Sécuriser les sessions en PHP

par Adrien Pellegrini (Page d'accueil)

Date de publication : 1/03/2008

Dernière mise à jour :

Les sessions étant très utilisées de nos jours, il ne faut jamais négliger le coté sécurité. Cet article expose les différentes attaques possibles vis à vis des sessions et comment s'en protéger efficacement.

I - Introduction

I.1 - Pourquoi ?

Il existe divers types d'attaques qui permettent de récupérer l'identifiant de session et de récupérer des informations sensibles. Grâce à ces informations une personne malveillante pourra, par exemple, usurper votre compte.

Ces différents types d'attaques sont :

1 - Session Exposure

2- Session Fixation

3 - Session Sniffing

4 - Session Prediction

5 - Session Hijacking

6 - Session Poisoning

7 - Session Injection

Comme vous pouvez le constater nous avons l'embarra du choix, ce n'est vraiment pas ce qui manque.

I.2 - Comment ?

Pour se protéger de ces types d'attaques, il n'y à pas grand chose à faire. Les problèmes se résolvent assez rapidement avec des solutions plus ou moins simples.

Pour la plupart des attaques, je vous énoncerais une ou plusieurs solutions possibles à mettre en oeuvre et je mettrais en application une ou plusieurs d'entre elles. Une solution protège généralement de plusieurs attaques ce qui réduit assez bien le code pour parer à la plupart des éventualités.

Mais ayez toujours à l'esprit qu'il est impossible de se protéger complètement de tout types d'attaque. Que certaines solutions sont assez lourdes à mettre en place car elle allonge considérablement le temps d'exécution de la page. Que d'autres bien qu'elles soient utiles, excluent certains groupes de webmasters (pas de possibilité de configuration de leur serveur car ils se trouvent sur un mutualisé) ou excluent certain groupes d'utilisateurs (ceux qui auraitent désactivés les cookies).

I.3 - Requis

- PHP 5

*- Extension **PHP Data Objects - PDO***

- *Extension **Chiffrement mcrypt***

- *Quelques notions de programmation orienté objet en PHP.*

II - Les attaques

Tout au long de ce chapitre, nous allons construire petit à petit des classes qui permettront de nous protéger de la plupart des attaques.

Ces classes nous donnerons la possibilité de gérer les sessions soit via une base de données soit via le système de fichiers. Seul le code pour la gestion par base de données sera affiché ici. Pour voir celui concernant la gestion via fichiers, rendez-vous en fin d'article et téléchargez les sources.

II.1 - Session Exposure

II.1.1 - Problème

Cette vulnérabilité est étroitement liée au hébergement mutualisé. Dans ce type d'hébergement, un serveur est partagé entre plusieurs sites. Toutefois, PHP **enregistre** les sessions dans le **même répertoire** (/tmp par défaut). PHP crée un fichier de session par session utilisateur nommé par l'identifiant de session. Ces fichiers sont de bêtes fichiers texte qu'il suffit d'ouvrir avec votre éditeur de texte favori et contiennent les variables de session **en texte clair** !

Imaginez alors que l'administrateur du serveur n'a pas bien fait son travail et qu'un utilisateur puisse remonter dans l'arborescence des répertoires. Il aurait alors accès à toutes les sessions des sites.

II.1.2 - Solutions

Solution 1 :

Vous faite **confiance** à votre **administrateur serveurs**.

Solution 2 :

Vous **changer** le **répertoire** de stockage des sessions. Cette solutions n'est que très rarement envisageable sur un serveur mutualisé car vous n'avez pas accès au php.ini. Il faut pour cela changer la directive de configuration ***session.save_path***. Et même si vous pouviez changer le répertoire, il faut encore s'assurer que personne n'y aura accès.

Solution 3 :

Vous **sauvegardez** vos sessions dans une **base de données**. C'est un peu plus lourd à mettre en oeuvre que les autres solutions mais c'est aussi la meilleures des protections pour ce genre d'attaques.

C'est donc cette dernière solution qui va être envisagée ici.

Un autre problème mineure survient alors. Si on vous volait votre login/mot de passe de votre compte pour accéder à la base de données, cet utilisateur pourrait aussi voir les variables de sessions. C'est pour cela que j'ai décidé tout simplement de crypter en plus les données.

II.1.3 - Code

Pour pouvoir stocker les variables de session dans la base de données ou dans des fichiers , il faut bien évidemment utiliser l'handler de session via la fonction ***session_set_save_handler()*** et dans notre cas avoir une base de données !

Initialement cette base de données sera construite comme ceci :

sql

```
CREATE TABLE "sessions" (
  "session_id" varchar(32) NOT NULL,
  "session_data" text NOT NULL,
  "session_time" datetime NOT NULL,
  PRIMARY KEY  ("session_id")
);
```

Passons maintenant à la création des méthodes pour l'handler de session.

