UNIVERSITÉ DU QUÉBEC À MONTRÉAL

PROTECTION AUTOMATIQUE DES APPLICATIONS WEB CONTRE L'ATTAQUE PAR INJECTION SQL

MÉMOIRE
PRÉSENTÉ
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE

PAR
MOHAMED YASSIN

FÉVRIER 2014

REMERCIEMENTS

Mes meilleurs remerciements à mon directeur de recherche Monsieur Guy Bégin pour ses conseils, sa patience et sa disponibilité. Je le remercie également pour ses remarques pertinentes qui ont contribué à améliorer ce travail.

Je tiens à remercier mes professeurs du département d'informatique de l'Université du Québec à Montréal qui, tout au long de la maîtrise, m'ont enseigné avec brio l'informatique.

Je voudrais ajouter un gros merci à ma femme Zeinab qui m'a soutenu pendant tout ce temps. Je remercie aussi mes parents et ma famille pour leur encouragement.

Merci à mes amis et à tous ceux qui ont participé de près ou de loin à l'accomplissement de ce travail.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

Apache[1] un serveur HTTP qui a été développé au sein de la fondation Apache

CPU unité centrale de traitement

HTTP[1] un protocole de communication client-serveur

HTTPS[1] la variante du HTTP sécurisée par l'usage des protocoles SSL ou TLS

IIS[1] le serveur Web (FTP, SMTP, HTTP, etc.) de Microsoft

JDBC[1] une interface de programmation pour les programmes Java

Jeton[1] une entité (ou unité) lexicale dans le cadre de l'analyse lexicale

J2EE[1] la plateforme du Java pour développer et déployer des applications Web

MS SQL[1] un système de gestion de base de données développé par Microsoft

MySQL[1] un système de gestion de base de données (licence GPL)

Pare-feu[1] un logiciel qui vérifie que la politique de sécurité est respectée

PHP[1] un langage de scripts libre utilisé pour produire des pages Web

SGBD système de gestion de base de données

SQL[1] langage de manipulation des bases de données relationnelles

SQLIA attaque par injection SQL

TimeStamp[1] représente la date et le temps de l'occurrence d'un paquet

URL[1] adresse Web sous la forme d'un localisateur uniforme de ressource

.NET[1] la plateforme Microsoft qui rend les applications portables sur Internet

---

1. http ://www.wikipedia.org/

LISTE DES ALGORITHMES

LISTE DES LISTAGES

RÉSUMÉ

Dans ce travail, nous proposons une approche de détection d'anomalies contre les attaques par injection SQL et qui n'exige ni lecture, ni analyse, ni modification du code source de l'application Web à protéger. Cette approche offre une nouvelle méthode de corrélation d'évènements qui se base sur une logique temporelle et sur une mesure de similarité entre les demandes HTTP et les requêtes SQL. Elle est capable d'analyser, de corréler plusieurs événements dans le temps et de prendre des contremesures.

Nous présentons également l'implémentation et l'évaluation d'un système de détection d'intrusions fondé sur l'approche proposée, nommé SQLIA-IDS, qui surveille une application Web dans le but de fournir un outil de détection d'attaques visant les bases de données de cette application. Ce système se déroule en deux phases : la phase d'apprentissage qui permet de créer le comportement normal de l'application Web en modélisant les requêtes SQL normales et la phase de protection qui consiste à vérifier lors de l'exécution que les requêtes SQL respectent le comportement normal de l'application Web.

Mots clés : application Web, demande HTTP, requête SQL, attaque par injection SQL, détection d'intrusions, SQLIA-IDS.

INTRODUCTION

Les applications déployées dans les environnements Web se sont fortement développées la dernière décennie, surtout avec l'avènement du Web 2.0 qui a autorisé grand nombre d'utilisateurs d'accéder aux données et aux ressources informatiques par les navigateurs Web. Cette ouverture a considérablement augmenté la surface d'attaque et la vulnérabilité des serveurs et des applications Web.

Les techniques d'attaques sur les applications Web ont particulièrement évolué avec la naissance des outils d'injection automatique sur le Web comme SQLPowerInjector [Lar11], rendant le risque plus élevé et l'exploitation des failles de sécurité plus automatisée. Les attaques Web sont alors fréquentes, mais la plupart des entreprises n'ont pas sécurisé leurs sites Web.

Malgré l'existence des dizaines types d'attaques sur le web, notre recherche porte sur la protection contre l'attaque par injection SQL, plus communément connue sous le nom SQLIA. D'après le projet OWASP (Open Web Application Security Project), la SQLIA est classée en première parmi les dix risques de sécurité applicatifs web les plus critiques [OWA10]. Le projet WASC (The Web Application Security Consortium) [JCmA12], qui fournit des statistiques sur les attaques Web en temps réel, a classé aussi l'attaque par injection SQL dans la première position lorsque nous avons consulté le site Web de ce projet.

L'attaque par injection SQL représente une faille de sécurité très dangereuse pour les applications Web. Le comportement normal de l'application attaquée peut alors se changer dramatiquement et l'attaquant peut contourner les

contrôles d'accès mis en place et accéder à cette application comme un administrateur. Dans le pire des cas, un pirate peut manipuler complètement une base de données [Cer10]. La protection des applications web contre les attaques par injection SQL devient alors une problématique fondamentale pour les individus et les entreprises qui accèdent à des applications Web dynamiques.

Problématique

En général, les développeurs des pages Web construisent des requêtes SQL dynamiques à partir des entrées d'utilisateur sans valider s'ils contiennent des caractères suspects. Un attaquant peut donc exploiter cette faille de sécurité et injecter un code SQL malveillant afin de s'exécuter sur la base de données. L'attaque par injection SQL peut réussir compte tenu des facteurs suivants :

- Les développeurs continuent à implanter des applications vulnérables à cause de leur manque d'expérience ou de l'absence de temps pour pouvoir appliquer les bonnes pratiques de codage sécurisé.
- Les pare-feu et les systèmes classiques de détection d'intrusions qui filtrent souvent le trafic réseau (IP, TCP, UDP, etc.) ne peuvent pas protéger une base de données contre les attaques par injection SQL [Hol04].
- L'installation d'un certificat SSL n'empêche pas non plus ce type d'attaques.
- Le balayage automatisé de vulnérabilités web [JFM08] génère des rapports de sécurité sans être capable de prévenir les attaques par injection après le déploiement des applications Web.

Objectifs

Nous tentons dans ce projet de recherche de développer notre propre contre-mesure en considérant l'application comme une boîte noire. L'avantage est de

protéger n'importe quelle application écrite avec n'importe quel langage de programmation. Le système proposé doit donc être portable ou au moins facilement adaptable à tous les environnements de développement.

L'objectif principal de notre étude est de réaliser une protection automatique et robuste contre les attaques par injection SQL. Pour ce faire, nous proposons de développer un système de détection d'intrusions. L'idée principale est de construire les modèles de référence (requêtes normales) par analyse dynamique sans accéder au code source de l'application Web à protéger. Ce système repose sur plusieurs techniques pour assurer une protection automatique lors de l'exécution des requêtes dynamiques avec un impact minimal sur la performance des applications Web, des serveurs Web et des systèmes de gestion de bases de données.

Pour atteindre notre objectif, nous allons tout d'abord étudier en profondeur les techniques de SQLIA. Ensuite, nous allons analyser la protection relative au code source et quelques contremesures avancées proposées par d'autres chercheurs. Enfin, nous allons proposer notre propre approche face aux SQLIAs. Afin de valider pratiquement l'approche proposée, nous allons implémenter le système SQLIA-IDS qui, à son tour, sera évalué selon des résultats expérimentaux.

Structure du mémoire

Ce document présente notre recherche et comporte six chapitres. Les trois premiers chapitres présentent notre contexte du travail, les techniques d'attaques par injection les plus populaires et l'état de l'art. Les trois derniers chapitres présentent notre propre contribution.

Dans le premier chapitre, nous allons faire une introduction à la sécurité informatique. Nous allons nous intéresser particulièrement aux applications Web et à la détection d'intrusions.

Dans le deuxième chapitre, nous allons détailler les types et les préjudices d'attaques par injection SQL.

Dans le troisième chapitre, nous allons discuter certaines techniques pour améliorer le code des applications Web et certains projets de recherche ayant mis en œuvre des contremesures avancées de SQLIA.

Dans le quatrième chapitre, nous allons proposer notre propre contremesure qui se fonde sur la détection d'anomalies et sur une méthode de corrélation entre le trafic HTTP et le trafic SQL.

Dans le cinquième chapitre, nous allons décrire la mise en œuvre de l'approche proposée, SQLIA-IDS, qui nous permettra d'évaluer pratiquement les algorithmes et les méthodes implémentés au sein de l'approche proposée.

Dans le sixième chapitre, nous allons présenter les expériences effectuées sur notre environnement de test ainsi que les résultats obtenus selon lesquels le système SQLIA-IDS sera évalué.

Finalement, nous allons conclure le mémoire en posant quelques perspectives pour continuer ce travail.

CHAPITRE I

CONTEXTE

Ce chapitre présente le contexte de notre mémoire dans lequel plusieurs concepts liés à la sécurité informatique, les applications Web et la détection d'intrusions sont brièvement présentés.

Nous introduisons tout d'abord quelques aspects de la sécurité informatique. Ensuite, nous présentons les applications Web avec un accent particulier sur leur architecture. Enfin, nous détaillons la détection d'intrusions.

1.1 Introduction à la sécurité informatique

La sécurité informatique consiste à mettre en place une politique pour assurer la sécurité du système informatique [Wik12b]. Les objectifs suivants déterminent les enjeux principaux de la sécurité informatique [BÉ10, Wik12b] :

- **Intégrité :** les données ne doivent pas être altérées par un utilisateur malveillant [BÉ10, Wik12b].
- **Confidentialité :** assurer que les données confidentielles ne soient accessibles, en lecture, que par les utilisateurs légitimes [BÉ10, Wik12b].
- **Disponibilité :** garantir que les ressources informatiques et les données soient toujours disponibles pour les utilisateurs légitimes [BÉ10, Wik12b].
- **Authentification :** l'autorisation à une entité (personne, ordinateur, etc.) qui

a le droit d'accéder à des ressources informatiques [BÉ10, Wik12b].

Les aspects suivants sont souvent utilisés lorsque nous parlons des attaques :

- **Préjudice :** « une perte ou un dommage possible dans un système informatique. » [BÉ10]
- **Vulnérabilité :** « une faiblesse dans un système qui pourrait être exploitée pour causer des préjudices. » [BÉ10]
- **Menace :** « ensemble de circonstances pouvant conduire à des préjudices. » [BÉ10]

1.2 Les applications Web

La majorité des nouveaux projets informatiques sont aujourd'hui des applications web qui sont devenues une solution universelle d'utilisation grâce à de nombreux avantages :

- Une application web est accessible par un navigateur Web sans déployer aucun autre logiciel sur le poste client et les mises à jour sont simplifiées puisque l'application est centralisée sur un serveur.
- La gratuité des serveurs web, comme le serveur HTTP de la fondation Apache (Apache) et le serveur Web de Microsoft (IIS), encourage les entreprises à mettre en place leurs applications Web afin d'augmenter leurs clientèles.
- La simplicité des plateformes, des outils et des langages de programmation permet aux ressources experts et inexperts de développer en mettant peu d'efforts toutes sortes d'applications Web.

