# Développement sécurisé Web

## Table des matières

# 1 OWASP

## 1.1 Description

"**OWASP**" est le "Open Web Application Security Project".
Est une association à but non lucrative pour mettre en place des outils, documents et forum utilisables gratuitement. Ceux-ci ont pour but de concevoir, développer, opérer, et maintenir des applications qui peuvent être considérées de confiance.

## 1.2 Top Ten

Le projet le plus connu est son "**Top 10**". Celui-ci permet de décrire les 10 plus grandes familles de failles.
Référence :

- https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

#### 1.2.1.1 1 Injection

L'injection dans un interpréteur SQL ou shell de données malveillantes permet à l'attaquant de changer la sémantique de la requête conçue par le développeur et donc d'avoir accès à des données qu'il n'aurait pas du accéder, ou d'exécuter des fonctions qu'il n'aurait pas du pouvoir utiliser. Un exemple d'attaque est l'injection SQL.

#### 1.2.1.2 2 Broken Authentication and Session Management

La conception des mécanismes d'authentification et de gestion de session permettent à un attaquant d'obtenir par divers moyens un accès aux privilèges d'autres utilisateurs. Un exemple d'attaque est la fixation de session.

#### 1.2.1.3 3 Cross-Site Scripting (XSS)

L'envoie à un utilisateur d'une donnée reçue d'un autre permet à ce dernier de faire exécuter du code dynamique dans le navigateur du premier.

#### 1.2.1.4 4 Insecure Direct Object References

L'application permet aux utilisateurs de l'application d'accéder à des objets sans authentification ni autorisation.

#### 1.2.1.5 5 Security Misconfiguration

La mauvaise configuration d'un logiciel permet à un attaquant d'obtenir des privilèges sur l'application ou sur le système. Cela peut être un composant de l'application WEB (apache, tomcat…) qu'un composant système (service de transfert de fichiers, serveur de bases de données…).

#### 1.2.1.6 6 Sensitive Data Exposure

L'envoie de données sensibles à l'attaquant lui permet de trouver une faille et/ou d'améliorer sa tratégie d'exploitation d'une faille.

#### 1.2.1.7 7 Missing Function Level Access Control

Un utilisateur non authentifié ou non autorisé est capable d'accéder à une fonction dont l'accès devrait être restreint.

#### 1.2.1.8 8 Cross-Site Request Forgery (CSRF)

Un attaquant est capable d'envoyer à un utilisateur authentifié un lien qui lui permet de faire exécuter une fonction restreinte (ie nécessitant authentification et autorisation), mais ne nécessitant pas de recevoir une réponse, par exemple placer un compte utilisateur dans un groupe privilégié.

#### 1.2.1.9 9 Using Known Vulnerable Components

L'application utilise des composant logiciels vulnérables qu'un attaquant peut exploiter pour obtenir des privilèges sur l'application ou le système, par exemple une bibliothèque (json, xml…) ou un logiciel (php, ssh, base de données…)

#### 1.2.1.10 10 Unvalidated Redirects and Forwards

L'application génère des redirection en utilisant sans validation une donnée reçue d'un utilisateur. Un attaquant peut rediriger un utilisateur vers un site de non confiance.

### 1.3 Tools

L'**OWASP** distribue également des outils permettant de tester la sécurité d'applications WEB.
L'outil le plus connu est "**zaproxy**" (OWASP Zed Attack Proxy Project). Configuré comme un relais **HTTP(S)** entre un navigateur et un serveur **HTTP(S)**, il permet de montrer les requêtes et les réponses relayées, et de rejouer les requêtes après modifications manuelles (rejeu) ou automatiques (fuzzing).
Il permet d'abord d'analyser le comportement d'une application quand elle n'est pas soumise à des attaques en visualisant les contenus des échanges, par exemple pour tester la gestion de session. Il permet ensuite d'analyser le comportement lors d'attaques, par exemple pour tester **XSS**, **SQL injection**…
Référence :

- https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

## *2 Failles WEB*

### 2.1 SQL Injection

Le serveur utilise la donnée fournie par l'utilisateur pour effectuer une requête **SQL**. Cette donnée change la sémantique de la requête **SQL**. L'utilisateur peut exécuter les requêtes **SQL** qu'il souhaite et donc accéder (en lecture et/ou écriture) aux données qu'il souhaite.
⇒ Problèmes de sécurité: **confidentialité** et **intégrité**.

