

Secteur Tertiaire Informatique  
Filière « Etude et développement »

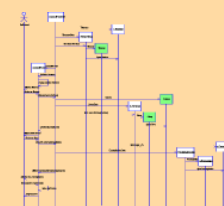
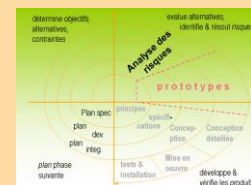
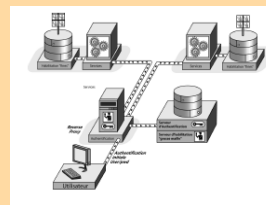
Séquence « Développer des composants d'accès  
aux données »

Sécuriser l'accès et l'utilisation de la base de  
données

Apprentissage

Mise en situation

Evaluation



Version	Date	Auteur(s)	Action(s)
1.0	28/08/16	Lécu Régis	Création du document

Sécuriser l'accès et l'utilisation de la base de données

Afpa © 2014 – Section Tertiaire Informatique – Filière « Etude et développement »

## TABLE DES MATIERES

Table des matières .....	2
1. Introduction .....	5
2. Présentation de l'attaque par injection .....	6
2.1 Principe de l'attaque par injection SQL.....	6
2.2 Fréquence et gravité des attaques par injection .....	7
2.3 Précisons les choses à l'aide d'OWASP .....	8
3. L'attaque par injection SQL en Java et ses parades .....	9
3.1 Attaque par injection SQL en Java avec JDBC.....	9
3.2 Première parade : les requêtes paramétrées ( <i>Prepared Statement</i> ).....	10
3.3 Deuxième parade : les Procédures Stockées.....	12
3.3.1 Une bonne pratique pour la qualité et la sécurité du logiciel .....	12
3.3.2 Le bon usage des procédures stockées.....	12
3.3.3 Limiter et contrôler ses sorties .....	13
3.3.4 Ce qu'il vaut mieux éviter.....	14
3.4 Valider ou « échapper » les données en entrée .....	14
3.4.1 Valider les données en entrée par une liste blanche.....	15
3.4.2 Méthode d'échappement des données ( <i>Escape</i> ) .....	15
3.5 Troisième parade : la couche de persistance (ORM).....	15
4. Séparer et limiter les privilèges des utilisateurs.....	15
4.1 Gérer les utilisateurs authentifiés .....	16
4.1.1 A éviter : protéger l'application cliente par un mot de passe local .....	16
4.1.2 A éviter : une connexion de service pour tous les utilisateurs .....	16
4.1.3 Ce qu'il faut faire : utiliser des connexions distinctes avec des droits limités.....	16
4.1.4 Une bonne mise en œuvre : droit d'exécution sur les procédures stockées, lecture sur les tables et les vues .....	17
4.2 Gérer les clients externes.....	17
5. Conclusion .....	18

## Objectifs

A l'issue de cette séance, le stagiaire sera capable de sécuriser l'accès et l'utilisation de la base de données, tant au niveau du *middleware* que des composants serveurs :

- connaître l'attaque par injection SQL
- prévenir les attaques par injection côté client, en n'utilisant que des requêtes paramétrées ou en appelant des procédures stockées, et en vérifiant systématiquement les entrées ;
- construire une « défense en profondeur » en testant une deuxième fois toutes les entrées des procédures stockées (pour détecter en particulier des tentatives d'injection SQL) ;
- authentifier les utilisateurs en s'appuyant sur la sécurité du SDBD, afin d'assurer la confidentialité des informations.

L'administration du SGBD et son paramétrage sécurisé seront précisés ultérieurement dans la séquence « *Mettre en place une base de donnée* ». Mais nous allons présenter dès maintenant les bonnes pratiques sur les connexions et les permissions.

## Pré requis

Cette séance est une synthèse sur la sécurisation de l'accès et de l'utilisation des bases de données. Elle a pour prérequis la connaissance d'un langage objet et du langage SQL : séquence « *Développer une interface utilisateur* », et trois premières séances de la séquence en cours : « *Ecrire des requêtes SQL simples* », « *Programmer des fonctions, des procédures stockées et des déclencheurs* », et « *Utiliser un middleware d'accès aux données* ».

## Méthodologie

Ce document peut être utilisé en présentiel ou à distance.

Il précise la situation professionnelle visée par la séance, la situe dans la formation, et guide le stagiaire dans son apprentissage et ses recherches complémentaires.

## Mode d'emploi

Symboles utilisés :



Renvoie à des supports de cours, des livres ou à la documentation en ligne constructeur.



Propose des exercices ou des mises en situation pratiques.



Point important qui mérite d'être souligné !