Architecture des applications web

Souvent, une application web est basée sur une architecture client-serveur [Wik11a, Nga09] qui comprend un client Web, un serveur Web sur lequel

l'application web est installée, et un serveur de bases de données (figure 1.1). Ces composants communiquent entre eux comme suit :

- Le client Web envoie une demande HTTP à l'application Web qui lui retourne une page Web.
- L'application Web envoie une requête SQL au serveur de données qui lui retourne des données.

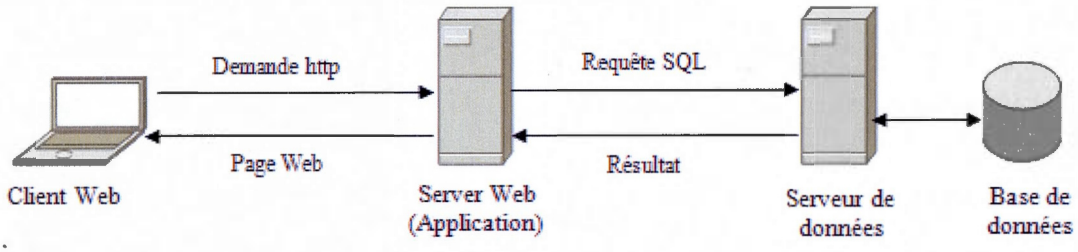

Figure 1.1 Architecture d'une application Web

Les données saisies par l'utilisateur sont envoyées à l'application web par les deux fonctions POST et GET [Gro11, HVO06] :

- « POST » , illustrée dans le listage 1.1, envoie au serveur Web les paires nom/valeur des entrées de l'utilisateur dans le corps du message de demande HTTP.
- « GET » , illustrée dans le listage 1.2, envoie au serveur Web une chaîne des paires nom/valeur et ajoute ensuite la chaîne de requête dans le corps du message de demande HTTP.

```
POST /Default.aspx HTTP/1.1
userID=admin
password=123
Accept: image/gif, image/x-xbitmap, image/jpeg, ...
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 ...
Host: localhost
Cookie: ASP.NET_SessionId=xshitibm0r1nlpawjvwfzn55
Connection: Keep-Alive
```

Listage 1.1 Exemple d'une demande HTTP en utilisant la méthode POST

```
GET /Default.aspx?userID=admin&password=123 HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, ...
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 ...
Host: localhost
Cookie: ASP.NET_SessionId=xshitibm0r1nlpawjvwfzn55
Connection: Keep-Alive
```

Listage 1.2 Exemple d'une demande HTTP en utilisant la méthode GET

La méthode POST est plus sécurisée que la méthode GET qui fait passer les variables par l'adresse d'une page Web (URL) ; ces variables peuvent être facilement détournées.

1.3 La détection d'intrusions

Une intrusion est toute tentative pour compromettre le comportement normal d'un système ou d'un réseau informatique [Deb05, Wik12a]. Comme il est difficile de prévoir toutes les intrusions lors du développement, les systèmes de détection d'intrusions ont été conçus pour leur faire face [Deb05].

Un système de détection d'intrusions (IDS) a pour rôle de surveiller et d'analyser le flux d'évènements dirigés vers un hôte (ou un réseau) pour détecter et journaliser ceux qui sont suspects [Deb05]. Un IDS est appelé IPS (système de prévention d'intrusions) si celui-ci réussit à prévenir les intrusions [Wik12a].

Certains termes sont souvent employés lorsque nous parlons de détection d'intrusions :

- **Faux positif** : une alerte levée par un IDS, mais qui ne correspond pas à une attaque réelle [Wik12a].
- **Faux négatif** : une attaque réelle qui n'a pas été détectée par un IDS [Wik12a].
- **Technique d'évasion** : c'est une méthode pour contourner les moyens de protection afin d'effectuer des attaques sans être détectées par les IDS [Wik12a].

1.3.1 Critères d'évaluation d'un système de détection d'intrusions

Les IDS sont généralement évalués suivant plusieurs critères [Deb05, CD12] :

- **Exhaustivité :** c'est la capacité de détecter les attaques connues et inconnues. Un IDS est exhaustif s'il fonctionne avec un taux minimal des faux négatifs [Deb05, CD12].
- **Précision :** un IDS est précis s'il détecte les attaques avec un taux minimal des faux positifs [Deb05, CD12].
- **Performance :** la performance d'un IDS est mesurée par sa consommation de ressources informatiques telles que l'unité centrale de traitement (CPU), la mémoire, etc. [Deb05].

Nous ajoutons aussi trois critères supplémentaires :

- **Facilité de mise en place :** le déploiement et la configuration d'un IDS doivent être faciles. Un bon IDS ne doit pas nécessiter des modifications majeures du système qu'il protège.
- **Portabilité**[1] **:** c'est la capacité d'un IDS à fonctionner plus ou moins sans problèmes dans différents environnements. Vu la difficulté d'avoir un IDS portable sur tous les environnements de développement au préalable, un bon IDS doit être aisément adaptable à tous les environnements.
- **Autonomie :** un IDS autonome est celui qui évolue automatiquement pour détecter les nouvelles attaques sans aucune intervention humaine.

1.3.2 Approches de détection d'intrusions

À l'heure actuelle, deux approches de détection d'intrusions sont répandues : l'approche par signature (misuse detection) et l'approche comportementale (anomaly detection) [GT07, Deb05, Wik12a].

L'approche par signature [GT07, Deb05, Wik12a, Saf]

Un IDS basé sur cette approche est équipé d'une base d'attaques (signatures) connues qui doit être mise à jour régulièrement par un administrateur ou un responsable de la sécurité. Lors de l'exécution, cet IDS surveille les actions qui visent l'entité à protéger puis les compare avec les scénarios de la base d'attaques. À cet égard, toute action qui n'est pas explicitement reconnue comme une attaque est acceptée. C'est pour cela que la précision de cette approche est bonne. Cette méthode ne consomme pas trop de ressources informatiques, mais elle comporte plusieurs faiblesses :

- Son exhaustivité dépend de la mise à jour régulière et de la qualité des scénarios

1. http ://fr.wikipedia.org/wiki/Portabilit%C3%A9_(informatique)

d'attaques. Cette approche est incapable d'identifier les nouvelles attaques.
- Son autonomie est faible puisqu'elle nécessite une analyse humaine et approfondie de chaque vulnérabilité pour pouvoir mettre à jour la base d'attaques, ce qui représente une tâche fastidieuse pour le responsable de la sécurité.
- La connaissance de la base d'attaques facilite la tâche d'un attaquant qui peut camoufler ses attaques à l'aide de techniques d'évasion.

L'approche comportementale [GT07, Deb05, Wik12a]

L'approche comportementale se divise en deux phases : la phase d'apprentissage durant laquelle les événements sont surveillés afin de modéliser le comportement normal ou valide du système cible et la phase de protection qui permet de détecter une intrusion en observant que les activités actuelles s'écartent des comportements attendus du système ou des utilisateurs.

L'avantage principal de cette approche est sa capacité de détecter automatiquement les nouvelles attaques et même les attaques imprévues puisque tout ce qui ne correspond pas à un comportement modélisé lors de la phase d'apprentissage est considéré comme suspect.

Cependant l'approche comportementale présente quelques inconvénients :
- D'une part, elle peut générer un grand nombre de fausses alertes à cause de la difficulté de couvrir tous les comportements normaux pendant l'apprentissage ; d'autre part, le comportement peut changer au fil du temps, ce qui nécessite parfois de reprendre l'apprentissage.
- Elle consomme une quantité considérable de ressources informatiques pour pouvoir mesurer les comportements normaux et les déviations.
- Elle ne permet pas toujours d'identifier les types d'attaques détectées, car elles

ne sont pas classifiées a priori.

1.3.3 Catégories de systèmes de détection d'intrusions

Il est possible de classer les IDS selon les sources d'information qu'ils surveillent [Deb05]. Certains IDS permettent d'analyser un réseau, d'autres un hôte (ou une application) et d'autres sont hybrides [Deb05].

- **Les HIDS :** un HIDS (Host Intrusion Detection System) est installé sur un hôte pour surveiller les activités (machine et utilisateur) d'un système (ou une application) et détecter celles qui sont malveillantes [Deb05, Wik13].
- **Les NIDS :** un NIDS (Network Intrusion Detection System) surveille les paquets circulant sur un réseau, et ensuite détecte les attaques contre ce réseau [Deb05, Wik13].
- **Les IDS hybrides :** ces IDS sont une combinaison entre les NIDS et HIDS visant à améliorer la précision et à minimiser les faux positifs [Deb05, Wik13].

Conclusion

Au cours de ce chapitre, nous avons introduit quelques concepts concernant la sécurité informatique et les applications Web et nous avons détaillé les différentes approches de détection d'intrusions. Dans le prochain chapitre, nous allons voir comment l'attaque par injection SQL (SQLIA) fonctionne et pourquoi elle réussit souvent à compromettre la sécurité des applications Web. Nous allons alors aborder les types et les préjudices de cette attaque.

CHAPITRE II

L'ATTAQUE PAR INJECTION SQL

Dans ce chapitre, nous détaillons et approfondissons l'attaque par injection SQL, communément appelée SQLIA. Nous commençons par expliquer comment cette attaque se déroule. Ensuite, nous discutons les mécanismes d'injection et les types de SQLIA (sections 2.2 et 2.3). Finalement, nous résumons les préjudices des SQLIAs.

2.1 Principe de l'attaque par injection SQL

La possibilité d'accéder aux applications Web à distance multiplie les points d'attaque potentiels pour les pirates qui veulent attaquer ces applications ainsi que leurs bases de données.

Le but essentiel du pirate est de changer la structure (ou la sémantique) des requêtes SQL pour qu'elles soient interprétées différemment de ce qui avait été prévu par le programmeur [OWA10, SWLL06]. Pour ce faire, ce pirate injecte des caractères dangereux[1] (', /, –, etc.) et des mots clés SQL (union, drop, etc.) dans les champs d'entrée. Intuitivement, si le programmeur ne valide pas les

1. Il s'agit d'un caractère qui peut être injecté dans un champ d'entrée dans le but de changer la structure d'une requête SQL dynamique pendant l'exécution.

entrées d'utilisateur avant de les employer dans des requêtes SQL dynamiques, alors l'attaque pourra réussir (figure 2.1).

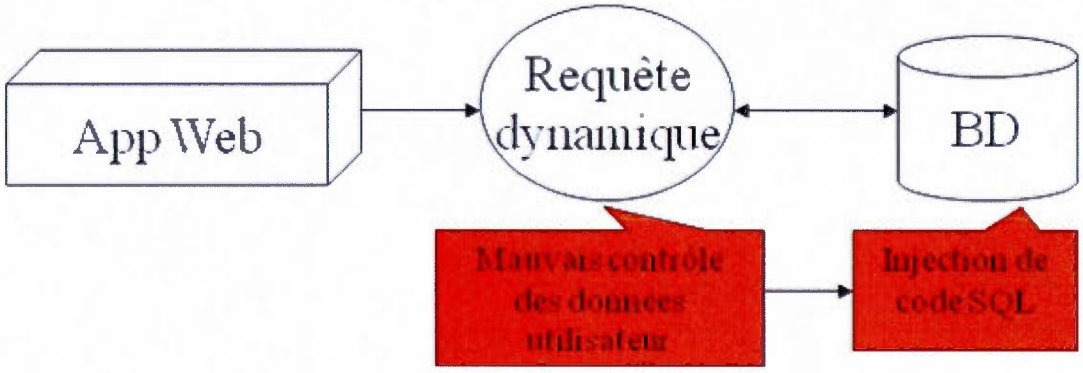

Figure 2.1 L'attaque par injection SQL

2.2 Les mécanismes d'injection

Il existe trois mécanismes d'injection [HVO06, Nga09] permettant d'exécuter un code SQL malveillant sur les bases de données d'une application Web.