### 2.2 XSS reflected

**XSS** = Cross Site Scripting
Une donnée reçue d'un utilisateur est renvoyée à un autre utilisateur. Cette donnée est interprétée par le navigateur du destinataire. L'expéditeur peut exécuter du code dynamique (javascript…) dans le navigateur du destinataire
⇒ Problèmes de sécurité: **confidentialité** et **intégrité**

### 2.3 File Upload

Les utilisateurs qui téléchargent un fichier sur un serveur espèrent que ce serveur protège ce fichier. Un manque de protection peut permettre à un utilisateur tiers d'accéder à ce fichier, voire de le modifier.
⇒ Problèmes de sécurité: **confidentialité** et **intégrité**

### 2.4 Command Execution

Le serveur utilise la donnée fournie par l'utilisateur pour exécuter une commande système. Cette donnée change la sémentique de la commande système. L'utilisateur peut exécuter les commandes systèmes qu'il souhaite et donc accéder aux données qu'il souhaite.
⇒ Problèmes de sécurité: **confidentialité** et **intégrité**

## 2.5 Cross Site Request Forgery (CSRF)

Un utilisateur est forcé d'exécuter une commande dans une application WEB dans laquelle il est authentifié. Il faut que l'application WEB accepte les soumissions de formulaires par des requêtes GET. L'utilisateur est forcé à cliquer sur un lien via une page WEB ou un email.
⇒ Problèmes de sécurité: **confidentialité** et **intégrité**

# *3 Développement sécurisé*

## 3.1 Requêtes préparées SQL

### *3.1.1 mysqli*

Le module **mysqli** permet d'exécuter des requêtes préparées, ou paramétrées. Si originellement conçues pour exécuter rapidement plusieurs fois la même requête, elles permettent d'améliorer la sécurité des requêtes **SQL** car les paramètres sont passés indépendamment de la requête et donc leurs valeurs ne peuvent pas changer la sémantique de la requête.

```php
<?php
$mysqli = new mysqli("my_host", "my_user", "my_password",
"my_db");
if (mysqli_connect_errno()) {
printf("Échec de la connexion : %s\n",
mysqli_connect_error());
exit();
}
$my_column = "foo";
if ($stmt = $mysqli->prepare("SELECT column FROM table WHERE
var=?")) {
$stmt->bind_param("s", $my_column);
$stmt->execute();
$stmt->bind_result($my_result);
$stmt->fetch();
printf("%s is %s\n", $my_column, $my_result);

$my_column = "bar";
$stmt->execute();
$stmt->close();
}
$mysqli->close();
?>
```

Si la requête **SQL** possède plusieurs marqueurs alors il faut passer autant de variables à la commande **bind**, en spécifiant respectivement pour chacune quel est leur type dans le premier paramètre.

```
$stmt->bind_param("idsb", $my_integer, $my_float,
$my_string, $my_blob);
```

De la même façon il faut donner à la commande **bind_result** autant de variables qu'il y a de colonnes retournées dans la requête **SQL**.
Il est également possible d'exécuter une requête statique avec la fonction **query**.

```
$res = $mysqli->query("SELECT id FROM test");
var_dump($res->fetch_all());
```

Dans les deux cas, il est possible d'utiliser les fonctions **fetch()** ou **fetch_all()**.
Références :
* http://php.net/manual/fr/mysqli.quickstart.prepared-statements.php

