter input, escape output* (en

français, ça donnerait «filtrer les entrées, échapper les sorties»), qui explique la bonne pratique qui n'est pas mise en place dans l'exemple (qui ferait plutôt *escape input* tout simplement).