Injection dans les entrées d'utilisateur [HVO06, Nga09]

Si le code source d'une application Web contient des requêtes SQL construites avec les entrées d'utilisateur, un pirate peut facilement saisir des caractères dangereux dans ces entrées pour monter une SQLIA. Les entrées d'utilisateur constituent les points d'attaques les plus exploitables par les pirates et les moins détectables par les pare-feu et les IDS.

Injection dans les témoins de connexion [HVO06, Nga09]

Les témoins de connexion (cookies) sont des fichiers placés sur le poste du client, associés à une application Web et contiennent des informations sur les préférences des utilisateurs, etc. Comme les entrées d'utilisateur, les cookies peuvent être exploitables par un attaquant lorsque l'application Web stocke les données dans ces cookies sans les chiffrer. Les nouvelles applications web chiffrent les contenus de cookies pour contrer les attaques provenant de ces derniers.

Injection dans les variables du serveur [HVO06, Nga09]

La connexion entre le client et le serveur Web se rompt fréquemment, car HTTP (ou HTTPS) est un protocole déconnecté. Pour identifier les clients, le serveur crée des variables, appelées variables de serveur, contenant des informations concernant les clients et qui sont extraites à partir des entêtes http. Un pirate peut donc injecter ses propres valeurs dans l'entête http par l'URL pour perpétrer une SQLIA qui se déclenche automatiquement lorsque le serveur communique les valeurs des variables du serveur avec la base de données aux fins de statistique.

2.3 Les types d'attaques par injection SQL

Les attaques par injection SQL sont bien expliquées et classifiées dans [HVO06]. Nous allons nous en inspirer pour expliciter les types d'attaques par injection SQL les plus populaires, à l'aide d'une requête dynamique.

Considérons la requête SQL dynamique dont le code source est présenté dans le listage 2.1 et qui correspond à une authentification d'un client à partir d'un formulaire tel qu'il est illustré par la figure 2.2. On y trouve deux champs d'entrée : « txtUsager » et « txtMotPasse ». Nous allons illustrer ensuite chaque

type de SQLIA en servant d'un exemple basé sur cette requête.

```
strQuery= "Select * from TblUsagers
Where Usager='" + txtUsager.text + "' and MotPasse='" + txtMotPasse.text + "' "
```

Listage 2.1 Exemple d'une requête SQL dynamique

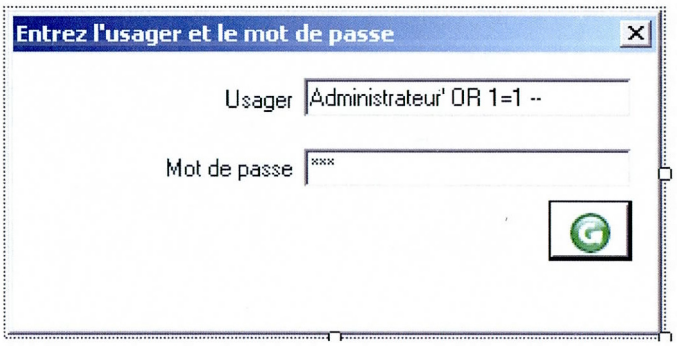

Figure 2.2 Formulaire d'authentification

2.3.1 Requête syntaxiquement incorrecte

Souvent, la première étape de l'attaquant est de collecter certaines informations nécessaires pour compromettre l'application cible. Les applications Web facilitent parfois cette étape en affichant des messages significatifs d'erreur générés par le SGBD. Cette vulnérabilité conduit l'attaquant à provoquer ces messages pour obtenir des informations confidentielles sur l'application Web, le serveur Web, le SGBD et la structure de données [HVO06, Nga09].

Par exemple : si le pirate saisit (') dans le champ d'usager et (123) dans le champ de mot de passe, la requête SQL du listage 2.2 sera envoyée à la base de données.

```
Select * from TblUsagers Where Usager='' and MotPasse='123'
```

Listage 2.2 Exemple d'une requête syntaxiquement incorrecte

Le pirate a donc pu provoquer le message d'erreur significatif, illustré dans le listage 2.3. Une simple analyse de ce message permet de savoir que le SGBD de l'application Web est Microsoft SQL Server (MS SQL) en se basant sur les numéros de messages (102 et 105).

```
Msg 102, Level 15, State 1, Line 3 Incorrect syntax near '123'.
Msg 105, Level 15, 1, Line 3 Unclosed quotation mark after the character String ''.
```

Listage 2.3 Exemple d'un message significatif d'erreur

2.3.2 Tautologie

L'objectif de ce type d'attaque est d'éviter le contrôle d'accès afin d'accéder à l'application Web comme un utilisateur légitime [HVO06, Nga09]. Pour ce faire, le pirate saisit des données dans les entrées d'utilisateur, de telle sorte que la condition de la requête SQL d'authentification soit toujours vérifiée (vrai) [HVO06, Nga09].

Par exemple : si le pirate saisit (Administrateur' OR 1=1- -) dans le champ d'usager et n'importe quelle valeur dans le champ de mot de passe, la requête SQL du listage 2.4 sera envoyée à la base de données.

```
Select * from TblUsagers Where Usager='Administrateur' OR 1=1 --' and MotPasse='123'
```

Listage 2.4 Exemple d'une tautologie

La condition (1=1) est toujours vérifiée et le reste de la commande est mis en commentaire grâce à la chaîne de caractères « - - ». Le pirate a donc réussi à s'authentifier en tant qu'administrateur et ensuite à accéder à toutes les données gérées par l'application, en lecture et en écriture.

2.3.3 Union de requêtes

L'objectif du pirate ici est de réunir sa propre requête avec celle qui est construite dans le code de l'application Web pour provoquer l'envoi des données confidentielles [HVO06].

Par exemple : si le pirate saisit (' UNION select * from TblUsagers - -) dans le champ d'usager et (123) dans le champ de mot de passe, la requête SQL du listage 2.5 sera envoyée à la base de données.

```
Select * from TblUsagers Where Usager=''
UNION
select * from TblUsagers – –' and MotPasse='123'
```

Listage 2.5 Exemple d'union de requêtes

Le pirate a pu extraire tous les usagers de l'application Web (TblUsagers), grâce à la deuxième requête, en injectant le mot clé SQL (UNION).

2.3.4 Commande SQL spécifique

La plupart des SGBDs offrent la possibilité d'appeler certaines commandes SQL ou système dans les instructions SQL. Un attaquant peut en profiter pour exécuter des commandes malveillantes sur le serveur de données [HVO06].

Par exemple : si le pirate saisit (' OR 1=1; Drop Table TblUsagers - -) dans le champ d'usager et (123) dans le champ de mot de passe, la requête SQL du listage 2.6 sera envoyée à la base de données.

```
Select * from TblUsagers Where Usager='' OR 1=1 ; Drop Table TblUsagers – –' and MotPasse='123''
```

Listage 2.6 Exemple d'une commande SQL spécifique

Si la gestion des privilèges n'est pas bonne, le pirate réussira à effacer la table (TblUsagers) grâce à la commande (Drop Table). Il peut aussi injecter

la commande (Shut Down) au lieu de (Drop Table) pour exécuter une attaque par déni de service contre le serveur de bases de données.

2.3.5 Injections SQL à l'aveuglette

L'attaquant utilise l'injection SQL à l'aveuglette (Blind SQL Injection) [OWA12] qui est un type avancé d'attaque lorsque l'application Web n'affiche pas des messages d'erreur significatifs. Pour ce faire, l'attaquant pose tout d'abord une série de questions de manière éclairée permettant de gagner du temps, mais sans modifier la structure des requêtes SQL finales [OWA12]. Ensuite, il observe les réponses (vrai ou faux) ou les temps de réponses à ses questions pour deviner certaines informations significatives [OWA12]. Ce type d'attaque est difficile à détecter, car la requête de l'attaquant respecte la structure attendue par le programmeur.

2.4 Les préjudices d'attaques par injection SQL

Les SQLIAs peuvent autoriser un attaquant en lui permettant de manipuler les données, d'accéder à l'application piratée, de voler les profils d'utilisateurs, d'obtenir des informations confidentielles, de réaliser un déni de service et d'exécuter des commandes malveillantes sur la base de données ainsi que sur le système d'exploitation [OWA10, SWLL06, HVO06].

Le tableau 2.1 résume comment chaque type de SQLIA viole un ou plusieurs objectifs de sécurité.

Objectifs de sécurité / Types d'attaque	Intégrité	Confidentialité	Disponibilité	Authentification
Requête syntaxiquement incorrecte	Non	Oui	Non	Non
Tautologie	Non	Oui	Non	Oui
Union de requêtes	Non	Oui	Non	Oui
Commande SQL spécifique	Oui	Non	Oui	Oui
Injections SQL à l'aveuglette	Non	Oui	Non	Non

Tableau 2.1 Violation des objectifs de sécurité par les différents types de SQLIA

Conclusion

Nous avons abordé dans ce chapitre les attaques par injection SQL. Nous avons montré à l'aide de plusieurs exemples pratiques que ces attaques sont relativement faciles à construire lorsqu'il existe des requêtes SQL dynamiques dans le code.

Pour détecter les différentes attaques illustrées dans ce chapitre, plusieurs chercheurs ont proposé plusieurs solutions. Dans le chapitre suivant, nous allons voir quelques techniques et contremesures proposées pour défendre contre les SQLIAs ainsi que leurs avantages et leurs inconvénients.

CHAPITRE III

LES CONTREMESURES DE L'ATTAQUE PAR INJECTION SQL

Les antivirus, les pare-feu et les IDS classiques ne bloquent pas les attaques par injection SQL, car ils opèrent sur les couches inférieures. Afin de protéger les pages Web écrites en ASP et ASP.NET contre ces attaques, Microsoft propose certaines méthodes de codage [Mic11a] sans offrir aucune solution automatisée de protection.

L'inefficacité relative de ces moyens justifie la nécessité de proposer des techniques et de développer de nouvelles contremesures ciblant particulièrement les applications Web afin de minimiser les risques d'attaques.

Ce chapitre vise à exposer certains travaux réalisés et publiés qui ont un rapport avec notre travail de recherche. Il est divisé en deux parties. Dans la première partie, nous discutons quelques bonnes pratiques de codage sécurisé tout en montrant leurs inconvénients. Dans la deuxième partie, nous discutons et évaluons certaines approches et contremesures avancées visant à se prémunir contre les SQLIAs.

3.1 Protection relative au code source

La validation insuffisante des entrées est la cause fondamentale des injections SQL [HVO06]. Une méthode intuitive de protection consiste donc à vérifier si ces entrées contiennent des caractères dangereux. Dans les sections suivantes, nous allons discuter de la validation des entrées, du filtrage des entrées, de l'utilisation des procédures stockées ainsi que de la limitation des privilèges proposés pour minimiser les risques d'attaques [HVO06, SWLL06, Nga09, Mic11b].