## Ressources

- Document CyberEdu stagiaire : CyberEdu\_module\_3\_reseau\_et\_applicatifsV2.pdf
- Les vidéos en anglais de « *l'OWASP AppSec Tutorial Series* » traitent de questions de sécurité sur le Web, sur les bases des données etc : vulnérabilités, attaques et les méthodes correctives :  
[https://www.owasp.org/index.php/OWASP\\_Appsec\\_Tutorial\\_Series#tab=Episode\\_List](https://www.owasp.org/index.php/OWASP_Appsec_Tutorial_Series#tab=Episode_List)
- Pour faciliter la tâche au lecteur non anglophone, nous fournissons l'épisode 2 de la série d'OWASP, qui traite de l'injection en général et de l'injection SQL, avec un sous-titrage en français :  
OWASP Appsec Tutorial Series - Episode 2 SQL Injection-fr.mp4 et  
OWASP Appsec Tutorial Series - Episode 2 SQL Injection-francais.srt
- N'hésitez pas à parcourir le site de l'OWASP qui est une véritable mine sur la sécurité (Java, C# etc., comme celui du CERT vu précédemment). Sur l'injection SQL :
  - [L'Injection SQL par l'OWASP](#) : bonne présentation de l'injection SQL
  - [Aide-mémoire pour la prévention de l'Injection SQL par l'OWASP](#) : référence pour les parades contre les injections SQL
- Autres documents fournis (en anglais) :
  - Advanced\_SQL\_Injection.ppt : définition, recherche et défense contre les injections SQL
  - Secure-Software-5-Call-Out.ppt : exposé de David Wheeler sur l'injection SQL

### Ressource formateur :

Projet CyberEdu : CyberEdu\_developpement\_09\_2015,

Fiche 1 « Vulnérabilités standard », injection SQL

# 1. INTRODUCTION

Cette séance est une synthèse sur la sécurisation de l'accès et de l'utilisation des bases de données. Elle n'introduit pas de nouveaux savoirs sur les bases de données, mais leur applique les principes de sécurité que nous avons déjà abordés, et que vous mettrez en œuvre dans la mise en situation de la séquence :

- posture de méfiance : toute donnée externe doit être considérée comme potentiellement toxique ;
- défense en profondeur : si un attaquant parvient à compromettre le client, il reste une chance que son attaque soit mise en échec par les contrôles serveur.

Dans les trois séances précédentes sur le développement sécurisé, nous avons suivi les objectifs du projet CyberEdu en :

- découvrant les vulnérabilités courantes des langages C et Java<sup>1</sup>,
- apprenant à les prévenir efficacement en codant dans un style défensif<sup>2</sup>,
- utilisant notre interface utilisateur, côté client, comme un premier niveau de défense<sup>3</sup>.

Nous avons conclu qu'une défense côté client est indispensable mais insuffisante : elle doit s'intégrer dans une stratégie globale de défense en profondeur, où chaque couche logicielle adopte une posture de méfiance et teste à nouveau ses entrées. Cette façon de faire permet de limiter la portée d'une attaque, comme dans un château-fort où il ne suffit pas de franchir le rempart pour s'emparer du donjon.

Dans cette séance, nous allons voir un premier cas de défense en profondeur, dans une architecture client/serveur classique (2 Tiers). Le serveur est le SGBD : la défense en profondeur doit donc être assurée par les clés primaires et étrangères, par les contraintes (CHECK), par des tests systématiques sur les paramètres en entrée dans les procédures stockées, et par des Déclencheurs.

Nous verrons que la gestion des utilisateurs, l'attribution de rôles et d'autorisations ciblées jouent un rôle important dans la sécurité de la base (Disponibilité, Intégrité, Confidentialité, et éventuellement Preuve), en limitant la portée de l'attaque : le *hacker* qui parviendrait à franchir les contrôles client et à se connecter à la base, sera limité dans son attaque aux objets qui lui sont autorisés.

Des transactions bien gérées sont également essentielles, à la fois pour la cohérence fonctionnelle de l'application et pour sa sécurité. Ce point sera examiné ultérieurement dans la séance « *Sécuriser la base de données à l'aide de transactions et de déclencheurs* ».

L'approche sécurité oblige le développeur de bases de données à prendre du recul : il doit envisager les conséquences de son travail sur la globalité de l'application, et en particulier éliminer les possibilités d'injection SQL qui peuvent créer des vulnérabilités.

---

<sup>1</sup> Identifier les spécificités des langages et les attaques classiques

<sup>2</sup> Codifier de façon défensive en suivant les bonnes pratiques de sécurité

<sup>3</sup> Sécuriser l'interface utilisateur

## 2. PRESENTATION DE L'ATTAQUE PAR INJECTION

### 2.1 PRINCIPE DE L'ATTAQUE PAR INJECTION SQL

Lire le document : [CyberEdu\\_module\\_3\\_reseau\\_et\\_applicatifsV2.pdf](#), pp. 72-77

Si vous découvrez ce chapitre en autonomie, suivre ce



Guide de lecture

(p. 73) Pour montrer la généralité de l'attaque par injection, le document la situe dans une application 4 Tiers. Le fonctionnement sera le même en client/serveur, qui est traité dans cette séance. Les attaques et la sécurisation des sites web et des architectures 4 Tiers seront traitées dans des séances ultérieures<sup>4</sup>.