### 3.1.2 *pdo*

Le module **pdo** permet d'exécuter des requêtes préparées, ou paramétrées. Si originellement conçues pour exécuter rapidement plusieurs fois la même requête, elles permettent d'améliorer la sécurité des requêtes **SQL** car les paramètres sont passés indépendamment de la requête et donc leurs valeurs ne peuvent pas changer la sémantique de la requête.
**pdo** permet de spécifier le moteur de base de données et n'est donc pas restreint à **mysql** comme le module **mysqli**.
Note : le module **pdo** permet également d'exécuter des procédures stockées qui permettent également d'améliorer la sécurité des requêtes **SQL**, mais elles ne seront pas vues ici.

```
<?php
$dbh = new
PDO("mysql:host=my_host;dbname=my_db;charset=utf8","usernam
e","password");
$sql = 'SELECT red, green, blue FROM colors where size < ?
and toto = ? and shape = ?';
$sth->bindParam(1, $taille, PDO::PARAM_INT);
$sth->bindParam(2, $toto, PDO::PARAM_STR, 12);

$sth->bindParam(3, $forme, PDO::PARAM_STR, 12);
try {
$stmt = $dbh->prepare($sql);
$stmt->execute();
$stmt->bindColumn(1, $rouge);
$stmt->bindColumn(2, $vert);
$stmt->bindColumn(3, $bleu);
```

```
 while ($row = $stmt->fetch(PDO::FETCH_BOUND)) {
  $data = $rouge . "\t" . $vert . "\t" . $bleu . "\n";
  print $data;
 }
}
catch (PDOException $e) {
 print $e->getMessage();
}
readData($dbh);
?>
```

Tout comme il est possible de nommer les colonnes, il est possible de nommer les paramètres :

```
$sql = 'SELECT red, green, blue, black FROM colors where
size < :taille and toto = :toto and shape = :forme';
$sth->bindParam(':taille', $taille, PDO::PARAM_INT);
$sth->bindParam(':forme', $forme, PDO::PARAM_STR, 12);

$sth->bindParam(':toto', $forme, PDO::PARAM_STR, 12);
[...]
$stmt->bindColumn('red', $rouge);
$stmt->bindColumn('green', $vert);
$stmt->bindColumn('blue', $bleu);

$stmt->bindColumn('black', $noir);
```

**PDO::exec** permet d'exécuter une requête **SQL** et retourne le nombre de lignes affectées, les requêtes **SELECT** ne retournent pas de résultat :

```
/* Effacement de toutes les lignes de la table FRUIT */
$count = $dbh->exec("DELETE FROM fruit WHERE couleur =
'rouge'");
```

**PDO::query** permet d'exécuter une requête **SQL** et retourne un jeu de résultats en tant qu'objet **PDOStatement** :

```
$sql = 'SELECT red, green, blue FROM colors ORDER BY name';
foreach ($conn->query($sql) as $row) {
```

```
print $row['red'] . "\t";
print $row['green'] . "\t";
print $row['blue'] . "\n";
}
```

Une autre façon de récupérer des résultats :

```
$query = $pdo->prepare("select name FROM tbl_name");
$query->execute();
for($i=0; $row = $query->fetch(); $i++){
 echo $i." - ".$row['name']."<br/>";
}
```

Références :

- http://php.net/manual/en/pdo.prepared-statements.php

## 3.2 Cookies

### *3.2.1 Principe*

Un cookie est une donnée créée par un serveur et envoyée au client où il est stocké. Ce cookie est associé à une arborescence web, un serveur web ou un domaine. A chaque fois que le client recontactera une page dans le domaine de validité du cookie, le client envoie le cookie au serveur.
Ce cookie a soit une date de fin de validité et il est alors stocké sur le disque par le navigateur, soit il n'est gardé que dans la mémoire du navigateur et est détruit quand le navigateur est fermé.
Du code dynamique appelé depuis le domaine du cookie est capable d'accéder au cookie et donc à sa valeur qu'il est également capable de modifier.

### *3.2.2 Sécurisation*

Il arrive couramment que des cookies contiennent des données sensibles. Le cas le plus courant est le cookie de session qui permet de ré-authentifier automatiquement un utilisateur déjà authentifié sans qu'il ait besoin de se ré-authentifier à chaque page accédée.
Ces cookies peuvent être protégés de plusieurs façons :

#### 3.2.2.1 secure

Le paramètre "**secure**" permet d'empêcher le navigateur d'envoyer ce cookie si la connexion n'est pas chiffrée, ainsi le cookie n'est jamais renvoyé en clair via **HTTP** et quand il est renvoyé ce n'est que protégé via **HTTPS**.
Ce paramètre permet d'empêcher le vol de cookie sensible par écoute passive du réseau.