3.1.1 Validation des entrées

En général, chaque champ d'entrée correspond à une colonne dans une table de la base de données. Avant de construire des requêtes SQL dynamiques, une bonne pratique de codage consiste à valider les données de chacun des champs d'entrée en les comparant avec le type de la colonne correspondante.

Cependant, cette validation est valable pour les données numériques où le programmeur rejette les caractères non numériques. En revanche, il ne peut valider les entrées de type chaîne de caractères d'où proviennent la plupart des injections SQL.

3.1.2 Filtrage des entrées

Le filtrage des entrées consiste à rejeter les caractères dangereux comme « ' » ou « - » nécessaires pour exécuter une SQLIA, et à accepter seulement les lettres et les chiffres [HVO06, Nga09]. Les données filtrées ne vont jamais contenir des caractères dangereux, ce qui complique la tâche de l'attaquant.

Le taux de faux positifs de cette méthode est élevé, car elle est incapable de faire une analyse approfondie pour savoir si un caractère dangereux constitue

une attaque potentielle ou non. Par exemple, le mot (O'Brian) est considéré comme une tentative de SQLIA, ce qui n'est pas toujours le cas [Nga09]. L'autre désavantage est la difficulté de connaître à l'avance tous les caractères dangereux et les mots clés des divers SGBDs qui évoluent rapidement.

3.1.3 Procédure stockée

Une procédure stockée est un ensemble d'instructions SQL qui s'exécute sur le serveur de données à la suite d'appel d'une application [Wik11b]. Elle peut recevoir des paramètres et retourner des résultats [Wik11b]. Une technique simple de protection consiste à implémenter des procédures stockées avec les champs d'entrée comme paramètres.

D'une part, les procédures stockées n'existent pas dans tous les SGBDs; d'autre part, une procédure stockée peut être exploitable par une SQLIA si son code contient des requêtes SQL dynamiques. Comme le montre le listage 3.1, un pirate peut réaliser toutes sortes de SQLIA puisque le programmeur exécute une requête SQL dynamique en se servant de la commande (EXEC) sans valider ni filtrer les deux variables @usager et @MotPasse.

```
CREATE PROCEDURE dbo.GetUsagerCount
@usager as nvarchar(50),@MotPasse as nvarchar(50)
AS
EXEC ("SELECT Count(*) from TblUsagers
Where Usager='" + @usager + "' and MotPasse='" + @MotPasse + "' ")
GO
```

Listage 3.1 Exemple d'une procédure stockée vulnérable

3.1.4 Limitation des privilèges

Ce moyen de défense consiste à donner aux utilisateurs des privilèges limités nécessaires pour accéder à leurs fonctionnalités autorisées à condition de réviser périodiquement ces privilèges [SWLL06, Mic11b]. Ce moyen n'arrête pas les attaques internes réalisées par des utilisateurs possédant des comptes. En outre, un utilisateur, qui ne possède pas un compte, peut exécuter des tautologies par les champs de recherche des pages Web.

3.2 Les contremesures avancées

Toutes les techniques de codage précédentes sont inefficaces, car elles ne détectent pas automatiquement toutes les SQLIAs et engendrent des faux positifs ou des faux négatifs, sans compter que ces techniques exigent une modification majeure du code source.

Aussi plusieurs chercheurs ont-ils proposé un ensemble de contremesures avancées [HO06, GD04, BK04, BWS05, BBMm07, SW, SS] afin d'automatiser le processus de protection contre les SQLIAs. Il est important de noter que la méthode HTTP utilisée (GET ou POST) n'a aucun impact sur le fonctionnement de ces contremesures. Dans cette section, nous analysons et évaluons sept contremesures, classifiées en trois approches différentes :

- Security Gateway [SS] qui se base sur le filtrage des paramètres d'entrée ;
- JDBC-checker [GD04] qui se base sur l'analyse du code source ;
- SQLRand [BK04], SQLCheck [SW], SQLGuard [BWS05], CANDID [BBMm07] et AMNESIA [HO06] qui reposent sur l'analyse du code et l'analyse des requêtes dynamiques au moment de l'exécution [Nga09].

3.2.1 Serveur mandataire de filtrage

Cette méthode consiste à valider les paramètres d'entrée selon certaines règles ou signatures d'attaques.

Security Gateway

Security Gateway [SS] est une passerelle de filtrage dédiée à une application Web où les développeurs doivent spécifier, avec un langage de description de politique de sécurité (SPDL), certaines règles de validation à partir desquelles un code de vérification est généré automatiquement. Durant l'exécution, une passerelle de sécurité est placée sur le serveur Web pour rejeter (filtrer) les demandes HTTP qui ne respectent pas ces règles.

Cette passerelle n'exige ni l'analyse ni la modification du code source et détecte non seulement les SQLIAs, mais aussi d'autres attaques. En revanche, le taux de faux positifs et de faux négatifs est très élevé si les règles de validation sont incorrectes ou inappropriées. De surcroît, la modification de l'application Web nécessite systématiquement une modification des règles de validation.

3.2.2 Analyse du code source

Cette méthode est fondée sur l'analyse du code source de l'application Web cible.

JDBC-checker

JDBC-checker [GD04, Nga09] est un analyseur de classes Java qui vérifie si les entrées d'utilisateur pourraient causer des requêtes syntaxiquement incorrectes au moment de l'exécution. Pour ce faire, il génère tout d'abord un automate fini qui représente potentiellement chaque requête SQL dynamique. Ensuite, il vérifie

si l'automate peut produire des requêtes syntaxiquement incorrectes.

Cet analyseur est efficace puisqu'il couvre toutes les requêtes (Select) possibles durant l'exécution. Par contre, il s'applique uniquement aux requêtes de type Select et aux applications Java/JDBC. Il ne détecte que les attaques de type requête syntaxiquement incorrecte et le programmeur devra modifier le code source pour pouvoir prévenir ces attaques. Ceci est problématique puisque les attaques sont souvent des requêtes SQL syntaxiquement correctes.

3.2.3 Analyse du code et analyse dynamique

Cette approche est fondée sur l'analyse du code et l'analyse des requêtes SQL dynamiques au moment de l'exécution [Nga09].

SQLRand

Dans cette approche [BK04, Nga09], les développeurs des pages Web doivent appliquer un algorithme de randomisation, en utilisant une clé secrète, afin de remplacer les mots-clés SQL (Select, Insert, etc.) par des mots-clés aléatoires (Select 123, Insert456, etc.). Au moment de l'exécution, un filtre mandataire est installé entre l'application web et le SGBD pour convertir, avec la même clé de chiffrage, les requêtes aléatoires en des requêtes SQL appropriées pour que le SGBD puisse les interpréter correctement. Un attaquant, qui ne possède pas la clé secrète, injecte des mots clés SQL classiques. Dans ce cas, le SGBD déclenche un message d'erreur de syntaxe.

Cette approche est efficace si l'attaquant ne connaît pas la clé secrète, mais elle exige la modification du code source pour pouvoir appliquer l'algorithme de randomisation. L'autre inconvénient est que les messages d'erreur retournés par le SGBD peuvent révéler certaines informations sensibles à l'attaquant.

SQLCheck

Cette approche [SW, Nga09] offre aux développeurs une bibliothèque pour pouvoir marquer le début et la fin des champs d'entrée respectivement, par les caractères «(—» et «—)». Durant l'exécution et pour une requête SQL interceptée, une analyse grammaticale est faite pour vérifier si les données saisies par l'utilisateur ont changé la syntaxe de cette requête. Dans ce cas, elle sera bloquée. Sinon, elle sera envoyée sans les caractères «(—» et «—)» à la base de données.

SQLCheck s'applique uniquement aux applications Web écrites en PHP (ou JSP) et exige l'utilisation d'une bibliothèque intermédiaire et d'un analyseur lexical ainsi que syntaxique pour interpréter grammaticalement les requêtes SQL.

SQLGuard

L'approche SQLGuard [BWS05] est très semblable à celle de SQLCheck [SW] dans le sens où les développeurs doivent aussi marquer manuellement le début et la fin des champs d'entrée employés dans les requêtes SQL dynamiques. La seule différence est que durant l'exécution, SQLGuard compare les arbres syntaxiques avec et sans les entrées d'utilisateur au lieu de faire une analyse grammaticale. Si les deux arbres ne sont pas identiques, la requête SQL avec les entrées est considérée comme une attaque.

Le désavantage de SQLGuard est qu'elle exige une modification manuelle du code source et qu'elle est valable uniquement pour les applications Web développées en Java. De plus, la comparaison entre les arbres syntaxiques peut être très coûteuse en termes de performance.

CANDID

CANDID [BBMm07] analyse le code source pour construire l'ensemble des requêtes SQL bénignes en affectant aux variables de type chaîne de caractères des valeurs sans aucun caractère dangereux. La structure de chaque requête est donc celle qui est prévue par le programmeur. Au moment de l'exécution, l'arbre syntaxique de la requête (candidate) avec les entrées d'utilisateur sera comparé avec celui de la requête bénigne. Si les deux arbres ne sont pas identiques, la requête candidate est considérée comme une SQLIA.

Le désavantage de cette approche est qu'elle s'applique uniquement aux applications Web développées en Java et qu'elle exige l'analyse du code source et la reconstruction des requêtes bénignes à la suite de toute modification sur l'application Web en question. Comme SQLGuard, la comparaison entre les arbres syntaxiques peut être très coûteuse en termes de performance.

AMNESIA

AMNESIA [HO06, Nga09] balaye tout d'abord le code source pour identifier toutes les parties (hotspots) qui accèdent à la base de données en utilisant des requêtes SQL dynamiques. Ensuite, un automate fini qui représente toutes les requêtes SQL possibles est associé à chacun de ces hotspots. Au moment de l'exécution, AMNESIA intercepte les requêtes SQL produites dans chaque hotspot et les valide ensuite en regard des automates déjà générés. Si une requête SQL ne respecte pas l'automate associé, elle sera rejetée et considérée comme une attaque.

Cette approche efficace et automatique n'impose pas la récriture du code source. En revanche, elle peut produire des faux positifs lorsque l'analyse du code et la modélisation des requêtes dynamiques ne sont pas assez précises,

et des faux négatifs si l'attaquant génère des requêtes qui correspondent à des automates erronés. Actuellement, AMNESIA protège uniquement les applications Web écrites en PHP et MySQL. Il serait difficile de l'adapter aux autres environnements puisqu'elle analyse le code, qui est certainement spécifique pour chaque langage de programmation et pour chaque SGBD.

3.2.4 Évaluation qualitative

Nous avons évalué les approches mentionnées, comme les auteurs de [HVO06, Nga09], mais cette fois selon l'efficacité et la portabilité.

Efficacité

Nous avons évalué l'efficacité des contremesures précédentes, en fonction des critères suivants :

- Les types d'attaques détectées (tautologie, union de requêtes, requête syntaxiquement incorrecte et commande SQL spécifique), la valeur "Tous" signifie que la contremesure détecte tous les types de SQLIA sauf l'injection à l'aveuglette puisqu'elle est non détectable quelle que soit la contremesure.
- Si la contremesure engendre des faux positifs ou négatifs.
- Si la contremesure résiste face aux techniques d'évasion.

Le tableau 3.1 résume l'évaluation qualitative de l'efficacité des contremesures étudiées.