Pour faire les choses proprement, j'ai créé une interface qui contiendra les prototype des méthode nécessaire pour l'handler de session, c'est à dire les méthodes *_open()*, *_close()*, *_read()*, *_write()*, *_destroy()* et *_gc()* ainsi qu'une méthode qui vérifiera des données par la suite.

Yume_Session_Interface - Yume/Session/Interface.php

```
<?php

interface Yume_Session_Interface
{
 public function _open($save_path, $name);

 public function _close();

 public function _read($id);

 public function _write($id, $data);

 public function _destroy($id);

 public function _gc($maxLifeTime);
}

?>
```

Ensuite, j'ai créé une classe de base abstraite. Cette classe va définir les méthodes de base nécessaire pour sauvegarder les sessions dans une base de données ou dans des fichiers ainsi que la fonction de cryptage des données.

Le cryptage par défaut sera le Blowfish en mode CBC via les fonctions fournies grâce à l'extension **mcrypt**. Tout cryptage est le bienvenu du moment qu'il soit réversible.


*Merci de **changer** obligatoirement le **grain de sel** !*

Parmi ses méthodes, nous trouverons quelques unes que voici :

- une méthode pour définir les paramètres de sécurité tels que l'algorithme de cryptage, son mode et le grain de sel (voir *setCryptOptions()*).

- une méthode pour définir les paramètres de la ressource utilisée pour sauvegarder les sessions (voir *setRessourceOptions()*).

- une méthode pour démarrer la session (voir *start()*).

- une méthode pour crypter les données (voir *mycrypt_encrypt()*).

- une méthode pour décrypter les données (voir *mycrypt_decrypt()*).

Yume_Session_Abstract - Yume/Session/Abstract.php

```php
<?php

require_once 'Interface.php';

abstract class Yume_Session_Abstract implements Yume_Session_Interface
{
 protected $_regenerateTime = 10;

 public function start($params = false)
 {
  // Récupère les paramètre et les traîte
  if ($params)
  {
   $this->setRessourceOptions($params);
  }

  // Vérifie les paramètre
  if ($this->_ressourceOptions['type'] == 'database')
  {
   if (empty($this->_ressourceOptions['hostname']) ||
!is_string($this->_ressourceOptions['hostname']))
   {
    throw new Exception('Unknown database host.');
   }
   elseif (empty($this->_ressourceOptions['dbname']) ||
!is_string($this->_ressourceOptions['dbname']))
   {
    throw new Exception('Unknown database name.');
   }
   elseif (empty($this->_ressourceOptions['username']) ||
!is_string($this->_ressourceOptions['username']))
   {
    throw new Exception('Unknown database username.');
   }
   elseif (!is_string($this->_ressourceOptions['password']))
   {
    throw new Exception('Unknown database password.');
   }
  }
  elseif ($this->_ressourceOptions['type'] == 'file')
  {
   if (isset($this->_ressourceOptions['save_path']) &&
is_string($this->_ressourceOptions['save_path']) == false)
   {
    throw new Exception('Invalid save path.');
   }
   else if (empty($this->_ressourceOptions['save_path']))
   {
    $this->_ressourceOptions['save_path'] = session_save_path();
   }
```

Yume_Session_Abstract - Yume/Session/Abstract.php

```
    }

    // Configure l'handler de session
    if (!session_set_save_handler(
      array($this, '_open'),
      array($this, '_close'),
      array($this, '_read'),
      array($this, '_write'),
      array($this, '_destroy'),
      array($this, '_gc')
     ))
    {
     throw new Exception('Session handler error.');
    }

    // Récupère le maxlifetime
    $this->_maxLifeTime = ini_get('session.gc_maxlifetime');

    // Enregistre l'heure de début de la session
    $this->_time = date('Y-m-d H-i-s', time());

    // Démarre la session
    session_start();

    return $this;
   }

   public function setRessourceOptions(array $options)
   {
    foreach ($options as $option => $value)
    {
     $this->_ressourceOptions[$option] = $value;
    }

    return $this;
   }

   public function setCryptOptions($salt, $cypher = MCRYPT_BLOWFISH, $mode = MCRYPT_MODE_CBC)
   {
    $this->_cryptCypher = $cypher;
    $this->_cryptMode =  $mode;
    $this->_cryptSalt = $salt;

    return $this;
   }

   protected function mycrypt_encrypt($cryptCypher, $cryptMode, $cryptSalt, $data)
   {
    $iv_size = mcrypt_get_iv_size($cryptCypher, $cryptMode);
    $iv = mcrypt_create_iv($iv_size, MCRYPT_RAND);

          $data = mcrypt_encrypt($cryptCypher, $cryptSalt, $data, $cryptMode, $iv);

          $data = base64_encode($iv.$data);

    return $data;
   }

   protected function mycrypt_decrypt($cryptCypher, $cryptMode, $cryptSalt, $data)
   {
          $data = base64_decode($data);

          $iv_size = mcrypt_get_iv_size($cryptCypher, $cryptMode);
          $iv = substr($data, 0, $iv_size);

          $data = substr($data, $iv_size);
          $data = mcrypt_decrypt($cryptCypher, $cryptSalt, $data, $cryptMode, $iv);
```

