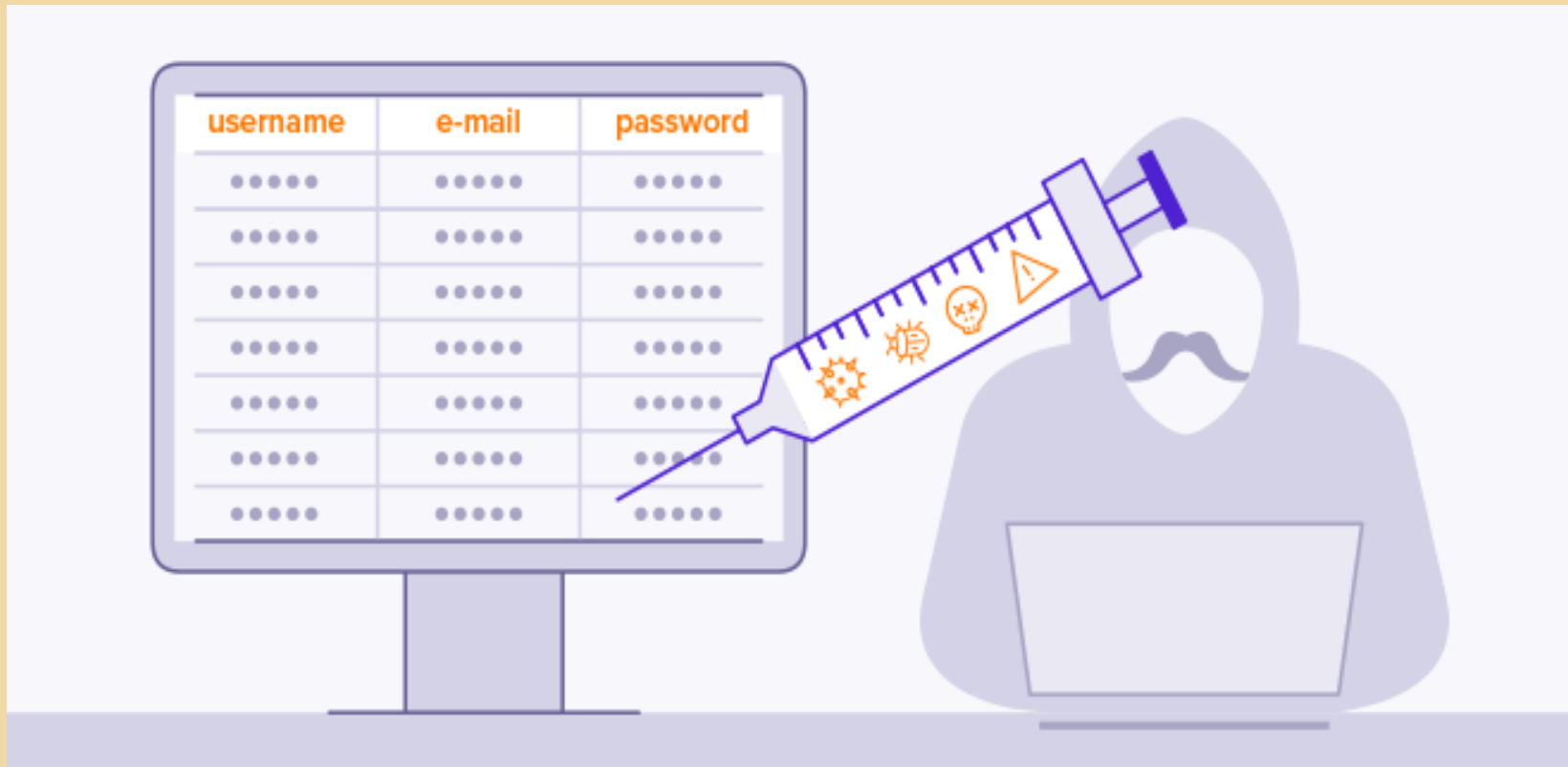


# ***SQL Injections***



# Quelques Définitions...

L'**attaque** SQL consiste à modifier une requête SQL en cours par l'injection d'un morceau de requête non prévu, souvent par le biais d'un **formulaire**. Le hacker peut ainsi accéder à la **base de données**, mais aussi **modifier le contenu** et donc compromettre la sécurité du système.