(p. 75) L'exemple est écrit en PHP où : `$user` est une variable ; pour l'interpréteur PHP, le `$` est prioritaire sur les apostrophes de la chaîne dans `'$user'` ; donc la variable `$user` est remplacée par son contenu dans la chaîne : `'thomas'`

(pp. 76-77) A bien noter dans les données de l'attaque :

- l'apostrophe qui termine artificiellement la chaîne de mot de passe : `abcd'`
- et permet donc d'enchaîner par un `or` sur n'importe quelle expression logique : par exemple `1=1` qui est toujours vrai
- le marqueur de commentaire SQL `--` est indispensable pour éliminer l'apostrophe qui termine la chaîne `$mdp` : `'$mdp'`
- avec le commentaire terminateur, la chaîne de requête est syntaxiquement correcte, puisque l'apostrophe dans le commentaire est ignorée :

```
select count(*) from user where name='azerty' and mdp='abcd' or 1=1 --'
```

Du fait de la précédence du ET sur le OU en SQL, cette requête équivaut à :

```
select count(*) from user where (name='azerty' and mdp='abcd') or 1=1
```

et elle retourne donc le nombre total d'utilisateurs de la table `user`, quelque soit le nom et le mot de passe fournis !

- sans la marque de commentaire `--`, PHP construirait la chaîne de requête :

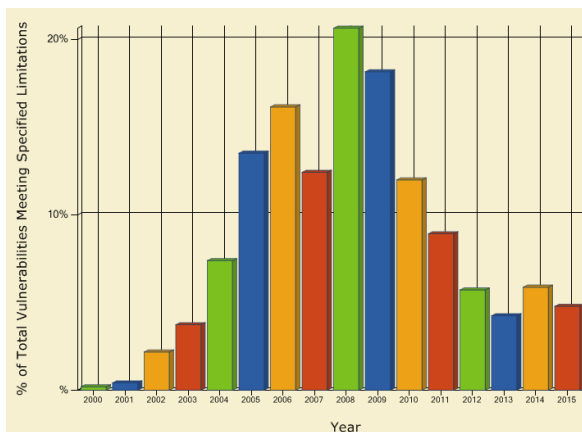
```
select count(*) from user where name='azerty' and mdp='abcd' or 1=1'
```

qui serait refusée par l'interpréteur SQL à cause de l'apostrophe terminateur, en rouge.

---

<sup>4</sup> « Identifier les failles de sécurité et appliquer les bonnes pratiques de sécurisation des applications web » et « Sécuriser les couches d'une application N Tiers »

## 2.2 FREQUENCE ET GRAVITE DES ATTAQUES PAR INJECTION



L'attaque par injection SQL est le cas le plus connu d'un grand ensemble, les attaques par injection, qui sont en baisse par rapport à 2008, mais restent nombreuses et préoccupantes.

Les failles d'injection sont courantes dans les applications web. Il en existe différents types: SQL, LDAP, XPath, XSLT, HTML, XML, commandes systèmes etc.

Une injection se produit lorsqu'une donnée fournie par l'utilisateur est envoyée à un interpréteur dans le cadre d'une commande ou d'une requête. Les attaquants dupent l'interpréteur en lui faisant exécuter des commandes arbitraires, via la soumission de données spécialement formées : par exemple, `abcd' or 1=1--` dans l'exemple précédent.

Les failles d'injection permettent aux attaquants de créer, lire, modifier ou effacer des données, de façon arbitraire, dans l'application.

Dans le pire des cas, ces failles permettent à un attaquant de compromettre complètement l'application et le système sous-jacent, ou de détruire la base de données, en parvenant à contourner des environnements de filtrage bien structurés :



Les radars automatiques utilisant la reconnaissance de caractère (OCR) pour identifier les véhicules de façon automatisée, cette injection SQL inhibe l'enregistrement de la plaque et détruit la base de données, empêchant le radar de fonctionner après le passage de cette voiture.

(<http://blogmotion.fr/internet/divertissement/detecteur-radar-pv-5266>)

Sécuriser l'accès et l'utilisation de la base de données

Afpa © 2014 – Section Tertiaire Informatique – Filière « Etude et développement »



## 2.3 PRECISONS LES CHOSES A L'AIDE D'OWASP

Le site OWASP est une référence pour la sécurité du web et une mine pour la sécurité en général. Il fournit à la fois des vidéos d'initiation et des documents techniques pour approfondir une question.



Suivre la vidéo d'OWASP sur les injections et l'injection SQL (sous-titrée en français) :

*OWASP Appsec Tutorial Series - Episode 2 SQL Injection-fr.mp4*

A la fin, la vidéo présente la documentation de référence d'OWASP, qui sera nécessaire pour approfondir :

[L'Injection SQL par l'OWASP](#) : page de référence avec les différentes entrées sur l'injection

[Aide-mémoire pour la prévention de l'Injection SQL](#) : détaille les parades contre l'injection SQL, dans différents langages (Java et C#), différentes bases de données et différents environnements (par exemple Java avec *Hibernate*)



N'hésitez pas à parcourir cette documentation.

Les attaques par injection SQL sont possibles dans tous les langages et les SGBD, indépendamment du nombre de tiers de l'architecture.