Yume_Session_Abstract - Yume/Session/Abstract.php

```
    return $data;
  }
}

?>
```

Voici maintenant la classe qui va gérer les sessions. Elle redéfinira les méthodes *_open*, *_close*, *_read*, *_write*, *_destroy* et *_gc*.

Avant de vous montrer le code, il est bon d'en savoir un minimum sur ces fonction utilisés par l'handler de session.

_open($save_path, $name):

Cette méthode à pour but d'initialiser les ressources que vous allez utiliser pour stocker les sessions. Cette méthode prend deux arguments qui sont le chemin vers le répertoire où stocker les sessions et le nom des sessions.

Dans notre cas ni l'un, ni l'autre n'a d'utilité, ni la méthodes elle-même car la connexion à la base de donnée est effectuée ailleurs dans la classe.

_close():

Cette méthode est utilisée pour fermer les ressources. Elle ne sera pas non plus utilisée dans notre cas.

_read($id):

Comme son nom l'indique cette méthode sert à lire les données contenu dans la session. Elle prend comme paramètre l'identifiant de session.

Cette méthode doit **impérativement** retourner une chaîne de caractères, même si celle-ci est vide !

_write($id, $data):

Cette méthode permet d'écrire des données dans le contenu d'une session. Elle prend comme paramètre l'identifiant de sessin et bien entendu les données à écrire.

_destroy($id):

Cette méthode supprime la session. Elle prend comme paramètre l'identifiant de session.

_gc($maxLifeTime):

Cette dernière méthode joue le rôle de ramasse miette (ou plus couramment appelé "*garbage collector*").

Chaque fois qu'une session est ouverte, la probabilité que cette méthode est exécutée dépend d'un petit calcul : *session.gc_probability*/*session.gc_divisor*. Un nombre aléatoire est alors généré et si celui-ci est inférieur au calcul, la méthode est exécutée. Par exemple 1/100 veut dire qu'il y a 1% de chance que le garbage collector soit exécuté à chaque requête.

La méthode efface alors toutes les sessions qui auraient un âge supérieur à la variable passée en paramètre.

Yume_Session_Abstract - Yume/Session/Database.php

```php
<?php

require_once 'Interface.php';

class Yume_Session_Database extends Yume_Session_Abstract
{
 protected $_ressource          = null;
 protected $_ressourceOptions   = array();

 protected $_time               = 0;
 protected $_maxLifeTime        = 3600;

 protected $_cryptCypher        = MCRYPT_BLOWFISH;
 protected $_cryptMode          = MCRYPT_MODE_CBC;
 protected $_cryptSalt          = 'Vive Developpez.com !';

 public function _open($save_path, $name)
 {
  if ($this->_ressource == null)
  {
   $this->_ressource = new PDO
   (

 'mysql:host='.$this->_ressourceOptions['hostname'].';dbname='.$this->_ressourceOptions['dbname'],
    $this->_ressourceOptions['username'],
    $this->_ressourceOptions['password']
   );

   $this->_ressource->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
  }
  return true;
 }

 public function _close()
 {
  return true;
 }

 public function _read($id)
 {
  // Récupère les données
  $sql = $this->_ressource->prepare("
   SELECT session_data
   FROM sessions
   WHERE session_id = :id
  ");

  $sql->bindParam(':id', $id);
  $sql->execute();

  $data = $sql->fetch(PDO::FETCH_ASSOC);
  $data = $data['session_data'];

  if (empty($data))
  {
```