**ORACLE**

Une injection SQL est une forme de **cyberattaque** lors de laquelle un pirate utilise un morceau de **code SQL**, pour **manipuler une base de données** et accéder à des informations potentiellement importantes.

**KAPERSKY**

L'injection SQL est **une technique permettant d'injecter des éléments**, notamment du code **de type SQL**, dans les **champs des formulaires web** ou dans les **liens des pages** afin de les envoyer au serveur web. De cette façon, les attaquants outrepassent les contrôles de sécurité et parviennent à afficher ou à **modifier** des éléments présents dans une **base de données**, par exemple des mots de passe ou des coordonnées bancaires. L'injection SQL permet ainsi l'accès à **toutes les données personnelles** (par exemple, des identités ou coordonnées d'utilisateurs ou d'employés) ou non personnelles (les articles présents, les prix de ces derniers, etc.), contenues dans une base de données SQL.

**CNIL**

# ***Différents types d'injection SQL***

Le type d'injection ***blind based*** injecte des morceaux qui vont retourner caractère par caractère ce que l'attaquant cherche à extraire de la base de données. Ce type d'injection dépend de la réponse du serveur : soit la requête d'origine renvoie le même résultat et indique que le caractère est valide, soit elle ne renvoie pas le même résultat et signifie que le caractère testé n'est pas le bon.

La méthode ***error based*** injecte des morceaux qui retournent champ par champ ce que le hacker cherche à extraire de la base de données. Ce type d'injection profite d'une faiblesse de la base de données et ainsi détourne le message d'erreur qu'elle génère.

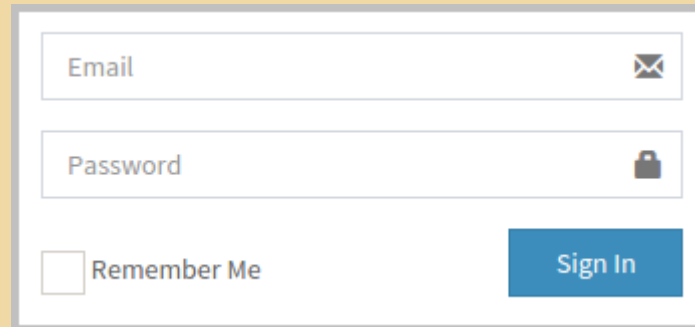
# ***Différents types d'injection SQL***

L'injection SQL **union based** injecte des morceaux qui vont retourner un ensemble de données directement extraites de la base de données. Ce type d'injection peut dans les cas les plus extrêmes récupérer des tables entières de base de données en une ou deux requêtes ! Mais, en général, cette méthode retourne entre 10 et 100 lignes de la base de données par requête SQL détournée.

La méthode **Stacked queries** est l'attaque la plus dangereuse. En raison d'une erreur de configuration du serveur de base de données, ce type d'injection peut exécuter n'importe quelle requête SQL sur le système ciblé. Non seulement elle récupère des données, mais elle peut aussi des données directement dans la base de données, par injection d'une autre requête SQL.

# ***Injection SQL par formulaire***

Imaginons que vous ayez sur votre site un formulaire de connexion pour vos utilisateurs de cette forme :



The image shows a login form with a light gray border. It contains two input fields: 'Email' with an envelope icon on the right, and 'Password' with a lock icon on the right. Below these fields is a checkbox labeled 'Remember Me'. To the right of the checkbox is a blue button with the text 'Sign In'.

# ***Injection SQL par formulaire***

Ce site utilise la requête SQL suivante pour identifier un utilisateur :

```
SELECT uid FROM Users WHERE name = '(nom)' AND password = '(mot de passe hashé)';
```

L'utilisateur Dupont souhaite se connecter avec son mot de passe « truc » hashé en [MD5](#). La requête suivante est exécutée :

```
SELECT uid FROM Users WHERE name = 'Dupont' AND password =  
'45723a2af3788c4ff17f8d1114760e62';
```

# ***Injection SQL par formulaire***

## ***Attaquer la requête :***

Imaginons à présent que le script PHP exécutant cette requête ne vérifie pas les données entrantes pour garantir sa sécurité. Un [hacker](#) pourrait alors fournir les informations suivantes :

Utilisateur : *Dupont';--*  
Mot de passe : *n'importe lequel*

La requête devient : `SELECT uid FROM Users WHERE name = 'Dupont';--' AND password = '4e383a1918b432a9bb7702f086c56596e';`

Les caractères `--` marquent le début d'un commentaire en SQL.