Contremesures	Type d'attaques	Faux positifs ou négatifs	Technique d'évasion
AMNESIA	Tous	Oui	Oui
CANDID	Tous	Oui	Non
JDBC-Checker	Requête syntaxiquement incorrecte	Oui	Oui
Security Gateway	Tous	Oui	Non
SQLCheck	Tous	Oui	Oui
SQLGuard	Tous	Oui	Oui
SQLRand	Tautologie, union de requêtes, commande SQL spécifique	Oui	Oui

Tableau 3.1 Évaluation de l'efficacité des contremesures avancées

Portabilité

Nous avons évalué la portabilité des contremesures précédentes, qui nous semble très importante, en fonction des critères suivants :

- Si la contremesure dépend de l'environnement de développement (.NET, J2EE, etc.).
- Si la contremesure dépend du SGBD (MS SQL, MySQL, etc.).
- Si la contremesure dépend du serveur Web (Apache, IIS, etc.).

Le tableau 3.2 résume l'évaluation qualitative de la portabilité des contremesures étudiées.

Contremesures	Dépend de l'environnement	Dépend du SGBD	Dépend du serveur Web
AMNESIA	Oui	Oui	Non
CANDID	Oui	Non	Non
JDBC-Checker	Oui	Non	Non
Security Gateway	Oui	Non	Non
SQLCheck	Oui	Non	Non
SQLGuard	Oui	Non	Non
SQLRand	Oui	Oui	Non

Tableau 3.2 Évaluation de la portabilité des contremesures avancées

Conclusion

Certaines contremesures avancées, comme SQLRand [BK04] et AMNESIA [HO06], peuvent être efficaces même si elles engendrent des faux positifs ou négatifs. Toutes les contremesures étudiées dépendent de l'environnement de développement et exigent une analyse du code ou une analyse dynamique lors de l'exécution. La vaste majorité de ces contremesures nécessitent donc des modifications, ou dans le meilleur des cas, une analyse du code source. Cette faiblesse est problématique si le code source n'existe pas ou les développeurs ne réalisent pas correctement les modifications nécessaires. C'est pourquoi, nous allons tenter de relever ces défis dans les trois chapitres suivants.

CHAPITRE IV

PROPOSITION : UNE APPROCHE DE DÉTECTION D'ANOMALIES CONTRE L'ATTAQUE PAR INJECTION SQL

L'autonomie du processus de détection nous semble cruciale pour la mise en place d'un système de détection d'anomalies facile à déployer et à utiliser. Le système doit être le plus autonome possible pour proposer des capacités de détection ne nécessitant pas d'intervention humaine et pour s'adapter automatiquement à l'évolution de toutes les composantes du trafic, normales et anormales. Nous proposons pour cela de développer une méthode d'apprentissage automatique ne requérant aucune intervention humaine.

L'objectif de ce chapitre est d'exposer notre approche face aux attaques par injection SQL. Nous présentons dans la section 4.1 le contexte de notre approche. La section 4.2 est consacrée à l'explication de l'approche proposée et ses deux phases. Dans la section 4.3, nous suggérons une méthode de corrélation entre le trafic HTTP et le trafic SQL. Dans la section 4.4, nous détaillons la modélisation des comportements normaux. La section 4.5 présente l'identification et la détection des attaques.

4.1 Contexte de l'approche

Dans cette section, nous citons tout d'abord les exigences fonctionnelles et non fonctionnelles de notre approche. Ensuite, nous évaluons la pertinence des approches existantes de détection par rapport aux attaques par injection SQL. Enfin, nous formalisons notre approche pour une meilleure compréhension de la suite de rapport.

4.1.1 Exigences de l'approche

En termes d'exigences fonctionnelles, l'approche proposée devra :

- protéger une application Web donnée sans accéder à son code source ni l'analyser ni le modifier ;
- détecter les attaques par injection SQL en temps réel ;
- fonctionner avec un taux réduit de faux positifs et de faux négatifs ;
- permettre à l'administrateur de générer des rapports de sécurité ;
- assurer une protection automatique en réduisant l'intervention d'opérateurs humains ;
- fonctionner correctement sous les différents protocoles Web (HTTP ou HTTPS).

En termes d'exigences non fonctionnelles, elle devra :

- être portable, en protégeant une application Web donnée indépendamment du code source, de la structure de données, du fournisseur de bases de données (MS SQL, MySQL, etc.) et du type de serveur Web (IIS, Apache, etc.) ;
- être simple à déployer et à configurer ;
- n'exiger aucune formation pour les développeurs des pages Web ;
- avoir une influence minimale sur la performance des serveurs Web ;
- avoir une augmentation acceptable du temps de réponse aux demandes HTTP.

4.1.2 Pourquoi la détection d'anomalies ?

Comme nous avons déjà expliqué dans le premier chapitre, il existe deux approches principales dans le domaine de la détection d'intrusions : la détection par signature et la détection d'anomalies.

La détection par signature ne consomme pas trop de ressources informatiques et son implémentation est généralement simple. Par contre, elle ne permet pas de trouver les nouvelles attaques ou les attaques pour lesquelles les signatures n'existent pas. Elle comporte aussi plusieurs problèmes majeurs à l'égard des attaques par injection SQL :

- Les signatures diffèrent d'un SGBD à un autre, car les caractères dangereux et les mots clés ne sont pas les mêmes dans tous les SGBDs.
- Les signatures ne résistent pas aux techniques d'évasion appliquées pour contourner les signatures d'attaques prédéfinies. Cette faiblesse augmente grandement le taux de faux négatifs.
- Le taux de faux positifs est très élevé dû au fait que le système déclenche des alertes dès qu'il trouve un caractère dangereux (ou un mot clé) sans être capable de distinguer entre l'utilisation normale et l'utilisation illégale de ce caractère. Par exemple, la chaîne de caractères (O'BRAIN) est considérée toujours comme une tentative d'attaque car cette chaîne contient le caractère (').

Nous avons adopté la détection d'anomalies pour contrer les attaques par injection SQL en tenant compte de certains avantages majeurs par rapport à l'approche par signature :

- elle est plus efficace et permet de détecter les nouvelles attaques sans mettre à jour régulièrement l'IDS ;
- elle convient mieux à nos exigences, notamment le défi de ne pas accéder au

code source ;

- elle permet de développer une solution automatisée puisque nous n'aurons pas besoin de maintenir périodiquement une base de signatures d'attaques ;
- elle permet de développer un système autonome et indépendant du SGBD et du langage de programmation.

Malheureusement, la détection d'anomalies peut être coûteuse en termes de performance et peut entraîner des faux positifs. Conscients de ces problèmes, nous allons chercher à améliorer la performance et réduire les faux positifs afin de concevoir un système efficace de détection d'anomalies.

4.1.3 Formalisation de l'approche

Pour simplifier les explications et l'expérimentation de notre proposition, nous formalisons les définitions et les notions de la manière suivante :

Définition 1 : littéral. Un littéral est tout élément dans une instruction SQL qui n'est ni un identificateur (mot clé, nom d'une table, nom d'un champ, etc.) ni un opérateur dans le langage SQL. Les littéraux sont extraits par un analyseur lexical SQL.

Définition 2 : caractère dangereux. Un caractère est dangereux s'il n'est ni une lettre ni un chiffre.

Définition 3 : Template. Un template est un modèle SQL obtenu en remplaçant les littéraux d'une instruction SQL par le caractère (?). Le listage 4.1 illustre un exemple de template.

```
select * from TblUsagers Where Usager='?' and MotPasse='?'
```

Listage 4.1 Exemple d'un modèle SQL (Template)

Définition 4 : lstTemplate. Soit lstTemplate=($Template_1$,$Template_2$,...,$Template_m$) une liste ordonnée des modèles SQL (templates).

Définition 5 : SetLstTemplate. Soit SetLstTemplate={$lstTemplate_1$,...,$lstTemplate_l$} un ensemble de listes des templates.

Définition 6 : URL. Soit URL une chaîne de caractères représentant le chemin vers une page web.

Définition 7 : DB. Soit DB la base de connaissances qui contient les modèles SQL pour chaque URL de l'application Web à protéger. Chaque entrée de cette base est définie par le tuple suivant : u =< URL ;SetLstTemplate > où :

- URL représente une adresse URL (voir définition 6).
- SetLstTemplate (voir définition 5).

Définition 8 : AttacksDB. Soit AttacksDB la base d'attaques détectées où chaque entrée correspond à une requête SQL qui a été considérée comme une SQLIA. Chaque entrée de cette base est définie par le tuple suivant :
a =< IDAttack ;URL ;SQLStatement ;IPSource ;AttackDate > où :

- IDAttack identifie de manière unique chaque entrée.
- URL est l'adresse de la page Web à partir de laquelle la requête SQL a été produite.
- SQLStatement est le code de la requête SQL.
- IPSource est l'adresse IP source du paquet qui contient la requête SQL.
- AttackDate est la date et le temps d'occurrence de l'attaque.

Définition 9 : LITERALS. Soit LITERALS ={$literal_1$,$literal_2$,...,$literal_n$} l'ensemble de littéraux d'une requête SQL.

Définition 10 : ReqSql. Soit ReqSql une requête SQL qui peut être définie par

le tuple suivant :

ReqSql =< Requete ;Template ;Ordre ;TimeStamp ;LITERALS > où :

- Requete est le code de la requête SQL.
- Template est le modèle SQL généré à partir de la requête SQL.
- Ordre est l'ordre d'exécution de la requête SQL dans la liste des requêtes SQL produites par la même demande HTTP.
- TimeStamp est la date et le temps d'occurrence de la requête SQL.
- LITERALS est l'ensemble de littéraux de la requête SQL (voir définition 9).

Définition 11 : lstReqSql. Soit lstReqSql=(ReqSql_1,ReqSql_2,...,$ReqSql_k$) une liste des requêtes SQL, ordonnées selon le TimeStamp (la date et le temps d'occurrence).

Définition 12 : PARAMS. Soit PARAMS={Param_1,Param_2,...,Param_m} l'ensemble de valeurs des paramètres d'entrée (champs d'entrée, cookies, etc.) d'une demande HTTP. Param_i est la valeur de $i^{ème}$ paramètre.

Définition 13 : DemHttp. Soit DemHttp une demande HTTP qui peut être définie par le tuple suivant :

DemHttp =< URL ;TimeStamp ;Seuil ;PARAMS ;Type ;lstReqSql > où :

- URL est l'adresse de la page Web à partir de laquelle la demande HTTP a été exécutée.
- TimeStamp est la date et le temps d'occurrence de la demande HTTP.
- Seuil est le seuil pour accepter qu'une requête SQL soit corrélée à la demande HTTP (détails dans la troisième section).
- PARAMS est l'ensemble de valeurs des paramètres d'entrée de la demande HTTP (voir définition 12).
- Type est le type de la demande HTTP : contaminée ou normale. Si la valeur d'un ou plusieurs paramètres d'entrée contient au moins un caractère

dangereux, la demande HTTP est considérée comme contaminée. Sinon, elle est considérée comme normale.

- lstReqSql est la liste des requêtes SQL produites par la demande HTTP (voir définition 11).

Définition 14 : lstDemHttp. Soit lstDemHttp=($DemHttp_1$,$DemHttp_2$,...,$DemHttp_n$) une liste des demandes HTTP (DemHttp), ordonnées selon le TimeStamp.