Yume_Session_Abstract - Yume/Session/Database.php

```
        $data = '';
       }
       else
       {
        // Decrypte les données
        $data = base64_decode($data);

        $iv_size = mcrypt_get_iv_size($this->_cryptCypher, $this->_cryptMode);
        $iv = substr($data, 0, $iv_size);
        $data = substr($data, $iv_size);
        $data = mcrypt_decrypt($this->_cryptCypher, $this->_cryptSalt, $data, $this->_cryptMode, $iv);
       }

       return $data;
      }

      public function _write($id, $data)
      {
       // Crypte les données
       $data = $this->mycrypt_encrypt($this->_cryptCypher, $this->_cryptMode, $this->_cryptSalt, $data);

       // Met à jour la base de données
       $sql = $this->_ressource->prepare("
        UPDATE sessions
        SET session_data = :data,
         session_time = :time;
        WHERE session_id = :id
       ");

       $sql->bindParam(':id', $id);
       $sql->bindParam(':data', $data);
       $sql->bindParam(':time', $this->_time);
       $sql->execute();

       // Ou alors insère les données
       if ($sql->rowCount() == 0)
       {
        $sql = $this->_ressource->prepare("
         INSERT INTO sessions (session_id, session_data, session_time)
         VALUES (:id, :data, :time);
        ");

        $sql->bindParam(':id', $id);
        $sql->bindParam(':data', $data);
        $sql->bindParam(':time', $this->_time);
        $sql->execute();
       }

       return true;
      }

      public function _destroy($id)
      {
       $sql = $this->_ressource->prepare("
        DELETE FROM sessions
        WHERE session_id = :id
       ");

       $sql->bindParam(':id', $id);
       $sql->execute();

       return true;
      }

      public function _gc($maxLifeTime)
      {
       $sql = $this->_ressource->prepare("
```

Yume_Session_Abstract - Yume/Session/Database.php

```
      DELETE FROM sessions
      WHERE DATE_ADD(session_time, INTERVAL :maxlifetime SECOND) < NOW()
     ");

     $sql->bindParam(':maxlifetime', $this->_maxLifeTime);
     $sql->execute();

     return true;
    }
   }

   ?>
```

Pour finir voici la classe qui permet de créeer l'objet sessions.

Yume_Session - Yume/Session.php

```
   <?php

   require_once 'Session/File.php';
   require_once 'Session/Database.php';

   class Yume_Session
   {
    const SESSION_FILE = 'file';
    const SESSION_DATABASE = 'database';

    private function __construct() {}
    public function __destruct() {}

    public static function getSession($type = self::SESSION_FILE)
    {
     switch ($type)
     {
      case self::SESSION_FILE:
       $session = new Yume_Session_File;
       $session->setRessourceOptions(array('type' => self::SESSION_FILE));
       return $session;
      case self::SESSION_DATABASE:
       $session = new Yume_Session_Database;
       $session->setRessourceOptions(array('type' => self::SESSION_DATABASE));
       return $session;
     }
    }
   }

   ?>
```

II.2 - Session Fixation

II.2.1 - Problème

Pour qu'une session puisse transiter d'une page à l'autre, il faut évidemment sauvegarder l'identifiant de session quelque part. Généralement cet identifiant de session peut être sauvegardé de deux manières différentes.

Note : L'identifiant de session sera désormais noté SID.

La première consiste à mettre le SID dans l'URL via une variable GET nommée PHPSESSID par défaut.

Ceci est possible si la directive de configuration *session.use_trans_sid* est activée.

Cette directive activée peut poser quelques problèmes lorsqu'un utilisateurs non attentif, qui est connecté à son compte sur un tel site, copie-colle bêtement un lien contenant le SID. Dès lors l'utilisateur qui reçoit ce lien aura accès au compte comme bon lui semble.

La deuxième solution consiste à sauvegarder le SID dans un **cookie**.

Ceci est possible si la directive de configuration *session.use_cookies* est activée.

L'attaque dite "Session Fixation" consiste donc à faire utiliser à la victime un SID prédéfini par l'attaquant.

Avant de vous montrer un petit exemple pour vous aider à comprendre, il reste un point à éclaircir. C'est la restriction des serveurs vis à vis d'un SID passé manuellement dans l'URL.

Il existe deux grandes catégories de serveurs à ce niveaux :

1 - Les serveurs dit "**permissif**" : c'est à dire des serveurs qui autorisent un SID quelconque et qui créeront une nouvelle session avec cet SID.

2 - Les serveurs dit "**strict**" : c'est à dire des serveurs qui n'autorisent pas des SID externe, donc qui n'acceptent que les SID crées par lui-même.

PHP se trouvant dans la première catégorie.

Nous allons créer deux scripts PHP. L'un qui créera une variable de session et l'autre qui l'affichera.

page1.php

```php
<?php
error_reporting(E_ALL | E_STRICT);
date_default_timezone_set('Europe/Brussels');

require_once 'Yume/Session.php';

try
{
 $session = Yume_Session::getSession(Yume_Session::SESSION_DATABASE);
 $session->setRessourceOptions(array(
    'hostname' => 'localhost',
    'dbname' => 'session',
    'username' => 'root',
    'password' => ''
   ))
   ->setCryptOptions('Bouhhhhhhhh !')
   ->start();

} catch(Exception $e)
{
 echo $e->getMessage();
}

$session->test = 'oki';

?>
```

page2.php

```php
<?php
error_reporting(E_ALL | E_STRICT);
date_default_timezone_set('Europe/Brussels');

require_once 'Yume/Session.php';

try
{
 $session = Yume_Session::getSession(Yume_Session::SESSION_DATABASE);
 $session->setRessourceOptions(array(
    'hostname' => 'localhost',
    'dbname' => 'session',
    'username' => 'root',
    'password' => ''
   ))
   ->setCryptOptions('Bouhhhhhhhh !')
   ->start();

} catch(Exception $e)
{
 echo $e->getMessage();
}

echo $session->test;

?>
```

