**Etat de l’art**

* **Analyse**
* **Specifier comment doit etre la bibliotheque a la fin**

**Contexte**

Les applications Web occupant une place, de plus en plus importante au sein des organisations. La plupart des organisations dépendent d’elles en partant de la productivité de leurs employés au renforcement de leur domaine d’activités en passant par l’analyse des données qu’elles fournissent. De nos jours, les attaquants comprennent ô combien critiques ces applications sont, et tentent de plus en plus à les compromettre.

Cependant, il est possible de ces prévenir contre ces attaques mais les organisations n’arrivent pas très souvent à prendre en charge la sécurité de leurs applications ou n’en font pas une priorité. Cela est principalement dû aux raisons suivantes :

* La nécessité d’un développement et d’un déploiement rapide de ces applications
* La non-application de bonnes pratiques de sécurité lors du développement des applications
* Entre autres raisons

Ainsi, la plupart des applications développées aujourd’hui plusieurs failles de sécurité.

1. **SQL Injection**
   1. Definition

Une attaque par injection SQL cible les applications utilisant des systèmes de Gestion de base de données.

Lors d’une attaque par injection SQL, l’attaquant fournit un/des segments malicieux de code SQL en tant qu’input transformant la requête SQL initiale en une autre souvent dangereuse. Les attaques par injection SQL sont le plus souvent utilisées pour dérober des informations sensibles de la base de données.

Par une injection SQL, l’attaquant, selon ses capacités peut obtenir des informations confidentielles ou même arriver à modifier ces informations.

* 1. Causes

La cause principale d’une injection SQL est le fait qu’une application accepte des données provenant de sources non sures comme l’utilisateur par exemple (input) ou le navigateur de l’utilisateur (cookies ou headers des requêtes http)

* 1. Vecteurs principaux
* http POST
* http GET
* Cookies
  1. Préventions
     1. Input validation

Une première défense est de valider les données fournies à l’application en mettant assez de contrainte sur les inputs, préciser éventuellement la longueur maximale, la longueur minimale, le type. Ces données peuvent provenir de champs de formulaire, de cookies, d’APIS externe ou encore d’un serveur proxy. Il faut se dire que toute donnée fournie à l’application est dangereuse. Pour s’assurer qu’un certain format est respecté, on peut utiliser les regex.

La validation des données saisies est faite pour s’assurer que seules les données correctement formées sont acceptées par le système, ce qui empêche la persistence des données mal formées et éventuellement dangereuses pour la base de données. La validation des entrées doit se faire le plus tôt possible. Cela permet de détecter les données malicieuses avant même qu’elles ne parviennent à l’application. Cette validation doit être faite à la fois côté client et côté serveur pour se protéger pour assurer le maximum de sécurité.

Plusieurs validations doivent être effectuées :

* Validation de la source des inputs : il faut aussi valider la source de l’input. Si l’input est attendu d’une requête GET par exemple, ne pas accepter un input provenant d’une requête POST.
* validation des types des données : s’assurer que les données saisies sont du type requis par l’application : on peut faire une conversion vers le type requis pour s’assurer que c’est le cas (Integer.parseInt() par exemple)
* validation de la taille des données saisies : préciser des valeurs minimales et maximales pour les données numériques et les dates, longueurs minimale et maximale pour les chaines de caractères. Peut être fait avec les attributs min et max, minLength et maxLength de l’élément input côté client et avec l’utilisation de regex côté serveur.
* validation des formats des données : la meilleure façon de faire cette validation est l’utilisation des regex. De même, il faut privilégier le white listing au black listing : n’accepter que les données respectant un certain format. Utilisation des Regex
* sanitisation : il s’agit de rendre les données saisies potentiellement dangereux pour les interpréteurs SQL non interprétables par un processus d’échappement. Pour ce faire, il est recommandé d’utiliser MySQL Codec ou Oracle Codec.

En résumé, pour toute donnée provenant de l’utilisateur, il faut (éventuellement):

* définir l’ensemble des valeurs acceptées avec l’aide de regex
* caster la (les) donnée(s) pour s’assurer qu’elles sont du type demandé
* définir les longueurs maximale et minimale
* sanitiser
* définir un regex de validation
* définir la source des données attendues
  + 1. Requêtes préparées

L’utilisation de requêtes préparées est la forme de protection la plus effective contre les injections SQL. En effet les requêtes préparées sont pré compilées par le SGBD c’est-à-dire que le SGBD sait déjà ce que doit faire cette requête et peuvent contenir des paramètres qui leur seront fournis lors de l’exécution. Lorsque ces paramètres sont reçus, ils sont placés comme valeurs dans la requête précompilée et ne sont jamais interprétés en tant que code SQL. Ainsi, la pré-compilation, (une compilation faite dès le début et une seule fois) couplée à la prise en compte de paramètres au moment de l'exécution, permet d'améliorer considérablement les performances de requêtes paramétrées destinées à être exécutées plusieurs fois et protègent contre les injections SQL.