#### 3.2.2.2 httponly

Le paramètre "**httponly**" permet d'empêcher le code dynamique, tel que le javascript, d'accéder au cookie et donc à sa valeur. Le cookie est toujours renvoyé au serveur.
Ce paramètre permet d'empêcher le vol de cookie sensible par attaque XSS.

#### 3.2.2.3 setcookie

La création d'un cookie en **PHP** se fait par la fonction "**setcookie**" :

```
bool setcookie ( string $name [, string $value
 [, int $expire = 0 [, string $path [, string $domain
 [, bool $secure = false [, bool $httponly = false ]]]]]] )
```

Par défaut le cookie n'a pas de date d'expiration. Il est alors gardé en mémoire par le navigateur et détruit quand le navigateur est fermé. Pour avoir une date d'expiration il faut la calculer à partir de la date actuelle, comme ceci par exemple :

```
time() + 3600
```

Il est également important de limiter au maximum le domaine de validité via les variables "**domaine**" qui doit contenir le nom du serveur et "**path**" qui doit contenir l'arborescence la plus restreinte possible. Par exemple plutôt que mentionner respectivement "example.com" et "/", il est préférable de mentionner "**www.example.com**" et "**/path/to/the/web/app/**".

## 3.3 Gestion de session

Une problématique du protocole **HTTP** est que les requêtes d'une page à l'autre sont indépendantes. De ce fait il faudrait authentifier l'utilisateur à chaque page accédée. Pour ne pas obliger l'utilisateur à se ré-authentifier tout le temps, le serveur fourni au client un secret, nommé identifiant de session ("**session ID**" ou "**SID**"). Le client renvoie le secret reçu à chaque accès, le serveur associant l'accès à l'utilisateur pré-authentifié.
Afin que cette gestion de session soit sécurisée, plusieurs règles doivent être suivies :

- La valeur du **SID** ne doit pas être devinable. Pour cela sa valeur doit être un nombre aléatoire parmi un espace de grande dimension, 128 bits est généralement admis comme minimum. Sinon 32 bits pour une application très peu sensible.

- La valeur du **SID** ne doit pas être stockée ailleurs que par le navigateur (en mémoire ou dans sa base de cookies) et par le serveur (dans les fichiers de session). Il est recommandé de ne pas envoyer le **SID** dans les URL car celles-ci peuvent être stockées dans les journaux des relais et des serveurs. Le moyen le plus sûr est d'utiliser les cookies pour le transport des **SID**.

- La valeur du **SID** doit être changée à chaque changement de sensibilité de l'utilisateur : quand il s'authentifie, quand il se ré-authentifie (passage d'un utilisateur à un autre) et quand il ferme sa session. Ceci permet de contrer les attaques par fixation de session.

- La valeur du cookie doit être protégée de l'écoute passive du réseau. Le cookie doit être créé par le serveur avec le paramètre "**secure**".

- La valeur du cookie doit être protégée de l'accès par du code malveillant, par exemple une exploitation d'une faille **XSS**. Le cookie doit être créé par le serveur avec le paramètre "**httponly**".

## 3.4 Encodage

Tout texte n'étant pas sensé contenir des **tags HTML** doit être échapé avec la fonction "**htmlspecialchars()**".