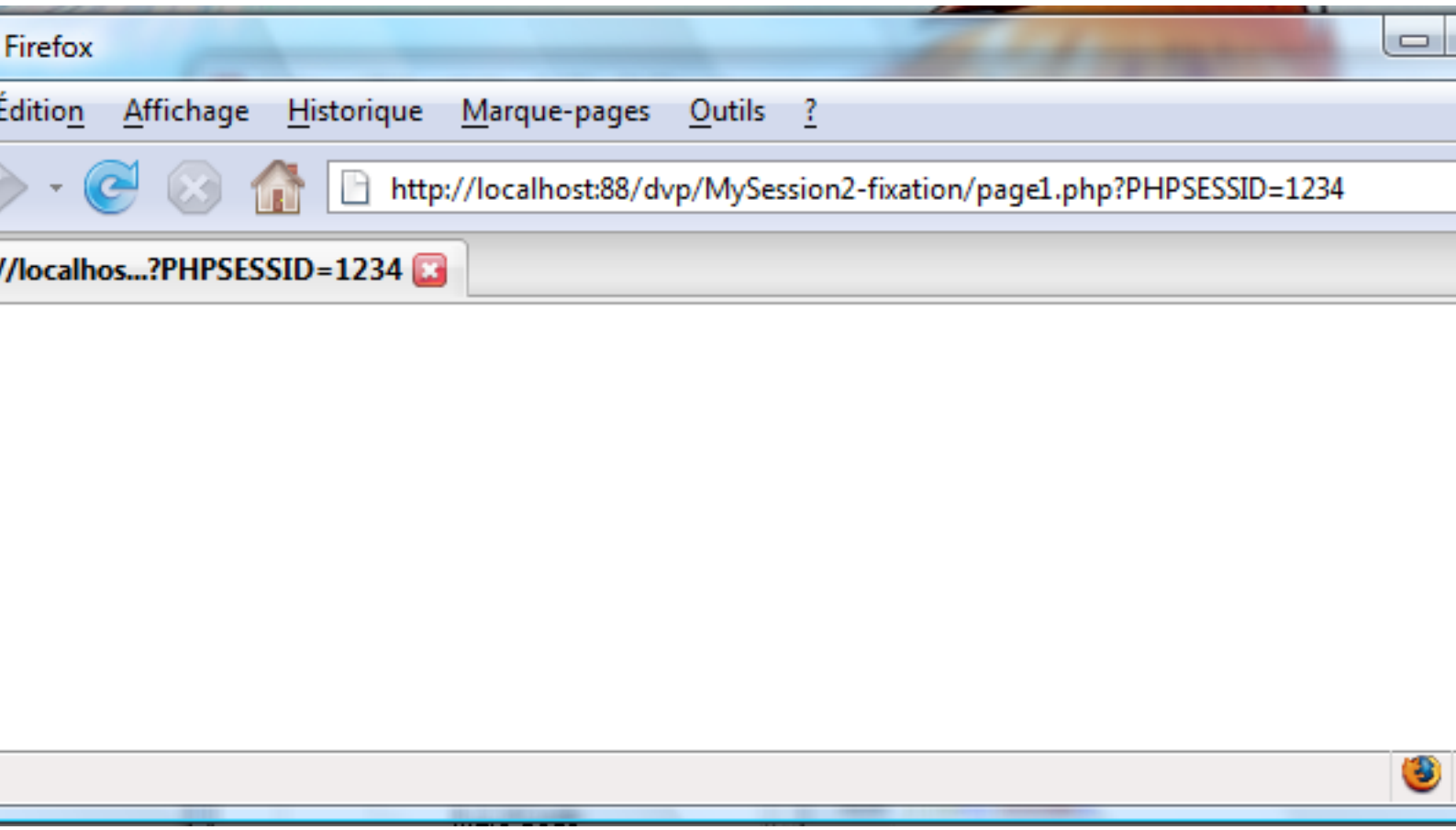
Tout d'abord on va choisir un SID simple pour les test qui sera 1234. Nous voila donc avec comme URL :