Elles peuvent survenir dès qu'un programme construit dynamiquement ses requêtes SQL en y insérant des données provenant de l'extérieur : champ de saisie dans un client lourd, paramètre *http*, donnée issue d'une autre base non sûre, d'un fichier XML etc.

Si le développeur n'adopte pas une posture de méfiance en contrôlant au préalable toutes les données externes qui vont intervenir dans la requête, il sera possible à un pirate de détourner la requête de son objectif, pour consulter l'ensemble de la base de données, voire en modifier le contenu ou la détruire.

Nous allons maintenant illustrer l'attaque par injection SQL et ses parades, par des exemples en Java JDBC et Microsoft Transact SQL.

### 3. L'ATTAQUE PAR INJECTION SQL EN JAVA ET SES PARADES

#### 3.1 ATTAQUE PAR INJECTION SQL EN JAVA AVEC JDBC

Reprenons de l'exemple PHP fourni par CyberEdu qui vérifie qu'un utilisateur est connu dans la base, avec un mot de passe donné :

```
select count(*) from user where name='$name' and mdp='$mdp'
```

En Java, on peut exécuter cette requête par une instruction JDBC non paramétrée :

```
public static boolean userExist (Connection co, String nom, String motpasse)
{
    // ici, il aurait fallu valider tous les paramètres en entrée!

    boolean existe = false;
    // on construit dynamiquement la requête en mélangeant instruction SQL et données non sûres
    String req="select count(*) from utilisateur where name='" + nom
                + "' and mdp='" + motpasse + "'";

    try
    {
        // on crée la requête non paramétrée à partir d'une connexion JDBC existante : co
        Statement stmt = co.createStatement();
        ResultSet rs = stmt.executeQuery(req);
        if (rs.next())
        {
            // on code approximativement la fonctionnalité demandée !
            if (rs.getInt(1) != 0)
                existe = true ;
        }
    }
    catch (SQLException e)
    {
        System.out.println ("Exception : " + e.getMessage()) ;
    }
    return existe ;
}
```

A noter :

- les apostrophes ' qui entourent le nom en SQL (de type varchar) et les guillemets " de la chaîne Java.
- avec les paramètres : lecu et 007

la fonction userExist construit dynamiquement et exécute la requête :

```
select count(*) from utilisateur where name='lecu' and mdp='007'
```

qui remplit la fonctionnalité demandée (renvoie true uniquement si le couple name, mdp existe dans la table utilisateur)

- mais, sans contrôle des paramètres par le développeur, il suffit au pirate de saisir un mot de passe tel que : **regis' or 1=1 --** pour que la requête devienne :  

```
select count(*) from utilisateur where name='lecu' and mdp='regis' or 1=1 --'
```
- le commentaire du langage TRANSACT SQL -- élimine la fin de la ligne et l'apostrophe restante. Comme c'est un OU et que 1=1 est toujours vraie, tous les utilisateurs seront affichés. Mais certains interpréteurs détectent les chaînes non terminées : dans ce cas, on écrira **regis' or 1=1 -- ' '** pour terminer la chaîne et provoquer l'injection SQL :

```
select count(*) from utilisateur where name='lecu' and mdp='regis' or 1=1 -- ' '
```

Sécuriser l'accès et l'utilisation de la base de données

Afpa © 2014 – Section Tertiaire Informatique – Filière « Etude et développement »

- le plus souvent, l'injection SQL n'est efficace que si elle est accompagnée par des maladresses de programmation : la requête malicieuse ne renvoie pas 1 mais le nombre total d'utilisateurs ; mais le code teste `(rs.getInt(1) != 0)` là où il aurait fallu vérifier précisément l'unicité du compte `(rs.getInt(1) == 1)`.
- 

Cette attaque est déjà grave, puisqu'elle compromet le mécanisme d'authentification.

Mais il est toujours possible d'ignorer la fin d'une requête par le marqueur de commentaire `--` ou d'enchaîner plusieurs requêtes SQL en les séparant par le marqueur `;` ;

En insérant ces marqueurs dans un champ de saisie, un pirate peut exécuter la requête de son choix : avec les paramètres : `lecu et regis';delete utilisateur --`

la fonction `userExist` exécute la requête :

```
select count(*) from utilisateur where name='lecu' and mdp='regis';delete utilisateur--'
```

qui supprime toutes les lignes de la table `user`



### Mise en situation

Créez une base de données de démonstration avec une table `utilisateur` (deux colonnes : `name` et `mdp`).

Codez la fonction `userExist` et son programme de test, en Java

Préparez un jeu d'essai avec une utilisation normale de la fonction (compte existant / inexistant).

Préparez un jeu d'essai « malveillant » avec au moins les deux cas d'injections ci-dessus.

On peut en trouver d'autres.

Par exemple, on peut casser l'authentification comme dans le premier cas, sans utiliser de commentaire : avec les paramètres `lecu et regis' or '1'='1`,

la fonction `userExist` exécute la requête suivante qui a le même résultat :

```
select count(*) from utilisateur where name='lecu' and mdp='regis' or '1'='1'
```

---

Script de création SQL et trame du client Java dans le répertoire **Corrigés**.