La requête est équivalente à :

```
SELECT uid FROM Users WHERE name = 'Dupont';
```

L'attaquant peut alors se connecter sous l'utilisateur Dupont avec n'importe quel mot de passe. Il s'agit d'une injection de SQL réussie, car l'attaquant est parvenu à injecter les caractères qu'il voulait pour modifier le comportement de la requête.

# *Injection SQL par formulaire*

## ***Attaquer la requête :***

Supposons maintenant que l'attaquant veuille non pas tromper le [script](#) SQL sur le nom d'utilisateur, mais sur le mot de passe. Il pourra alors injecter le code suivant :

Utilisateur : *Dupont*

Mot de passe : '*or 1 --*

L'apostrophe indique la fin de la zone de frappe de l'utilisateur, le code « *or 1* » demande au script si *1 est vrai, or c'est toujours le cas*, et *--* indique le début d'un commentaire.

La requête devient alors : `SELECT uid FROM Users WHERE name = 'Dupont' AND password = " or 1 --';`

Ainsi, le script programmé pour vérifier si ce que l'utilisateur tape est vrai, il verra que 1 est vrai, et ***l'attaquant sera connecté sous la session Dupont.***



# ***Injection SQL en aveugle***

Une « **Total Blind SQL Injection** », c'est une injection qui renvoie un résultat mais où ce dernier n'est pas affiché et où vous êtes donc incapable, visuellement, de savoir si oui ou non la requête a renvoyé un résultat (ce qui ne veut pas dire qu'elle n'a forcément rien renvoyé pour autant....)

*un autre nom de la « Total Blind SQL Injection », c'est... « **TIME Based SQL Injection** ».*

*Le temps ! Voilà ce qui va nous permettre de déterminer si la requête a renvoyé, ou non, un résultat.*

# ***Injection SQL en aveugle***

## *Extrait de code en php*

```
<body>
  <?php
    if(!empty($_GET['id']))
    {
      $id = mysqli_real_escape_string($db, $_GET['id']);
      $query = "SELECT id, username FROM users WHERE id = ".$id;
      $rs_article = mysqli_query($db, $query);
    }
  ?>
</body>
```

# ***Injection SQL en aveugle***

## ***Attaque booléenne :***

Si une requête SQL aboutit à une erreur qui n'a pas été traitée en interne dans l'application, la page Web résultante peut lancer une erreur, charger une page blanche ou se charger partiellement. Dans une injection SQL booléenne, un attaquant évalue quelles parties de l'entrée d'un utilisateur sont vulnérables aux injections SQL en essayant deux versions différentes d'une clause booléenne à travers l'entrée :

« ... and 1=1 »

« ... and 1=2 »

Si l'application fonctionne normalement dans le premier cas mais présente une anomalie dans le second cas, cela indique que l'application est vulnérable à une attaque par injection SQL.

# ***Injection SQL en aveugle***

**Attaque booléenne :** "SELECT id, username FROM users WHERE id = ".\$id;

Tentons d'effectuer quelques tests :

`http://localhost/zds/injections_sql/time.php?id=1 AND 1=1`

Requête : `SELECT id, username FROM users WHERE id = 1 AND 1=1`

Et maintenant une requête où la condition n'est pas vérifiée (car 1 n'est pas égal à 2).

`http://localhost/zds/injections_sql/time.php?id=1 AND 1=2`

Requête : `SELECT id, username FROM users WHERE id = 1 AND 1=2`

***Absolument aucune différence (en apparence) et pourtant la première requête a bel et bien renvoyé un résultat.***

***A suivre l'histoire d'un sleep()...***

# ***Injection SQL en aveugle***

## ***Attaque basée sur le temps :***

Une attaque par injection SQL basée sur le temps peut également aider un attaquant à déterminer si une vulnérabilité est présente dans une application web. Un attaquant utilise une fonction temporelle prédéfinie du système de gestion de base de données utilisé par l'application. Par exemple, dans MySQL, la **fonction sleep()** indique à la base de données d'attendre un certain nombre de secondes.

```
select * from comments  
WHERE post_id=1-SLEEP(15);
```

Si une telle requête entraîne un délai, l'attaquant sait qu'elle est vulnérable.