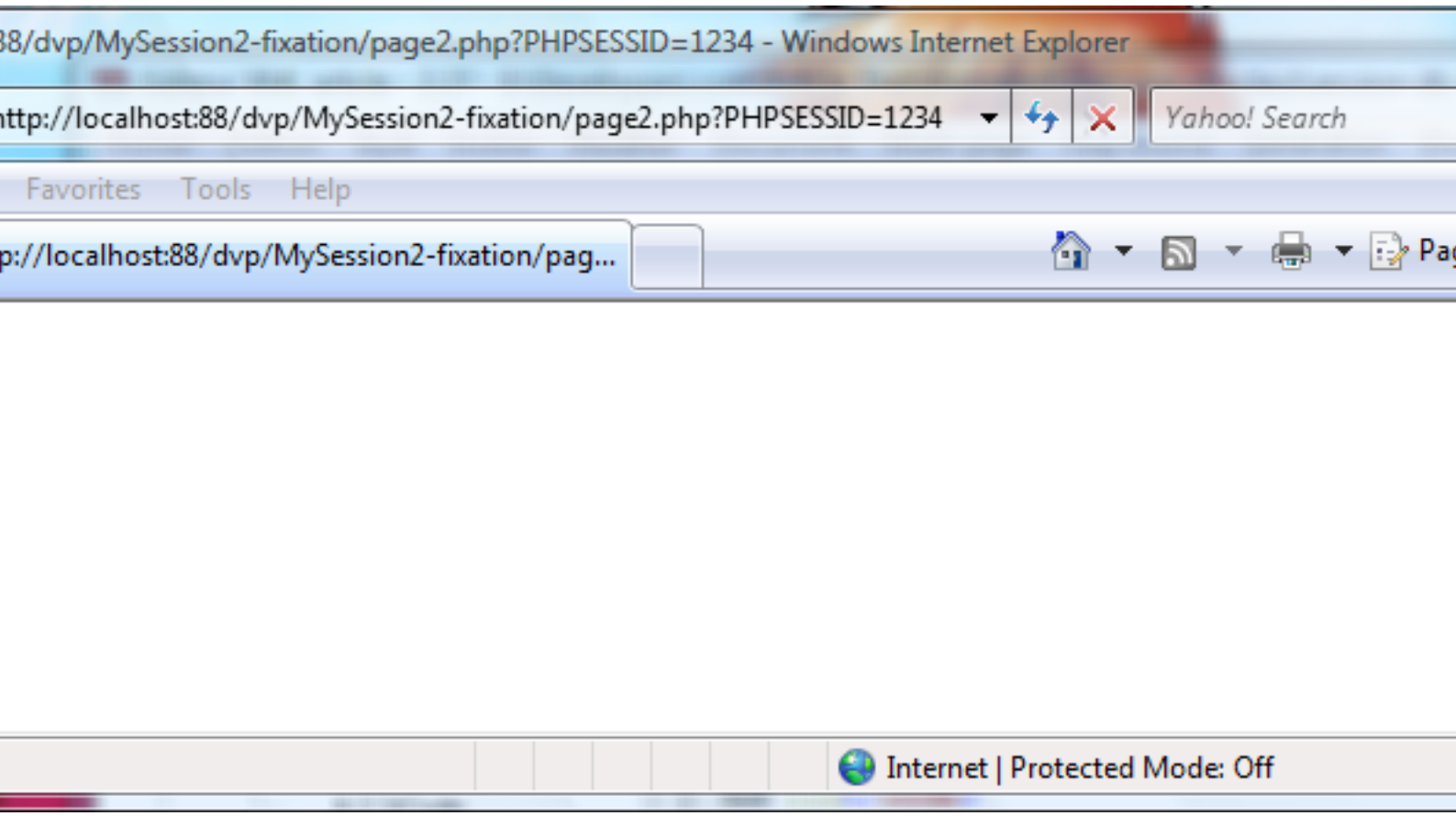
http://localhost:88/dvp/MySession2-fixation/page1.php?PHPSESSID=1234.

Maintenant, soit avec un autre navigateur, soit un autre ordinateur, vous allez visiter la page2.php avec l'URL :

http://localhost:88/dvp/MySession2-fixation/page2.php?PHPSESSID=1234.

Bien que la session n'est pas censée être démarrée sur la page2.php, vous constaterez que on obtient bien l'affichage de "oki".





II.2.2 - Solutions

Il existe plusieurs solutions à ce type d'attaque, toutes ont leurs avantages et inconvénients.

Pour toutes solutions :

session.use_trans_sid doit bien évidemment être à *0* quel que soit la solution adoptée (par défaut cette directive est désactivée).

Solution 1 :

La solution plus simple consiste sans doute à modifier deux directives de configuration qui sont :

- *session.use_cookies* à mettre à *1*

- *session.use_only_cookies* à mettre à *1*

Les inconvénients de cette solutions sont que si l'utilisateur n'accepte pas les cookies, il ne pourra pas démarrer une session et que vous devez avoir accès au fichier de configuration php.ini.