Définition 15 : Session. Une session Web d'un client est définie par le tuple suivant :

Session =< SessionID ;IPSource ;lstDemHttp > où :

- SessionID est l'identificateur (ID) unique de la session. Cet ID existe souvent dans l'un des cookies d'une demande HTTP (voir listage 1.1).
- IPSource est l'adresse IP de l'ordinateur du client qui a démarré la session.
- lstDemHttp est la liste des demandes HTTP lancées durant la session, ordonnées selon le TimeStamp (voir définition 14).

4.2 Principe de l'approche

Notre proposition est une solution automatisée dont l'approche principale est la détection d'anomalies. Les comportements normaux sont les modèles SQL produits par des demandes HTTP normales (les valeurs de tous les paramètres d'entrée ne contiennent pas des caractères dangereux) et l'anomalie est une requête SQL malveillante tentant de s'exécuter sur la base de données de l'application Web cible. Les attaques sont identifiées en comparant les requêtes SQL interceptées avec les modèles SQL qui ont été modélisés dans la base de connaissances (DB). Ces requêtes détectées (anomalies) sont considérées comme des attaques par injection SQL. Le nombre de fausses alertes et le nombre d'attaques non détectées dépendent respectivement de la précision et

de l'exhaustivité des comportements normaux modélisés au préalable.

L'approche proposée comporte deux phases : l'une d'apprentissage, qui sert à modéliser les comportements normaux de l'application Web ; l'autre de protection, qui consiste à identifier les comportements anormaux parmi les nouvelles activités. Le processus de modélisation prend fin lorsqu'aucun ensemble de modèles SQL n'est ajouté à la base de connaissances pendant une longue période d'apprentissage. Dans ce cas-ci, l'administrateur de l'application Web confirme le passage de la phase d'apprentissage à la phase de protection.

4.2.1 Phase d'apprentissage

La phase d'apprentissage consiste à modéliser les requêtes SQL normales (comportements normaux) de l'application Web dans la base de connaissances (DB). La figure 4.1 montre le déroulement de cette phase qui se fait comme suit :

1. Le capteur HTTP intercepte les demandes HTTP en laissant passer celles qui sont normales. Il extrait les paramètres, le TimeStamp, l'URL et la session pour chacune de ces demandes HTTP.
2. Le capteur SQL intercepte les requêtes SQL et remplace les littéraux de celles-ci par un caractère spécial (par exemple : ?) afin de construire les modèles SQL (Templates) de ces requêtes. Ce capteur met aussi en relation les requêtes SQL et les demandes HTTP selon une méthode de corrélation qui sera détaillée dans la section 4.3.
3. Le générateur de profils normaux (modèles de requêtes SQL) s'exécute après l'expiration d'une session client. Il modélise les requêtes SQL pour chaque demande HTTP de la session expirée dans la base de connaissances.

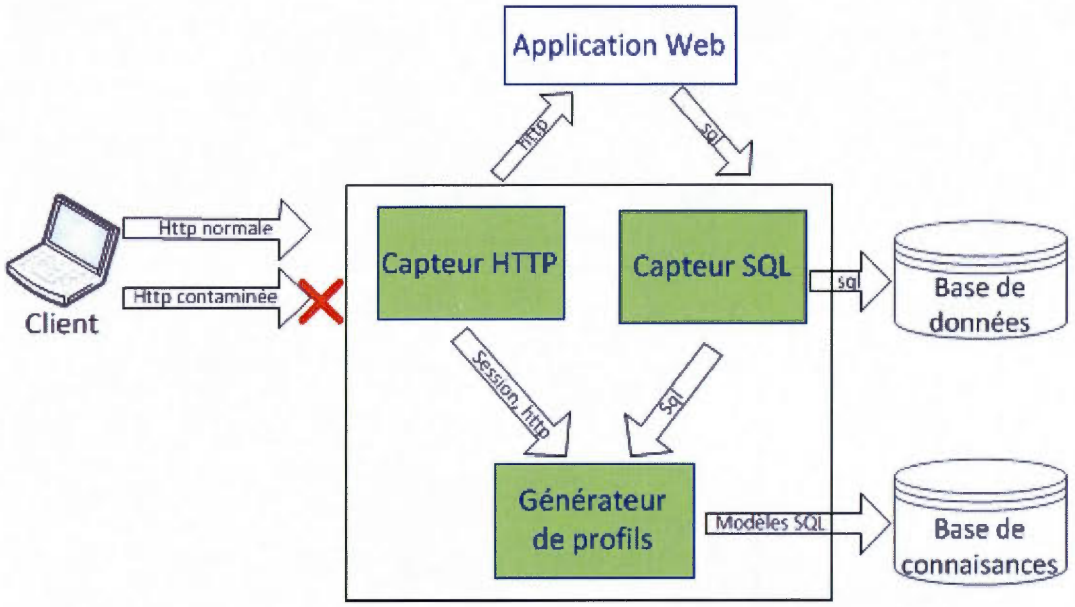

Figure 4.1 Phase d'apprentissage de l'approche proposée

4.2.2 Phase de protection

La phase de protection consiste à détecter les attaques par injection SQL (les intrusions). Il est donc nécessaire de classifier les requêtes SQL en deux catégories : celles qui sont normales; celles qui sont malveillantes. Certaines informations liées à la détection des requêtes malveillantes seront ensuite enregistrées dans la base d'attaques (AttacksDB) afin d'offrir à l'administrateur la capacité de générer des rapports de sécurité. La figure 4.2 montre la détection des requêtes SQL malveillantes qui se déroule comme suit :

1. Le capteur HTTP intercepte et classifie les demandes HTTP (normales ou contaminées). Il extrait les paramètres, le TimeStamp, l'URL et la session pour chacune de ces demandes.
2. Le capteur SQL intercepte et traite les requêtes SQL de la même manière que la phase d'apprentissage.

3. Le détecteur d'attaques reçoit une requête SQL de la part du capteur SQL et décide ensuite si elle est malveillante. En ce cas, il enregistre l'IPSource, l'URL, le code de la requête et l'instant d'occurrence dans la base d'attaques (AttacksDB).

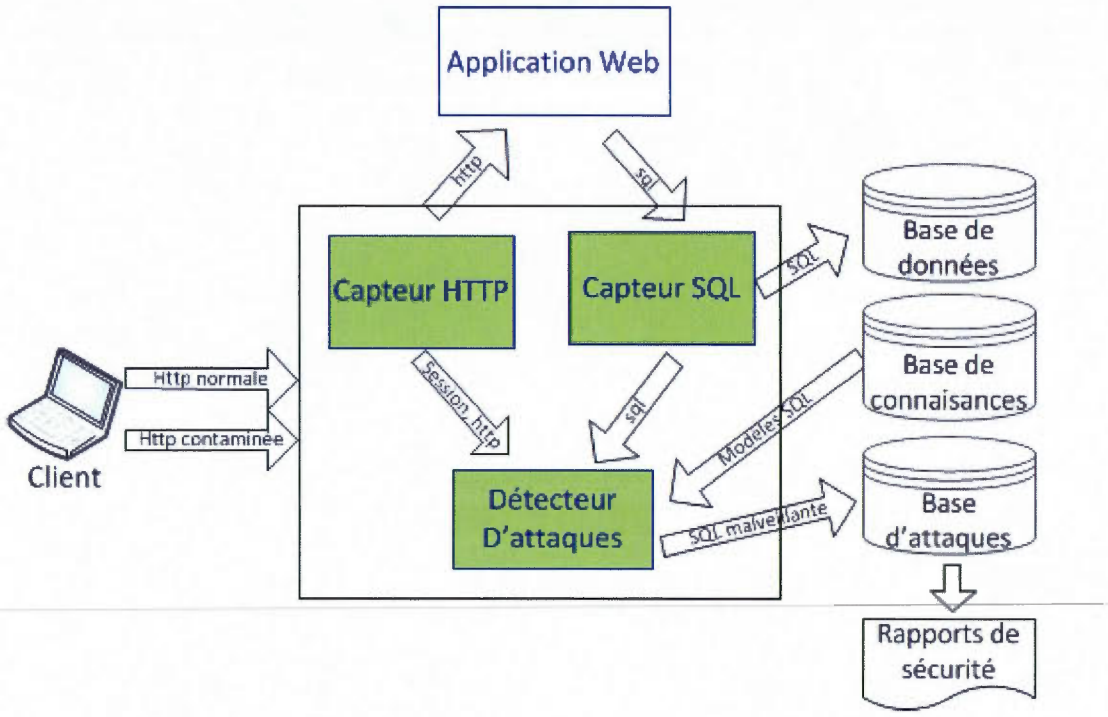

Figure 4.2 Phase de protection de l'approche proposée

4.3 Corrélation entre les demandes HTTP et les requêtes SQL

Dans une application Web, une demande HTTP se produit lorsqu'un utilisateur saisit des données et exécute une action à laquelle un code est associé. Un ensemble de requêtes SQL est généré pour répondre à la demande HTTP lancée durant une session Web. L'identification de l'utilisateur est souvent une variable de session Web, d'où la possibilité de déterminer l'utilisateur qui a exécuté

une demande HTTP comme dans le commerce électronique. Par contre, il est difficile - sinon impossible - d'identifier l'utilisateur final (ou la session) qui cause une requête SQL en tenant compte uniquement de la base de données. Techniquement, il n'y a pas de manière précise qui permet de connaître, pour une demande HTTP, les requêtes SQL générées par une application Web.

Pouvoir établir une corrélation entre les demandes HTTP et les requêtes SQL est nécessaire dans notre approche pour modéliser les comportements normaux de l'application Web et pour identifier l'utilisateur (ou la session) qui tente d'exécuter des requêtes SQL malveillantes sur la base de données.

Selon nos connaissances, il n'existe pas dans la littérature de méthode de corrélation entre les demandes HTTP et les requêtes SQL. Pourtant, il existe plusieurs algorithmes pour mesurer la ressemblance entre les chaînes de caractères. Nous allons nous servir de l'un de ces algorithmes pour construire notre méthode de corrélation en traitant les demandes HTTP et les requêtes SQL comme des ensembles de chaînes de caractères.

La corrélation est intéressante à plusieurs points de vue :

- L'identification de la session et de l'URL à partir desquelles une requête SQL a été exécutée dans les deux phases : phase d'apprentissage et phase de protection.
- La possibilité de bloquer complètement la session d'où provient une attaque détectée.
- Une amélioration de la performance puisqu'il est possible de s'intéresser uniquement aux requêtes SQL produites par des demandes HTTP contaminées durant la phase de protection.

Par la suite, nous allons dans un premier temps expliquer quelques algorithmes pour mesurer la ressemblance entre les séquences, puis nous allons présenter notre

méthode pour réaliser la corrélation HTTP et SQL et allons détailler un scénario de corrélation.

4.3.1 Ressemblance entre les séquences

Plusieurs algorithmes [LD12, Wik12c, Ham12] ont été proposés pour mesurer la distance ou la ressemblance entre deux séquences (chaînes de caractères).