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

Voici un exemple d'encapsulation de cette fonction pour en faciliter son utilisation :

```
//xss mitigation functions
function xssafe($data,$encoding='UTF-8')
{ return htmlspecialchars($data,ENT_QUOTES | ENT_HTML401,$encoding); }
function xecho($data)
{ echo xssafe($data); }
```

```
//usage example
<input type='text' name='test' value='<?php
xecho ("' onclick='alert(1)");
?>' />
```

Cette règle permet d'éviter des exploitations de failles de type **XSS**.
La fonction "**htmlspecialchars()**" ne traduit que les cinq caractères suivants :
& " ' < >

ce qui est suffisant pour les **XSS**.
La fonction "**htmlentities()**" permet de traduire toutes les entités html :

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

## 3.5 Jetons anti-CSRF

Pour contrer les attaques de type **CSRF**, il est nécessaire de créer un jeton **anti-CSRF** à chaque accès à une page contenant une requête dynamique, cette dernière incluant le jeton **anti-csrf**. Quand le navigateur va ensuite envoyer sa requête, celle-ci va contenir le jeton. La requête ne sera acceptée que lorsque le jeton reçu du client sera égal à celui qui lui a été envoyé précédemment. Pour vérifier cela il est nécessaire d'associer la valeur du jeton aux données de session.
Exemple de création d'un jeton quand l'utilisateur accède au formulaire :

```php
session_start();
$_SESSION['token'] = bin2hex(openssl_random_pseudo_bytes(16));
// Il faut ensuite envoyer cette donnée dans un champ caché du formulaire
```

Exemple de vérification d'un jeton quand l'utilisateur envoie ses données :

```php
session_start();
if (!empty($_POST['token'])) {
  if (hash_equals($_SESSION['token'], $_POST['token'])) {
    // Gérer la requête...
  } else {
    // L'utilisateur a réalisé une requête avec une valeur erronée du jeton anti-CSRF !!!
  }
} else {
  // L'utilisateur a réalisé une requête sans jeton anti-CSRF !!!
}
// Dans tous les cas, détruire le jeton anti-CSRF car il est à usage unique :
unset($_SESSION['token']);
```

## 3.6 Expressions rationnelles

Note : les expressions rationnelles POSIX ne sont plus supportées à partir de PHP7 => utiliser PCRE.
Une documentation sur les expressions rationnelles Perl (PCRE) est la suivante :

- https://www.pcre.org/current/doc/html/pcre2syntax.html

Il faut juste débuter et finir l'expression PCRE par le caractère "/", voir les exemples ci-dessous.
La fonction "**preg_match()**" permet que chercher les correspondances d'une expression rationnelle dans une chaine de caractères.

```php
int preg_match ( string $pattern , string $subject [, array &$matches [, int $flags = 0 [, int $offset = 0 ]]] )
```

- http://php.net/manual/en/function.preg-match.php

Exemples de recherches :

```php
// Sans classe :
preg_match("/^[0-9]{4,5}$/", "12345");
// Avec une classe posix :
preg_match("/^[[:digit:]]{4,5}$/", "12345");
// Avec une classe perl :
preg_match("/^\d{4,5}$/", "12345");
```

Exemple de recherche avec extraction de sous-chaines (ce qui correspond à un jeu de parenthèses) :

```php
preg_match('/(foo)bar(baz)/', 'foobarbaz', $matches,
PREG_OFFSET_CAPTURE);
print_r($matches);
```

## 3.7 Hash

La fonction "**hash()**" permet de calculer l'empreinte (le hash) d'une chaine de caractères :

```
string hash ( string $algo , string $data [, bool $raw_output = false ] )
```

- https://secure.php.net/manual/fr/function.hash.php

Exemple de calcul d'un hash :

```php
echo hash('ripemd160', 'Le rapide goupil brun sauta par
dessus le chien paresseux.');
```

La liste des algorithmes supportés est récupérable via la fonction "**hash_algos()**" :

```
array hash_algos ( void )
```

- https://secure.php.net/manual/fr/function.hash-algos.php

Exemple de récupération de la liste des algorithmes :

```php
print_r(hash_algos());
```

## 3.8 Aléatoire

La fonction "**openssl_random_pseudo_bytes()**" permet de générer un nombre souhaité d'octets aléatoires :

```
string openssl_random_pseudo_bytes ( int $length [, bool &$crypto_strong ] )
```

- http://php.net/manual/en/function.openssl-random-pseudo-bytes.php

Exemple de génération d'une chaine composée de 8 caractères hexadécimaux (ie 32 bits) aléatoires :

```
$bytes = openssl_random_pseudo_bytes(4, $cstrong);
$hex   = bin2hex($bytes);
```