A parameterized query is a query in which placeholders are used for parameters and the parameter values are supplied at execution time. The most important reason to use parameterized queries is to avoid SQL injection attacks.

* + 1. Défenses supplémentaires
       1. Réduire la surface d’attaques en donnant le moins de privilèges

Utiliser plusieurs utilisateurs de Base de données dans l’application. Par exemple pour la connexion, on a juste besoin d’un privilège de lecture de la base de données, on peut créer un utilisateur avec juste ce privilège et utiliser l’*entity manager\** avec la persistence unit utilisant une datasource avec cet utilisateur dans l’Ejb pour gérer la connexion. Pour l’inscription par contre on a besoin de pouvoir lire et insérer dans la base, on crée un utilisateur avec ces privilèges et ainsi de suite. Séparer les privilèges en plusieurs utlisateurs.

* + - 1. Crypter toutes les données sensibles

En matière de sécurité, il faut toujours considérer les pires des cas : vol d’informations, suppression, modification et de données. De ce fait, il est important de

* crypter toutes les données sensibles présentes en base de données. Ce cryptage doit aussi être fait sur les données enregistrées dans des fichiers ou encore sous forme de cookies au niveau du navigateur de l’utilisateur. Il faut crypter aussi les données sensibles en transit en utlisant un algorithme de cryptage avec sel.
* Logger toutes les requêtes arrivant à la BD : prévoir un fichier pour chaque type de requête (Create, Read, Update, Delete) et faire un audit périodique de ces fichiers.
* Prévoir au moins une base de données de réplication et faire périodiquement la copie des données présentes en base de données ainsi que leur conservation dans des endroits sécurisés de préférence physique.
  + - 1. Se méfier des headers http

Les en-têtes HTTP sont envoyés au début des requêtes HTTP et des réponses HTTP. L’ application Web doit s'assurer qu'elle ne base aucune décision de sécurité sur les informations contenues dans les en-têtes HTTP car il est facile pour un attaquant de manipuler l'en-tête. Par exemple, le champ referer dans l'en-tête contient l'URL de la page Web d'où provient la demande. Ne prenez aucune décision de sécurité basée sur la valeur du champ référant, par exemple, pour vérifier si la **requête** provient d'une page générée par l'application Web, car le champ est facilement falsifié.

1. **Cross-site Scripting**
   1. Definition

Le Cross-Site Scripting abrégé XSS est une faille de sécurité des applications utilisant un navigateur Web permettant à un attaquant d’injecter du code malveillant exécutable par ce navigateur. Il s’agit donc d’un cas d’injection de code à un interpréteur qui, à la différence des autres cas d’injection de code qui cible le serveur, cible le navigateur de l’utilisateur.

Le code injecté consiste généralement en du script JavaScript exécuté par le navigateur victime qui le prend pour du code légitime provenant de la même origine que le contenu html affiché. Ce script peut ainsi faire de nombreuses choses, allant d’une simple redirection à une usurpation de l’identité d’un utilisateur par vol des cookies de session. Et dans ce dernier cas, si l’utilisateur en question a des droits d’administrateur sur une certaine application, l’attaquant peut compromettre entièrement ladite application.

On note trois types d’attaques XSS :

* Reflected XSS
* Stored XSS
* DOM XSS

Lors d’une attaque par injection SQL, l’attaquant fournit un/des segments malicieux de code SQL en tant qu’input transformant la requête SQL initiale en une autre souvent dangereuse. Les attaques par injection SQL sont utilisées pour dérober des informations sensibles de la base de données.

Par une injection SQL, l’attaquant, selon ses capacités peut obtenir des informations confidentielles ou même arriver à modifier ces informations.

* 1. Causes

Étant donné que les sites Web dynamiques reposent sur l'entrée de l'utilisateur, un utilisateur malveillant peut entrer un script malveillant dans la page en le masquant dans des demandes légitimes.

Le cross site scripting (XSS) affecte essentiellement des sites qui acceptent des données en provenance de l'utilisateur sans en contrôler correctement la conformité

La cause principale d’une injection SQL est le fait qu’une application accepte des données provenant de sources non sures comme l’utilisateur par exemple (input) ou le navigateur de l’utilisateur (cookies ou headers des requêtes http)

* 1. Vecteurs principaux
* http POST
* http GET
* Cookies
  1. Préventions
     1. Input validation

Une première défense est de valider les données fournies à l’application en mettant assez de contrainte sur les inputs, préciser éventuellement la longueur maximale, la longueur minimale, le type. Ces données peuvent provenir de champs de formulaire, de cookies, d’APIS externe ou encore d’un serveur proxy. Il faut se dire que toute donnée fournie à l’application est dangereuse. Pour s’assurer qu’un certain format est respecté, on peut utiliser les regex.