Distance d'édition [LD12]

L'algorithme de la distance d'édition (ou distance de Levenshtein) prend en paramètre deux chaînes de caractères et retourne le nombre minimal d'opérations (insertion, suppression, substitution ou transposition) qui doivent être appliquer sur la première chaîne pour obtenir la deuxième chaîne. La version originale de cet algorithme manipule une matrice de dimension $(m+1) \times (n+1)$ où n et m sont les dimensions des deux chaînes de caractères. Une amélioration a été apportée à cet algorithme et qui consiste à stocker seulement la ligne précédente en mémoire. La complexité est devenue donc $O(m)$ où m est la longueur de la chaîne de caractères la plus longue.
Exemple : soit les deux séquences $S_1 = \{A, D, M, I, N\}$ et $S_2 = \{A, D, I, O, R\}$.
L'ensemble d'opérations pour passer de S_1 à S_2 est $\{Supp(S_1[2]), Subb(S_1[4]), Add(S_1[5])\}$, la distance d'édition entre S_1 et S_2 est donc égale à 3.

Longest Common Subsequence [Wik12c]

La sous-séquence commune entre deux séquences est celle dans laquelle ses éléments apparaissent, dans le même ordre, mais pas nécessairement consécutifs dans ces deux séquences. La LCS donne la sous-séquence commune de longueur maximale entre deux séquences. Évaluer la LCS est un problème NP-complet

difficile.

Exemple : soit les deux séquences $S_1 = \{A, D, M, I, N\}$ et $S_2 = \{A, M, N, O, R\}$. LCS(S_1,S_2) = $\{A, M\}$.

Distance de Hamming [Ham12]

La distance de Hamming entre deux séquences de même longueur est le nombre de positions où ces deux séquences diffèrent.

Exemple : soit les deux séquences $S_1 = \{A, D, M, I, N\}$ et $S_2 = \{A, D, I, O, R\}$. La distance de Hamming entre S_1 et S_2 est égale à 3 car les deux symboles A et D sont communs et les trois autres diffèrent.

4.3.2 Méthode de corrélation HTTP et SQL

Notre objectif est de déterminer, pour une requête SQL, la demande HTTP la plus similaire qui sera considérée comme celle qui a produit la requête SQL en question. Nous allons y parvenir en introduisant une méthode de corrélation qui repose sur une mesure de similarité entre les requêtes SQL et les demandes HTTP qui s'exécutent simultanément.

Calcul des scores de mappage

Afin de quantifier le degré de similarité entre une demande HTTP (DemHttp) et une requête SQL (ReqSql), nous allons calculer ce que nous appelons le score de mappage entre elles dans le but de mesurer le degré d'indépendance entre les paramètres HTTP et les littéraux SQL.

Considérons une demande DemHttp=$\{Param_1, Param_2, ..., Param_n\}$ et une requête

ReqSql=$\{Literal_1, Literal_2, ..., Literal_m\}$ pour lesquelles nous souhaitons calcu-

ler le score de mappage.

Souvent, le programmeur utilise un seul paramètre HTTP dans chaque littéral SQL pour construire une requête SQL dynamique. Pour cette raison, nous cherchons tout d'abord, pour chaque paramètre $Param_i$, le littéral $Literal_j$ le plus similaire en déterminant la distance minimale entre $Param_i$ et tous les littéraux de la requête SQL. Ensuite, nous calculons la moyenne de similarité en divisant la somme des distances minimales par le nombre de paramètres (n).

La démarche détaillée pour calculer le score de mappage entre la demande HTTP (DemHttp) et la requête SQL (ReqSql) peut être formalisée par la formule suivante (voir algorithme 4.1) :

$$Score(DemHttp, ReqSql) = \frac{\sum_{i=1}^{n} \min_{j=1}^{m} Dist(Param_i, Litteral_j)}{n} \qquad (4.1)$$

- n est le nombre de paramètres de la demande HTTP (DemHttp).
- m est le nombre de littéraux de la requête SQL (ReqSql).
- $Param_i$ est le $i^{\text{ème}}$ paramètre de la demande HTTP (DemHttp).
- $Litteral_j$ est le $j^{\text{ème}}$ littéral de la requête SQL (ReqSql).
- La fonction $Dist(Param_i, Litteral_j)$ retourne la distance entre les deux chaînes de caractères $Param_i$ et $Litteral_j$ en appliquant la distance d'édition (voir listage B.1 de l'appendice B). Cette fonction permet de quantifier le nombre d'opérations réalisées par le programmeur sur le paramètre ($Param_i$) de la demande HTTP (DemHttp) avant de l'employer dans le littéral ($Litteral_j$) de la requête SQL dynamique (ReqSql).

Nous avons choisi la distance d'édition pour mesurer la ressemblance entre un paramètre HTTP et un littéral SQL puisque cette distance couvre toutes les opérations sur les chaînes de caractères et mesure la distance entre les chaînes de longueurs variables. Néanmoins, le programmeur peut faire plusieurs types

d'opérations sur les paramètres d'entrée avant de les introduire dans une requête SQL, comme concaténer des chaînes, extraire une sous-chaîne, etc. De plus, les paramètres et les littéraux ne sont pas toujours de même taille.

Algorithme 4.1: Calcul du score de mappage entre une demande HTTP et une requête SQL

Input : $DemHttp = \{param_1, param_2, \ldots, param_n\}$.
Input : $ReqSql = \{literal_1, literal_2, \ldots, literal_m\}$.
Output : le score de mappage entre DemHttp et ReqSql.

```
1  score ← 0;
2  for i ← 1 to n do
3  |   min ← MAX_FLOAT;
4  |   for j ← 1 to m do
5  |   |   Dist ← EditDistance(param_i, literal_j);
6  |   |   if Dist < min then
7  |   |   |   min ← Dist ;
8  |   |   end
9  |   end
10 |   score ← score + min;
11 end
12 score ← score/n;
```

Afin d'améliorer la performance de la corrélation, nous allons calculer les scores de mappage uniquement entre les requêtes SQL et les demandes HTTP qui s'exécutent simultanément. Cela permet également d'optimiser le calcul en évitant des mesures qui n'ont pas de sens. Nous calculons donc les scores entre les couples (DemHttp,ReqSql) respectant la condition suivante :

- $\text{TimeStamp}_{ReqSql} > \text{TimeStamp}_{DemHttp}$ et $\text{TimeStamp}_{ReqSql} < \text{TimeStamp}_{DemHttp} + t$

où t est une fenêtre de temps en millisecondes, c'est le temps nécessaire (maximal) pour que l'application Web puisse générer les requêtes SQL d'une demande HTTP.

Nous visualisons les scores calculés dans une matrice virtuelle, qu'on l'appellera matrice de similarité HTTP et SQL. Chaque ligne de la matrice contient les scores de mappage d'une demande HTTP et chaque colonne contient les scores de mappage d'une requête SQL. En réalité, cette matrice n'existe pas en mémoire et les informations nécessaires, pour réaliser les calculs, sont stockées dans des objets.

Sélection de la demande HTTP la plus similaire

L'étape suivant le calcul des scores de mappage vise à identifier la demande HTTP la plus similaire pour chaque requête SQL. Pour ce faire, nous allons sélectionner la demande HTTP qui correspond au score minimal dû au fait que le score de mappage mesure le degré d'indépendance entre les paramètres d'entrée d'une demande HTTP et les littéraux d'une requête SQL. Pour accepter ce score, il doit être plus petit qu'un seuil d'acceptation qui a été défini comme suit :

$$Seuil(DemHttp) = 0,6 \times \frac{\sum_{i=1}^{n}(Param_i.length)}{n} \tag{4.2}$$

Il s'agit de la taille moyenne des paramètres de la demande HTTP (DemHttp) multipliée par 60%. Ce seuil mesure le degré d'indépendance acceptable pour qu'une requête SQL (ReqSql) soit corrélée à la demande HTTP (DemHttp). Si le score de mappage entre (DemHttp) et (ReqSQL) est plus grand que $Seuil(DemHttp)$, la requête SQL (ReqSQL) est considérée comme indépendante de (DemHttp).

Le choix du pourcentage (60%) n'est pas évident et a un impact sur les résultats de corrélation. Pour choisir un bon pourcentage sans perdre des corrélations, nous avons effectué des expérimentations en modifiant le pourcentage (50%, 55%, 60%, 65%, etc.) dans la formule 4.2. Nous avons obtenu les meilleurs résultats avec un pourcentage de 60%.

Si pour une requête SQL (ReqSql), il n'existe aucune demande HTTP (DemHttp) telle que : Score(DemHttp,ReqSql) est plus petit que Seuil(DemHttp), nous concluons que (ReqSql) est indépendante de toutes les demandes HTTP. En ce cas, la requête SQL (ReqSql) sera ignorée puisqu'elle ne dépend pas des paramètres d'entrée et ne constitue pas vraisemblablement un risque d'attaque.

4.3.3 Scénario de corrélation HTTP et SQL

Pour mettre en évidence la méthode de corrélation précédente, nous avons considéré trois pages Web qui s'exécutent simultanément.

1. **Page1 :** permet la recherche des clients grâce à deux champs d'entrée txtprenom et txtnom. Supposons que le programmeur a choisi d'agrandir la plage de résultats en filtrant selon une partie de nom et de prénom, comme le montre le listage 4.2.

```
strsql1 = "Select * from TblClients Where Prenom like '%" + txtprenom.Text.Substring(1, 4) + "%'
and Nom like'%" + txtnom.Text.Substring(1, 4) + "%' ";
```

Listage 4.2 Requête SQL dynamique pour la recherche des clients

2. **Page2 :** permet à un utilisateur de s'authentifier en saisissant le nom de l'usager et le mot de passe grâce aux champs d'entrée (txtusager et txtMotPasse). Supposons que le programmeur a décidé de retourner le nombre d'usagers (strsql2) avant de réaliser l'authentification (strsql3), comme le montre le listage 4.3.

```
strsql2 = "Select count(*) from TblUsagers Where Usager='" + txtUsager.text + "' ";
...
strsql3 = "Select * from TblUsagers Where Usager='" + txtUsager.text + "' and
          MotPasse='" + txtMotPasse.text + "' ";
```

Listage 4.3 Requêtes SQL dynamiques pour l'authentification

3. **Page3 :** permet d'abord la recherche des produits grâce au champ d'entrée (txtItem). Ensuite, les opérations de recherche ainsi que les utilisateurs qui ont effectué les recherches seront enregistrés aux fins de statistique. Le listage 4.4 montre la construction de trois requêtes SQL : strsql4, strsql5 et strsql6.

```
strsql4 = "Select * from TblItems Where ItemDesc like '%" + txtItem.text + "%' ";
...
strsql5 = "Insert into TblRecherchesLog (PageWeb,Mots)values ('Items','" + txtItem.text + "') ";
...
strsql6 = "Insert into TblUsagersLog (IPAdresse,dateSearch)values   (Session.IPSource,'" +

          DateTime.text + "') ";
```

Listage 4.4 Requêtes SQL dynamiques pour la recherche des produits

Les demandes HTTP permettant de générer les requêtes SQL précédentes sont formalisées comme suit :

1. Si l'utilisateur de page1 saisit ('jean') dans le champ txtprenom et ('jacques') dans le champ txtprenom, le seuil est calculé grâce à la formule 4.2 et on obtient la demande HTTP :
   $DemHttp1$=<Seuil=3,3 ;PARAMS={'jean','jacques'}>.
2. Si l'utilisateur de page2 saisit ('admin') dans le champ txtUsager et ('123') dans le champ txtMotPasse, le seuil est calculé grâce à la formule 4.2 et on obtient la demande HTTP :
   $DemHttp2$=<Seuil=2,4 ;PARAMS={'admin','123'}>.
3. Si l'utilisateur de page3 saisit ('portable') dans txtItem, le seuil est calculé grâce à la formule 4.2 et on obtient demande HTTP :
   $DemHttp3$=<Seuil=4,8 ;PARAMS={'portable'}>.