Solution 2 :

Une autre solution simple à mettre en place serait de stocké des informations et vérifier par la suite si elles ont ou pas changés. Car d'une page à l'autre il est peu probable qu'une informations choisie change. Mais quel information choisir ?

On pourrais stocker l'IP de l'utilisateur mais celle-ci est de loin fiable à 100%. Si l'utilisateur est derrière un proxy, il changera sans doute d'IP assez régulièrement. L'IP n'est donc pas à stocker.

Par contre, d'une page à l'autre, le naviguateur ne risque pas de changer. Nous pourrons donc utiliser cela.

Solution 3 :

Sans aucun doute, la meilleure des solutions est d'utiliser la fonction **session_regenerate_id()**.

Solution 4 :

Sans aucun doute, la plus sure des solutions est d'utiliser un serveur sécurisé SSL.

Ici, je ne vais pas utiliser une seule solution mais un mixte des solutions 2 et 3. La solution 1 ne dépendant pas du code mais de la configuration de votre serveur, il n'y aura donc rien de mis en plus pour celle la dans ce code.

II.2.3 - Code

Tout d'abord on va créér une méthode qui vérifiera les informations (voir *checkInfo()*).

Étant donné qu'il faut sauvegarder la naviguateur de l'utilisateur, il faudra créé un nouveau champ dans la base de données. Je l'ai nommé *session_browser*.

Pour la gestion de la régénération du SID, ca ne sert pas à grand chose de le regénérer à chaque page. J'ai donc décidé de le sauvegarder aussi dans la base de données. Je l'ai nommé *session_regenerate*. Pour la gestion à proprement parler, après un certain nombre de secondes, on regénérera le SID. Donc le champ dans la base de données stockera la date de la dernière regénération du SID. Quand on voudra définir une nouvelle variable de session ou bien en récupérer la valeur, on regardera si il faut regénéré le SID ou pas.

Méthode checkInfo() de la classe Yume_Session_Database - Yume/Session/Database.php

```
public function checkInfo()
{
 // Récupère l'identifiant de session
 $sessId = session_id();

 // Récupère le navigateur
 $userAgent = sha1($_SERVER['HTTP_USER_AGENT']);

 // Test si une session est sauvegardée
 $sql = $this->_ressource->prepare("
  SELECT session_id
  FROM sessions
  WHERE session_id = :id
 ");

 $sql->bindParam(':id', $sessId);
 $sql->execute();

 $result = $sql->fetchAll(PDO::FETCH_ASSOC);

 // Si elle ne l'est pas, on la crée
```

Méthode checkInfo() de la classe Yume_Session_Database - Yume/Session/Database.php

```
      if (count($result) == 0)
      {
       $sql = $this->_ressource->prepare("
        INSERT INTO sessions (session_id, session_browser)
        VALUES (:id, :browser);
       ");

       $sql->bindParam(':id', $sessId);
       $sql->bindParam(':browser', $userAgent);
       $sql->execute();
      }
      else
      {
       // Si elle est créée on test le navigateur et s'il faut la regénéré
       $sql = $this->_ressource->prepare("
        SELECT UNIX_TIMESTAMP(session_regenerate) as session_regenerate
        FROM sessions
        WHERE session_id = :id
        AND session_browser = :browser;
       ");

       $sql->bindParam(':id', $sessId);
       $sql->bindParam(':browser', $userAgent);
       $sql->execute();

       $result = $sql->fetchAll(PDO::FETCH_ASSOC);

       if (count($result) != 1)
       {
        return false;
       }
       else
       {
        $rgTime = $result[0]['session_regenerate'];

        $now = mktime(
         date('H'),
         date('i'),
         date('s'),
         date('m'),
         date('d'),
         date('Y')
        );

        if ($now - $rgTime >= $this->_regenerateTime)
        {
         session_regenerate_id();
         $sessIdNew = session_id();

         // Met à jour session_regenerate
         $sql = $this->_ressource->prepare("
          UPDATE sessions
          SET session_regenerate = NOW(),
           session_id = :newId
          WHERE session_id = :id;
         ");

         $sql->bindParam(':newId', $sessIdNew);
         $sql->bindParam(':id', $sessId);
         $sql->execute();
        }

        return true;
       }
      }
     }
```

Ensuite, il faut bien incorporer cette méthode quelque part. Pour se faire, j'ai utilisé les méthodes magiques __set() et __get() qui nous permettrons désormais de définir ou récupérer une valeur pour une variable de session. Donc il ne faudra plus utiliser la super-globale $_SESSION.

Pourquoi n'ai-je pas incorporer cela dans la classe abstraite *Yume_Session* me direz-vous.