*(Cette trame à compléter est commune pour les différents exercices de la séance).*

Un certain nombre de règles permettent de se prémunir des attaques par injection SQL. Empêcher une injection exige de séparer les données non fiables de la requête.

## 3.2 PREMIERE PARADE : LES REQUETES PARAMETREES (**PREPARED STATEMENT**)

Tous les développeurs devraient apprendre en premier les requêtes paramétrées :

- elles sont plus faciles à écrire et à comprendre que les requêtes dynamiques ;
- elles obligent le développeur à définir en premier tout le code SQL, et à ne passer qu'ensuite les valeurs des paramètres à la requête ;
- ce style de codage permet à la base de données de différencier le code et les données, quelque soit la valeur des paramètres en entrée ;

Sécuriser l'accès et l'utilisation de la base de données

Afpa © 2014 – Section Tertiaire Informatique – Filière « Etude et développement »

- JDBC filtre automatiquement les injections présentes dans les paramètres des *PreparedStatement*, qui sont représentés dans le texte de la requête par le marqueur ?
- les requêtes paramétrées garantissent qu'un attaquant ne pourra pas changer l'objectif d'une requête, même s'il insère des commandes SQL dans ses données.

#### Version sécurisée avec une requête paramétrée :

Si un attaquant entre comme mot de passe **regis' or '1'=1**, la requête paramétrée ne sera pas vulnérable pour autant, et recherchera dans la table un mot de passe qui correspond littéralement à la chaîne en entrée : **regis' or '1'=1**

```
public static boolean userExistV2 (Connection co, String nom, String motpasse)
{
    // TODO : tester La connexion et valider nom et motpasse
    // par des expressions régulières

    boolean existe = false;
    // Le texte de la requête paramétrée ne contient que du SQL sûr
    // Les emplacements des paramètres sont marqués par des points d'interrogation
    String req="select count(*) from utilisateur where name=? and mdp=?";
    try
    {
        // on crée la requête paramétrée à partir d'une connexion existante
        PreparedStatement pstmt = co.prepareStatement(req);

        // on donne les valeurs des paramètres à la requête
        pstmt.setString(1, nom);
        pstmt.setString(2, motpasse);

        // appel de la requête paramétrée
        ResultSet rs = pstmt.executeQuery( );
        if (rs.next())
        {
            // on code EXACTEMENT la fonctionnalité demandée
            // (en supposant l'unicité du nom)
            if (rs.getInt(1) == 1)
                existe = true ;
        }
    }
    catch (SQLException e)
    {
        System.out.println ("Exception : " + e.getMessage()) ;
    }
    return existe ;
}
```



#### Mise en situation

Testez cette version sécurisée : vérifiez que votre jeu d'essai malveillant ne parvient plus à compromettre le mécanisme d'authentification.

Les requêtes paramétrées ne nous dispensent pas de prendre des précautions élémentaires. Valider tous les paramètres en entrée (**TODO**) :

- *co* : connexion existante
- *nom* : uniquement des majuscules, minuscules et trait d'union
- *motpasse* : majuscules, minuscules, chiffres, ponctuation

Sécuriser l'accès et l'utilisation de la base de données

On remarque l'utilisation de « listes blanches » pour le nom et le mot de passe, ce qui est une bonne pratique.

### 3.3 DEUXIEME PARADE : LES PROCEDURES STOCKEES

#### 3.3.1 Une bonne pratique pour la qualité et la sécurité du logiciel

L'utilisation de procédures stockées dans une architecture client / serveur est une pratique favorable à la sécurité globale de l'application :

- les procédures stockées vont constituer une deuxième ligne de défense (le donjon) qui va à nouveau tester systématiquement les entrées, pour arrêter les attaques qui auraient passé la première ligne (le rempart extérieur, l'interface utilisateur) ;
- par le simple fait qu'elles sont stockées (*stored*) sur le serveur, elles sont plus robustes que l'interface utilisateur qui doit être déployée sur de nombreux postes clients, et peut être facilement décompilée, voir crackée ;
- comme elles sont centralisées, leur mise à jour est plus rapide et plus sûre que celles des postes client ;
- on peut limiter leurs permissions d'exécution à certains utilisateurs ou groupes (conformément au principe « *séparer et limiter les privilèges et les permissions* »).