La validation des données saisies est faite pour s’assurer que seules les données correctement formées sont acceptées par le système, ce qui empêche la persistence des données mal formées et éventuellement dangereuses pour la base de données. La validation des entrées doit se faire le plus tôt possible. Cela permet de détecter les données malicieuses avant même qu’elles ne parviennent à l’application. Cette validation doit être faite à la fois côté client et côté serveur pour se protéger pour assurer le maximum de sécurité.

Plusieurs validations doivent être effectuées :

* Validation de la source des inputs : il faut aussi valider la source de l’input. Si l’input est attendu d’une requête GET par exemple, ne pas accepter un input provenant d’une requête POST.
* validation des types des données : s’assurer que les données saisies sont du type requis par l’application : on peut faire une conversion vers le type requis pour s’assurer que c’est le cas (Integer.parseInt() par exemple)
* validation de la taille des données saisies : préciser des valeurs minimales et maximales pour les données numériques et les dates, longueurs minimale et maximale pour les chaines de caractères. Peut être fait avec les attributs min et max, minLength et maxLength de l’élément input côté client et avec l’utilisation de regex côté serveur.
* validation des formats des données : la meilleure façon de faire cette validation est l’utilisation des regex. De même, il faut privilégier le white listing au black listing : n’accepter que les données respectant un certain format. Utilisation des Regex
* sanitisation : il s’agit de rendre les données saisies potentiellement dangereux pour les interpréteurs SQL non interprétables par un processus d’échappement. Pour ce faire, il est recommandé d’utiliser MySQL Codec ou Oracle Codec.

En résumé, pour toute donnée provenant de l’utilisateur, il faut (éventuellement):

* définir l’ensemble des valeurs acceptées avec l’aide de regex
* caster la (les) donnée(s) pour s’assurer qu’elles sont du type demandé
* définir les longueurs maximale et minimale
* sanitiser
* définir un regex de validation
* définir la source des données attendues
  + 1. Requêtes préparées

L’utilisation de requêtes préparées est la forme de protection la plus effective contre les injections SQL. En effet les requêtes préparées sont pré compilées par le SGBD c’est-à-dire que le SGBD sait déjà ce que doit faire cette requête et peuvent contenir des paramètres qui leur seront fournis lors de l’exécution. Lorsque ces paramètres sont reçus, ils sont placés comme valeurs dans la requête précompilée et ne sont jamais interprétés en tant que code SQL. Ainsi, la pré-compilation, (une compilation faite dès le début et une seule fois) couplée à la prise en compte de paramètres au moment de l'exécution, permet d'améliorer considérablement les performances de requêtes paramétrées destinées à être exécutées plusieurs fois et protègent contre les injections SQL.

A parameterized query is a query in which placeholders are used for parameters and the parameter values are supplied at execution time. The most important reason to use parameterized queries is to avoid SQL injection attacks.

* + 1. Défenses supplémentaires
       1. Réduire la surface d’attaques en donnant le moins de privilèges

Utiliser plusieurs utilisateurs de Base de données dans l’application. Par exemple pour la connexion, on a juste besoin d’un privilège de lecture de la base de données, on peut créer un utilisateur avec juste ce privilège et utiliser l’*entity manager\** avec la persistence unit utilisant une datasource avec cet utilisateur dans l’Ejb pour gérer la connexion. Pour l’inscription par contre on a besoin de pouvoir lire et insérer dans la base, on crée un utilisateur avec ces privilèges et ainsi de suite. Séparer les privilèges en plusieurs utlisateurs.

* + - 1. Crypter toutes les données sensibles

En matière de sécurité, il faut toujours considérer les pires des cas : vol d’informations, suppression, modification et de données. De ce fait, il est important de

* crypter toutes les données sensibles présentes en base de données. Ce cryptage doit aussi être fait sur les données enregistrées dans des fichiers ou encore sous forme de cookies au niveau du navigateur de l’utilisateur. Il faut crypter aussi les données sensibles en transit en utlisant un algorithme de cryptage avec sel.
* Logger toutes les requêtes arrivant à la BD : prévoir un fichier pour chaque type de requête (Create, Read, Update, Delete) et faire un audit périodique de ces fichiers.
* Prévoir au moins une base de données de réplication et faire périodiquement la copie des données présentes en base de données ainsi que leur conservation dans des endroits sécurisés de préférence physique.
  + - 1. Se méfier des headers http

Les en-têtes HTTP sont envoyés au début des requêtes HTTP et des réponses HTTP. L’ application Web doit s'assurer qu'elle ne base aucune décision de sécurité sur les informations contenues dans les en-têtes HTTP car il est facile pour un attaquant de manipuler l'en-tête. Par exemple, le champ referer dans l'en-tête contient l'URL de la page Web d'où provient la demande. Ne prenez aucune décision de sécurité basée sur la valeur du champ référant, par exemple, pour vérifier si la **requête** provient d'une page générée par l'application Web, car le champ est facilement falsifié.