## 3.9 Crypt

La fonction "**crypt()**" permet de calculer l'empreinte d'un mot de passe via une fonction crypt Unix :

```
string crypt ( string $str [, string $salt ] )
```

- http://php.net/manual/en/function.crypt.php

Exemple de calcul de l'empreinte d'un mot de passe :

```
// $hex provient de l'exemple de
"openssl_random_pseudo_bytes()"
echo 'SHA-512:      ' . crypt('rasmuslerdorf', '$6$' . $hex .
'$') . "\n";
```

La fonction "**password_verify()**" permet de vérifier si un mot de passe correspond à une empreinte Unix calculée par une fonction crypt Unix :

```
bool password_verify ( string $password , string $hash )
```

- http://php.net/manual/en/function.password-verify.php

Exemple de vérification d'un mot de passe :

```
$hash =
```

```
'$6$10aca9db$uQYo/bkfjJ.fFDiCpcMkEHMIuL6UY/5KXqruRX5vOkSG3
Sc1dW/gBGBawLSnDC2UBfJ4YeBOzUdqUn1oT8CEE/';
if (password_verify('rasmuslerdorf', $hash)) {
  echo 'Password is valid!';
} else {
  echo 'Invalid password.';
}
```

## 3.10 Gestion de session en PHP

**TODO : ajouter des exemples**
PHP permet de gérer une session avec différentes fonctions.
La fonction "**session_start()**" permet de créer ou de restorer une session :

```
bool session_start ([ array $options = array() ] )
```

- http://www.php.net/manual/en/function.session-start.php

Celle-ci permet de positionner un verrou sur le fichier de la session. Quand le script se termine, la session est sauvegardée et le verrou est supprimé.
Exemples d'utilisation :

```
// Lors de la création ou de la restauration d'une session :
<?php
session_start();
if (!isset($_SESSION['count'])) {
 $_SESSION['count'] = 0;
} else {
 $_SESSION['count']++;
}
?>
```

```
// Lors de la terminaison d'une session :
<?php
session_start();
unset($_SESSION['count']);
?>
```

La fonction"**session_write_close()**" permet de fermer une session :

```
bool session_write_close ( void )
```

- http://www.php.net/manual/en/function.session-write-close.php

Celle-ci permet de sauvegarder la session immédiatement et de supprimer le verrou sur le fichier de session.
La fonction "**session_destroy()**" permet de détruire la session :

```
bool session_destroy ( void )
```

- http://www.php.net/manual/en/function.session-destroy.php

La fonction "**session_regenerate_id()**" permet de régénérer un ID de session :

```
bool session_regenerate_id ([ bool $delete_old_session = FALSE ] )
```

- http://www.php.net/manual/en/function.session-regenerate-id.php

La fonction "**session_create_id()**" permet de créer un ID de session :

```
string session_create_id ([ string $prefix ] )
```

- http://www.php.net/manual/en/function.session-create-id.php

La fonction "**session_set_cookie_params()**" permet de modifier les paramètres du cookie de session :

```
bool session_set_cookie_params ( int $lifetime [, string $path [, string
$domain [, bool $secure = FALSE [, bool $httponly = FALSE ]]]] )
```

- http://www.php.net/manual/en/function.session-set-cookie-params.php

Attention, la fonction "**session_set_cookie_params()**" est à appeler avant la fonction "**session_start()**".
Exemple d'utilisation de la fonction "**session_set_cookie_params()**" :

```php
// Qd on crée une session
 $lifetime=600;
 session_set_cookie_params($lifetime);
 session_start();
```

```php
// Qd on veut mettre à jour le cookie
 $lifetime=600;
 session_start();
 setcookie(session_name(),session_id(),time()+$lifetime);
```

La fonction "**session_get_cookie_params()**" permet de récupérer les paramètres du cookie de session :

```
array session_get_cookie_params ( void )
```

Liens :

- http://www.php.net/manual/en/function.session-get-cookie-params.php

- Examples : https://secure.php.net/manual/en/session.examples.php
- Sessions and Security : https://secure.php.net/manual/en/session.security.php