#### 3.3.2 Le bon usage des procédures stockées

Les procédures stockées peuvent elles aussi être rendues vulnérables aux injections SQL, mais l'on peut garantir le même niveau de sécurité qu'avec les requêtes paramétrées JDBC, si :

- o l'on évite de construire des requêtes dynamiques (en passant par exemple un nom de table ou de colonne en paramètre)
- o l'on n'utilise donc les paramètres en entrée et en sortie que comme valeur ou résultat des requêtes SQL de la procédure.
- o on récupère systématiquement les exceptions, afin de garder le contrôle du code quoi qu'il arrive
- o on vérifie systématiquement la longueur, le type et la sémantique des entrées
- o on contrôle et on minimise l'information communiquée en sortie

Procédure stockée simple qui remplit la même fonctionnalité que la requête paramétrée :

```
CREATE PROCEDURE userExist @name varchar(30), @mdp varchar(20)
AS
BEGIN
    -- TODO : valider Le nom et Le mdp (règles de gestion ?)
    declare @nb int;

    select @nb = count (*)
    from utilisateur
    where name = @name and mdp = @mdp;

    -- name est la clé primaire de la table utilisateur :
    -- INVARIANT : @nb ne peut valoir que 0 ou 1, ou c'est une attaque
    -- dans ce cas, on renvoie 0 (non identifié)
    if @nb = 1
        return 1;
    else
        return 0;
END
```

Sécuriser l'accès et l'utilisation de la base de données

Afpa © 2014 – Section Tertiaire Informatique – Filière « Etude et développement »

## Appel de la procédure stockée en JDBC

```
public static boolean userExistV3 (Connection co, String nom, String motpasse)
{
    boolean existe = false;
    try
    {
        CallableStatement cs = co.prepareCall("{?=call dbo.userExist(?,?)}");

        // type de la valeur de retour (forcément un entier)
        cs.registerOutParameter (1, java.sql.Types.INTEGER );

        cs.setString(1, nom);
        cs.setString(2, motpasse);
        cs.execute();
        existe = (cs.getInt(1) == 1);
    }
    catch (SQLException se)
    {
        System.out.println ("Exception : " + e.getMessage()) ;
    }
}
```



### Mise en situation

Cette technologie vous est connue<sup>5</sup>, mais nous allons vérifier qu'elle résout bien le problème des injections SQL :

- écrivez la procédure stockée `userExist`, et son test unitaire en JDBC
- vérifiez que votre jeu d'essai malveillant ne parvient pas à compromettre le mécanisme d'authentification.

### 3.3.3 Limiter et contrôler ses sorties

C'est une autre application de la posture de méfiance :

- si l'on ne s'adresse pas à un tiers de confiance, il faut lui en dire le moins possible ;
- vu du serveur, le client ne doit jamais être considéré comme totalement sûr, sinon le serveur ne servirait à rien dans la stratégie de sécurité (de même que les gardes dans le donjon doivent toujours se méfier des remparts extérieurs, même s'ils sont en théorie amis). On contrôlera donc soigneusement l'information renvoyée au client.

Pour la procédure précédente, le booléen en retour suffit : on autorise ou pas la connexion demandée ; il ne faut surtout pas préciser lequel des deux paramètres est erroné.

Pour des procédures stockées plus complexes, il faudra en général :

- retourner un booléen qui prévient globalement le client de la réussite / échec du traitement serveur ;
- ajouter un paramètre en sortie (en TRANSACT SQL: `@message varchar(255) OUTPUT`) qui renverra un message d'erreur fonctionnel : 'Utilisateur déjà créé', 'Livre emprunté', 'Solde insuffisant' etc. Les messages d'erreurs systèmes ne doivent pas être affichés, et

---

<sup>5</sup> Séance « *Utiliser un middleware d'accès aux données* »

ne devraient même pas être renvoyés par les procédures stockées dans l'application finale. (Comme ils sont utiles à la mise au point, ils peuvent être intégrés dans la version de test).

```
create PROCEDURE userExist @name varchar(30), @mdp varchar(20)
AS
BEGIN
    declare @nb int;
    declare @succes bit=0;

    -- TODO : valider Le nom et Le mdp (règles de gestion ?)
    begin try
        select @nb = count (*)
        from utilisateur
        where name = @name and mdp = @mdp;

        if @nb = 1
            set @succes = 1;
    end try
    begin catch
        print error_message();           -- en cours mise au point seulement
        set @succes = -1 ;               -- indique une erreur système sans précision
    end catch
    return @succes;
END
go
```

### 3.3.4 Ce qu'il vaut mieux éviter

Construire dynamiquement des requêtes en concaténant une commande SQL avec des paramètres en entrée :

- dans les cas où cette approche serait indispensable (ou tout au moins pratiquée dans votre entreprise), il faudra être strict sur la validation et le filtrage des paramètres en entrée (voir paragraphe suivant : valider ou échapper les données en entrée), si l'on veut éviter les injections SQL ;
- dans un audit sécurité de procédures stockées SQL Server, on vérifie systématiquement les outils qui permettent de lancer des requêtes SQL construites dynamiquement : instruction `exec` et procédure système `sp_execute`, car ces outils sont source de vulnérabilités.

## 3.4 VALIDER OU « ECHAPPER » LES DONNEES EN ENTREE

La vérification systématique des entrées pour éviter les injections SQL est absolument indispensable si l'on tolère le SQL dynamique dans nos procédures stockées (déconseillé).

Elle est recommandée dans tous les cas, pour éliminer d'autres injections muettes qui n'auront pas d'effet immédiat sur le comportement de notre code, mais attaqueront d'autres couches de l'application : par exemple, les pages web.