Tout simplement parce que pour utiliser la fonction *session_regenerate_id()*, il le fallait. En effet cette fonction utilise celles définies dans l'handler de session (c'est à dire _open, _write, ...). Donc si session_regenerate_id() fait appels à une de ces fonctions et que une des celles la font elles même appel à session_regenerate_id(). Il y aurait comme un léger problème de logique.

Voici le code de __set() et __get().

Méthode magiques __set() et __get() de la classe Yume_Session_Abstract - Yume/Session/Abstract.php

```
public function __set($key, $value)
{
 if ($this->checkInfo() === false)
 {
  session_destroy();
 }
 else
 {
  $_SESSION[$key] = $value;
 }
}

public function __get($key)
{
 if ($this->checkInfo() === false)
 {
  session_destroy();
 }
 else
 {
  return $_SESSION[$key];
 }
}
```

Bien évidemment, si on défini les méthode __set() et __get, il faut aussi définir les méthode __isset() et __unset().

Méthode magiques __isset() et __unset() de la classe Yume_Session_Abstract - Yume/Session/Abstract.php

```
public function __isset($key)
{
 $value = $this->__get($key);

 if (isset($value))
 {
  return true;
 }
 else
 {
  return false;
 }
}

public function __unset($key)
{
 if ($this->__isset($key) === true)
```

Méthode magiques __isset() et __unset() de la classe Yume_Session_Abstract - Yume/Session/Abstract.php

```
    {
     unset($_SESSION[$key]);
    }
   }
```

II.3 - Session Sniffing

II.3.1 - Problème

L'attaque "Session Sniffing" consiste à utiliser un **sniffer** pour récupéré l'identifiant de session ou toute autre information utiles.

II.3.2 - Solutions

Il n'y a qu'une solution possible pour se protéger de ce type d'attaque. C'est de protéger son serveur en utilisant **SSL**.

II.4 - Session Prediction

II.4.1 - Problème

L'attaque "Session Prediction" consiste comme son nom l'indique à analyser et comprendre la génération de l'identifiant de session et de pouvoir prédir cet identifiant.

II.4.2 - Solutions

La seule solution consiste à **modifier** quelques **directives de configuration** dans le fichier *php.ini*.

Ces directives sont les suivantes :

```
session.entropy_length = 0
session.entropy_file =
session.entropy_length = 16
session.entropy_file = /dev/urandom
session.hash_function = 1
session.hash_bits_per_character = 6
```

Il est bien au minimum de généré les identifiants de sessions avec **SHA1** et non plus MD5 (session.hash_function).

II.5 - Session Hijacking

II.3.1 - Problème

L'attaque "Session Hijacking" n'est rien d'autre que le résultat d'une autre attaque qui vise à récupéré un identifiant de session valide. C'est à dire que l'attaquant a réussi à avoir un SID et qu'il peut utiliser comme il le veux.

Les attaques possibles pour y arriver sont les suivantes :

- la "Session Fixation"

- la "Session Sniffing"

- la vulnérabilité très connue appellée **Cross-site scripting**.

II.3.2 - Solutions

Toutes les solutions possibles pouvant être mise en oeuvre ont déjà été énoncée dans les parties précédantes. En gros il faut regénéré le SID, utilisé si possible un serveur sécurisé SSL, propager le SID correctement.

II.6 - Session Poisoning et Session Injection

II.6.1 - Problème

Ces deux types d'attaques résulte au même problème, les données de la session se voient altérées.

Un exemple de code à ne pas mettre :

```
$_SESSION['login'] = $_REQUEST["'login'];
$_SESSION['password'] = $_REQUEST['password'];
```

On peut mettre directement ce que l'on veut dans la variable de session via l'URL : http://www.monsite.com/?login=moi&password=1234.

Voici encore un autre exemple mauvais :

```
$var = $_GET["something"];
$_SESSION[$var] = true;
```

II.6.2 - Solutions

Il n'y a pas de solutions concernant la gestion des sessions ici. Les seules solutions possibles sont de **vérifier** toutes les **données** que vous mettez dans une variable de session. Il ne faut jamais faire confiance à l'utilisateur.

Il faut **éviter** aussi d'utiliser la superglobale $_REQUEST, elle n'apporte rien de bien.

III - Conclusion

Hébergement mutualisé : ---------- Type d'hébergement où les utilisateurs qui louent un espace privé se partage un serveur dédié. Un hébergement mutualisé dispose habituellement d'un nombre de services assez restreind et est souvent limiter en ressources (afin de ne pas gêner les autres utilisateurs).
---------- Hébergement dédié : ---------- Type d'hébergement ou les utilisateurs se voient attribué un serveur comme espace privé. Un hébergement dédié permet à l'utilisateur d'installer ce que bon lui semble.