***De cette façon, nous pourrions facilement différencier le « oui » du « non »***

# ***Injection SQL en aveugle***

## ***Attaque basée sur le temps :***

Nous utiliserons également ce que l'on appelle une sous-requête : c'est une requête qui sera exécutée avant notre requête principale.

Il y a d'autres possibilités (en utilisant une condition IF() par exemple).

`http://localhost/zds/injections_sql/time.php?id=-1 OR (SELECT SLEEP(3)  
FROM users WHERE id=1 AND 1=2)`

Requête : `SELECT id, username FROM users WHERE id = -1 OR (SELECT  
SLEEP(3) FROM users WHERE id=1 AND 1=2)`

La sous-requête demande...

Patiente 3 secondes SI 1 est égal 2, *ici ce n'est pas le cas.*

# ***Injection SQL en aveugle***

## ***Attaque basée sur le temps :***

http://localhost/zds/injections\_sql/time.php?id=-1 OR (SELECT SLEEP(3)  
FROM users WHERE id=1 AND 1=1)

Requête :SELECT id, username FROM users WHERE id = -1 OR (SELECT  
SLEEP(3) FROM users WHERE id=1 AND 1=1)

Patiente 3 secondes Si 1 est égal 1, ici c'est le cas.

# ***Injection SQL en aveugle***

## ***Attaque basée sur le temps :***

Et ainsi de suite, pour tester les caractères d'un mot de passe...

Est-ce que le mot de passe commence par **a** ? **LIKE** avec des caractères hexadécimal.

`http://localhost/zds/injections_sql/time.php?id=-1 OR (SELECT SLEEP(3)  
FROM users WHERE id=1 AND password LIKE 0x6125)`



# *Injection SQL Solutions*

## *Échappement des caractères spéciaux:*

Vous pouvez aussi échapper une chaîne de caractères dans MySQL en utilisant la fonction `mysqli_real_escape_string()`.

Qui transformera la chaîne ' -- en \' -- , la requête deviendrait alors :

```
SELECT uid FROM Users WHERE name = 'Dupont\' -- ' AND password  
='4e383a1918b432a9bb7702f086c56596e';
```

Lors de l’affichage de la sortie en HTML, vous devrez également convertir la chaîne de caractères pour vous assurer que les caractères spéciaux n’interfèrent pas avec le balisage HTML.

En PHP en utilisant la fonction `htmlspecialchars()` par exemple,

# *Injection SQL Solutions*

## ***Utiliser des déclarations préparées:***

Vous pouvez également utiliser des déclarations préparées pour éviter les injections SQL. Une instruction préparée est un modèle de requête SQL, dans lequel vous spécifiez des paramètres à un stade ultérieur pour l'exécuter.

Voici un exemple d'une déclaration préparée en PHP et MySQLi.

```
$query = $mysql_connection->prepare("select * from user_table where  
username = ? and password = ?");  
$query->execute(array($username, $password));
```

# ***Injection SQL Solutions***

## ***Autres contrôles pour prévenir les attaques SQL:***

L'étape suivante pour atténuer cette vulnérabilité est de limiter l'accès à la base de données au strict nécessaire.

Par exemple, connectez votre application Web au DBMS en utilisant un utilisateur spécifique qui n'a accès qu'à la base de données pertinente.

Restreindre l'accès de la base de données utilisateur à tous les autres emplacements du serveur.

Vous pouvez également bloquer certains mots-clés SQL dans votre URL via votre serveur Web.

# ***Injection SQL Ressources***

<https://www.oracle.com/fr/security/injection-sql-attaque.html>

<https://zestedesavoir.com/tutoriels/945/les-injections-sql-le-tutoriel/>

[https://fr.wikipedia.org/wiki/Injection\\_SQL](https://fr.wikipedia.org/wiki/Injection_SQL)

<https://kinsta.com/fr/blog/injections-sql/>