Le paramètre *name* de l'utilisateur Dupont :

*Dupont*<script>alert('coucou') ;</script>

sera stocké dans la table utilisateur comme une simple donnée, mais il injectera ensuite du code JavaScript dans la page HTML qui affichera les données de Dupont.

Et le script pourrait être bien pire !

Sécuriser l'accès et l'utilisation de la base de données

Afpa © 2014 – Section Tertiaire Informatique – Filière « Etude et développement »

### 3.4.1 Valider les données en entrée par une liste blanche

Procéder par liste noire (*black list*) en filtrant les caractères et séquences interdits est souvent un combat perdu d'avance.

Il est toujours préférable de travailler par liste blanche (*white list*), en se basant sur les règles de gestion connues : longueur, type, syntaxe des entrées valides.

C'est une stratégie d'acceptation des bonnes valeurs, qui rejette toute autre entrée, plutôt que d'essayer d'assainir les données potentiellement hostiles.

Si vous attendez un code postal, vérifiez que c'est bien un code postal (type, longueur et syntaxe au format US). Si ce n'est pas le cas, refusez-le, en retournant une chaîne vide :

```
public String isPostcode(String postcode)
{
    return (postcode != null &&
        Pattern.matches("^((2|8|9)\\d{2})|((02|08|09)\\d{2})|([1-9]\\d{3})$", postcode)) ?
        postcode : "";
}
```

A noter que OWASP's ESAPI a une librairie extensible de routines de validation *whitelist* des entrées : [https://www.owasp.org/index.php/Positive\\_security\\_model](https://www.owasp.org/index.php/Positive_security_model)

### 3.4.2 Méthode d'échappement des données (*Escape*)

Elle consiste à retirer les séquences d'injection des entrées.

Si vous filtrez les caractères spéciaux, utilisez la syntaxe « d'échappement » spécifique à votre interpréteur.

A noter que OWASP's ESAPI fournit également des routines d'échappement.

Cette méthode est dans la plupart des cas, déconseillée : le but n'est pas de vouloir à tout prix comprendre la demande du client ; il vaut mieux refuser ce qui n'a aucun sens pour le serveur.

## 3.5 TROISIEME PARADE : LA COUCHE DE PERSISTANCE (ORM)

C'est une méthode alternative aux procédures stockées, qui évite également au code client d'appeler l'interpréteur SQL, via JDBC.

En Java ou .NET, on peut utiliser une couche de persistance de données (ORM : *Object Relational Mapping*) qui permet d'accéder à la base de données via des objets spécialisés, les « entités », qui sont associées (*mapped*) aux tables de même nom.

Cette couche de persistance évite en principe les injections SQL, mais il reste recommandé de valider les entrées.

(Cette technique sera vue dans la partie N Tiers de la formation, en Java JEE).

## 4. SEPARER ET LIMITER LES PRIVILEGES DES UTILISATEURS

Il faut respecter ce principe du développement sécurisé lorsque l'on se connecte à une base de données (comme d'ailleurs aux autres systèmes de back office : serveur web, serveur applicatif etc.)

Cela n'évite pas en tant que telle l'injection SQL, mais limitera ses exploitations possibles.

Sécuriser l'accès et l'utilisation de la base de données

Afpa © 2014 – Section Tertiaire Informatique – Filière « Etude et développement »



Pour organiser sa stratégie de sécurité, il faut distinguer deux cas :

- les utilisateurs authentifiés de l'entreprise (salariés, partenaire de confiance etc.) qui ont droit en tant que tels à certaines ressources bien délimitées ;
- et les clients sans autorisation particulière (équivalent du compte anonyme en web) :

#### 4.1 GERER LES UTILISATEURS AUTHENTIFIES

Rappelons que la première précaution en sécurité informatique est de ne pas s'improviser spécialiste en sécurité ou en cryptographie : une stratégie de sécurité doit toujours s'appuyer sur des mécanismes existants et éprouvés.

Commençons par lister des erreurs que l'on trouve encore dans des applications installées :

##### 4.1.1 A éviter : protéger l'application cliente par un mot de passe local

- Si le mot de passe est stocké en dur dans le code du client, ou dans un fichier de configuration, il sera facile à casser : par un décompilateur, voir un simple *dump* du fichier *jar* avec Notepad++. L'utilisateur malveillant peut même *cracker* le code client, en insérant un branchement pour sauter la vérification du mot de passe.
- Variante de l'erreur :
  - stocker côté client, le compte et le mot de passe d'une connexion de service.
  - une fois que l'utilisateur s'est connecté en saisissant le mot de passe local, l'application utilise la même connexion de service pour toutes ses requêtes au serveur SQL
  - cette variante aggrave la compromission du mot de passe local, en donnant une large surface d'attaque : la connexion de service a des permissions sur l'ensemble des tables, des vues, des procédures stockées de tous les utilisateurs de l'application.
- Une protection illusoire : crypter le mot de passe local :
  - la vérification du mot de passe client peut de toute façon être *crackée* : on n'utilise plus le mot de passe local, on s'arrange pour que le code n'en tienne pas compte ;
  - on peut aussi développer un client malveillant, en réutilisant le compte et le mot de passe de la connexion SQL stockés localement.

##### 4.1.2 A éviter : une connexion de service pour tous les utilisateurs

Une erreur moins grossière consiste à bien valider la connexion côté serveur, mais à n'utiliser qu'une seule connexion pour tous les utilisateurs (la « connexion de service »).

L'application est plus robuste que dans le cas précédent, puisqu'aucune donnée confidentielle n'est stockée côté client. Mais elle reste vulnérable à une attaque réseau, du type « *Man in the middle* », où l'attaquant intercepte le compte et le mot de passe, en écoutant le réseau entre le client et le serveur.

En cas d'attaque réussie, les possibilités d'escalade sont les mêmes : accès à toutes les ressources autorisées pour la connexion de service.

##### 4.1.3 Ce qu'il faut faire : utiliser des connexions distinctes avec des droits limités

En partant du principe réaliste que rien n'est inattaquable, il faut :

Sécuriser l'accès et l'utilisation de la base de données

- éviter de travailler avec une connexion de service unique, dotée des droits de plusieurs utilisateurs ;
- toujours envoyer les requêtes avec la connexion de l'utilisateur réellement authentifié ;
- chaque utilisateur, ou groupe d'utilisateurs (« rôle » en SQL Server) recevra le minimum de permissions sur les objets de la base de données, afin d'éviter l'escalade en cas d'attaque réussie.

#### 4.1.4 Une bonne mise en œuvre : droit d'exécution sur les procédures stockées, lecture sur les tables et les vues

- Toute action délicate (insertion, modification, suppression) qui peut mettre en danger l'intégrité de la base et exige la mise en œuvre de transaction, sera effectuée par une procédure stockée.
- Chaque utilisateur aura uniquement les droits d'exécution sur les procédures stockées nécessaires, mais n'aura pas de permissions en *Insert*, *Delete*, *Update* sur les tables concernées.
- Cette mise en œuvre est un nouvel exemple de défense en profondeur : il faut nécessairement passer par les procédures stockées AVANT d'accéder aux tables, en modification.
- En cas d'attaque réussie, cette mise en œuvre limite la portée de l'attaque et les possibilités d'escalade : l'attaquant qui est parvenu à usurper l'identité d'un utilisateur légitime pourra faire indument certains traitements par les procédures stockées, mais ne pourra pas faire n'importe quoi sur les tables ;
- Seules les lectures seront effectuées directement (par des appels JDBC) : les utilisateurs auront les droits minimum en lecture sur les tables et les vues autorisées.

## 4.2 GERER LES CLIENTS EXTERNES

Comment gérer un client externe au système d'information de l'entreprise : par exemple un utilisateur de site de vente en ligne, qui peut uniquement s'inscrire ou se connecter sur le site, gérer son profil et passer des commandes ?

Le site doit fournir une fonctionnalité d'Authentification pour les clients externes, afin de protéger données personnelles, profil et historique des commandes.

Mais de façon évidente, un utilisateur web n'a aucun rapport avec un utilisateur interne du système d'information et ne doit recevoir aucun droit ni permission sur le SGBD, ni même être un utilisateur du SGBD.

C'est seulement dans ce cas qu'il faudra intégrer ou développer dans le site, un mécanisme d'authentification applicatif :

- Les utilisateurs du site web seront stockés sur le serveur, dans une table *Utilisateur*, avec leur alias et leur mot de passe cryptés ;
- Les alias et les mots de passe seront cryptés côté client, avant l'appel des procédures stockées, et seront envoyés cryptés sur le réseau ;
- Le site web utilisera une connexion de service pourvue de droits minimaux : par exemple, le droit d'exécution sur les procédures stockées d'inscription d'un nouveau client, d'authentification d'un client existant etc.

Sécuriser l'accès et l'utilisation de la base de données

Afpa © 2014 – Section Tertiaire Informatique – Filière « Etude et développement »

## 5. CONCLUSION

Dans cette séance, nous avons vu comment :

- sécuriser l'utilisation de la base, en se protégeant contre les injections SQL ;
- limiter les conséquences d'une attaque, en sécurisant l'accès à la base de données, par de bonnes pratiques d'authentification et d'attribution des droits.

Cette séance s'articule avec la séance « *sécuriser l'interface utilisateur* » : les deux séances montrent comment organiser une défense en profondeur, dans le cadre d'un projet client / serveur classique, 2 Tiers.

Les points suivants seront précisés par la suite :

- nous avons supposé que les comptes utilisateur étaient bien configurés, avec des mots de passe suffisamment complexes etc.

Dans la séquence « *Mettre en place une base de données* », nous donnerons des indications pratiques pour sécuriser le SGBD ;

- il y a d'autres attaques sur les bases de données : des attaques de type *TOCTTOU* (*Time of check to time of use*) qui exploite le parallélisme des requêtes, peuvent être évitées par une bonne mise en œuvre des transactions (voir séance « *Sécuriser la base de données à l'aide de transactions et de déclencheurs* »).