Au moment de l'exécution, les requêtes SQL interceptées sont formalisées comme suit :

1. $strsql_1$=Select * from TblClients Where Prenom like '%jean%' and Nom like'%jacq%'
   $ReqSql_1$=<LITERALS={'jean','jacq'}>.
2. $strsql_2$=Select count(*) from TblUsagers Where Usager='admin'
   $ReqSql_2$=<LITERALS={'admin'}>.
3. $strsql_3$=Select * from TblUsagers Where Usager='admin' and Mot-Passe='123'
   $ReqSql_3$=<LITERALS={'admin','123'}>.
4. $strql_4$ = Select * from TblItems Where ItemDesc like '%portable%'
   $ReqSql_4$=<LITERALS={'portable'}>.
5. $strql_5$ = Insert into TblRechercheLogs (PageWeb,Mots)values ('Items','portable')
   $ReqSql_5$=<LITERALS={'Items','portable'}>.
6. $strsql_6$ = Insert into TblUsagersLog (IPAdresse,dateSearch)values ('192.168.10.1', '02/10/2012 16 :30')
   $ReqSql_6$=<LITERALS={'192.168.10.1','02/10/2012 16 :30'}>.

Le score de mappage entre ($DemHttp_1$) et ($ReqSql_1$) est calculé comme suit :

$$
\begin{aligned}
Score(DemHttp_1, ReqSql_1) &= \frac{\sum_{i=1}^{2} \min_{j=1}^{2} Dist(Param_i, Litteral_j)}{n} \\
&= \frac{\min(Dist('jean','jean'), Dist('jean','jacq')) + \min(Dist('jacques','jean'), Dist('jacques','jacq'))}{2} \\
&= \frac{min(0,3) + min(6,3)}{2} \\
&= \frac{0+3}{2} \\
&= 1,5
\end{aligned}
$$

Les autres scores de mappage sont calculés grâce à la formule 4.1 comme suit :

$$Score(DemHttp_2, ReqSql1) = \frac{Dist('admin','jean') + Dist('123','jean')}{2} = \frac{5+3}{2} = 4$$
$$Score(DemHttp_3, ReqSql1) = \frac{Dist('portable','jean')}{1} = \frac{7}{1} = 7$$
$$Score(DemHttp_1, ReqSql2) = \frac{Dist('jean','admin') + Dist('jacques','admin')}{2} = \frac{4+6}{2} = 5$$
$$Score(DemHttp_2, ReqSql2) = \frac{Dist('admin','admin') + Dist('123','admin')}{2} = \frac{0+3}{2} = 1,5$$
$$Score(DemHttp_3, ReqSql2) = \frac{Dist('portable','admin')}{1} = \frac{8}{1} = 8$$
$$Score(DemHttp_1, ReqSql3) = \frac{Dist('jean','123') + Dist('jacques','123')}{2} = \frac{4+7}{2} = 5,5$$
$$Score(DemHttp_2, ReqSql3) = \frac{Dist('admin','admin') + Dist('123','123')}{2} = \frac{0+0}{2} = 0$$
$$Score(DemHttp_3, ReqSql3) = \frac{Dist('portable','admin')}{1} = \frac{8}{1} = 8$$
$$Score(DemHttp_1, ReqSql4) = \frac{Dist('jean','portable') + Dist('jacques','portable')}{2} = \frac{7+8}{2} = 7,5$$
$$Score(DemHttp_2, ReqSql4) = \frac{Dist('admin','portable') + Dist('123','portable')}{2} = \frac{8+8}{2} = 8$$
$$Score(DemHttp_3, ReqSql4) = \frac{Dist('portable','portable')}{1} = \frac{0}{1} = 0$$
$$Score(DemHttp_1, ReqSql5) = \frac{Dist('jean','Items') + Dist('jacques','Items')}{2} = \frac{4+6}{2} = 5$$
$$Score(DemHttp_2, ReqSql5) = \frac{Dist('admin','Items') + Dist('123','Items')}{2} = \frac{5+3}{2} = 4$$
$$Score(DemHttp_3, ReqSql5) = \frac{Dist('portable','portable')}{1} = \frac{0}{1} = 0$$
$$Score(DemHttp_1, ReqSql6) = \frac{Dist('jean','192.168.10.1') + Dist('jacques','192.168.10.1')}{2} = \frac{12+12}{2} = 12$$
$$Score(DemHttp_2, ReqSql6) = \frac{Dist('admin','192.168.10.1') + Dist('123','192.168.10.1')}{2} = \frac{12+10}{2} = 11$$
$$Score(DemHttp_3, ReqSql6) = \frac{Dist('portable','192.168.10.1')}{1} = \frac{12}{1} = 12$$

Nous obtenons alors la matrice de similarité suivante :

Score	Seuil	$ReqSql_1$	$ReqSql_2$	$ReqSql_3$	$ReqSql_4$	$ReqSql_5$	$ReqSql_6$
$DemHttp_1$	3,3	**1,5**	5	5,5	7,5	5	12
$DemHttp_2$	2,4	4	**1,5**	**0**	8	4	11
$DemHttp_3$	4,8	7	8	8	**0**	**0**	12

Tableau 4.1 Matrice de similarité HTTP et SQL

Chacune des requêtes SQL précédentes devra être associée à la demande HTTP qui correspond au score minimal :

1. $ReqSql_1$ est associée à $DemHttp_1$.
2. $ReqSql_2$ et $ReqSql_3$ sont associées à $DemHttp_2$.
3. $ReqSql_4$ et $ReqSql_5$ sont associées à $DemHttp_3$.
4. $ReqSql_6$ est indépendante de $DemHttp_1$, $DemHttp_2$ et $DemHttp_3$ puisque $Score(DemHttp_1, ReqSql6)$, $Score(DemHttp_2, ReqSql6)$ et $Score(DemHttp_3, ReqSql6)$ sont respectivement plus grands que Seuil($DemHttp_1$), Seuil($DemHttp_2$) et Seuil($DemHttp_3$). $ReqSql_6$ sera donc ignorée, car elle ne représente vraisemblablement pas un risque d'attaque.

4.4 Modélisation des comportements normaux

Dans un IDS, le profil se compose généralement d'un certain nombre de mesures statistiques sur les activités du système comme l'utilisation de CPU et la fréquence de commandes. Ces mesures ne sont pas toujours précises et pertinentes, ce qui peut augmenter les faux positifs et négatifs. Pourtant, la modélisation des profils doit être rigoureuse pour éviter de modéliser des profils normaux qui sont en réalité des attaques, ce qui à pour effet de multiplier le nombre d'attaques non détectées. C'est pourquoi, toutes les demandes HTTP qui peuvent contenir des attaques sont rejetées lors de la phase d'apprentissage. En revanche, les requêtes SQL liées aux demandes HTTP normales seront modélisées dans la base de connaissances (DB).

La modélisation des requêtes SQL au moment de la phase d'apprentissage se fait comme suit :

1. Interception et analyse d'une demande HTTP (DemHttp), selon l'algorithme 4.2 :
   (a) Si un des paramètres d'entrée de (DemHttp) contient un caractère dangereux, la demande HTTP sera annulée.

(b) Sinon, le seuil d'acceptation sera calculé et la demande HTTP (DemHttp) sera ajoutée à la liste des demandes HTTP (lstDemHttp) de la session correspondante.

2. Interception et analyse d'une requête SQL (ReqSql), selon l'algorithme 4.3 :
   (a) Extraire les littéraux de la requête SQL à l'aide d'un analyseur lexical SQL (ligne 1).
   (b) Générer le modèle SQL relatif à la requête SQL (lignes 3-6).
   (c) Sélectionner la demande HTTP (This_demhttp) qui correspond au score minimal de mappage au cas où ce score serait plus petit que le seuil d'acceptation (lignes 7-21).
   (d) Insérer la requête SQL (ReqSql) dans la liste des requêtes SQL (lstReqSql) de la demande HTTP déjà sélectionnée (lignes 22-25).
3. Modélisation des requêtes SQL qui ont été exécutées durant une session expirée (algorithme 4.4). Pour chaque demande HTTP (demhttp) de la session expirée :
   (a) Construire la liste ordonnée des templates (lstTemplate). Une boucle est utilisée pour passer à travers la liste (lstReqSql) des requêtes SQL associées à la demande HTTP (lignes 2-4).
   (b) Récupérer l'ensemble de listes des templates de l'URL de la demande HTTP (demhttp.URL) à partir de la base de connaissances (ligne 5).
   (c) Si la liste des templates (lstTemplate) construite dans (a) ne correspond à aucune liste des templates récupérées dans (b), lstTemplate sera ajoutée dans la base de connaissances (lignes 6-18).

Algorithme 4.2: Analyse d'une demande HTTP lors de la phase d'apprentissage

Input : DemHttp =< URL ;TimeStamp ;Seuil ;PARAMS ;Type >.

Input : la session (session) durant laquelle la demande HTTP (DemHttp) a été lancée.

Result : si la demande HTTP (DemHttp) est normale, elle est ajoutée (après le calcul du seuil d'acceptation) à la liste des demandes HTTP (lstDemHttp) de la session correspondante.

```
1  Seuil ← 0;
2  Regex NormaleRegex = new Regex(@"[0-9a-zA-Z]");
3  for i ← 1 to n do
4  |  if not NormaleRegex.IsMatch(param_i) then
5  |  |  DemHttp.abondon();
6  |  |  Return ;
7  |  else
8  |  |  Seuil=Seuil+param_i.Length ;
9  |  end
10 end
11 Seuil ← 0,6 × (Seuil/n) ;
12 session.lstDemHttp.Insert(DemHttp);
```

Algorithme 4.3: Analyse d'une requête SQL lors de la phase d'apprentissage

Input : chaîne de caractères (strSQL) qui représente une requête SQL interceptée.

Input : la date et le temps d'occurrence de la requête SQL (TimeStamp).

Input : l'ensemble de sessions non expirées (SESSION).

Input : la fenêtre de temps pour que le score de mappage soit calculé (t).

Result : les littéraux sont extraits, le template est généré et la requête ReqSql =< Requete ;Template ;Ordre ;TimeStamp ;LITERALS > est ajoutée à la liste des requêtes SQL de la demande HTTP la plus similaire.

```
1  LITERALS ← Analyse_lexical(strSQL);
2  Requete ← strSQL;
3  Template ← strSQL;
4  foreach literal in LITERALS do
5  |  Template=Replace(Template,literal,'?');
6  end
7  MinScore ← MAX_FLOAT;
8  foreach session in SESSION do
9  |  foreach demhttp in session.lstDemHttp do
10 |  |  if ReqSql.TimeStamp > demhttp.TimeStamp and
   |  |  ReqSql.TimeStamp < demhttp.TimeStamp + t then
11 |  |  |  ScoreMapp ← Score(demhttp, ReqSql);
12 |  |  |  if ScoreMapp < MinScore and ScoreMapp <= demhttp.Seuil
   |  |  |  then
13 |  |  |  |  MinScore ← ScoreMapp;
14 |  |  |  |  This_Session ← session;
15 |  |  |  |  This_demhttp ← demhttp;
16 |  |  |  end
17 |  |  else
18 |  |  |  Exit ;
19 |  |  end
20 |  end
21 end
22 if MinScore! = MAX_FLOAT then
23 |  Ordre ← This_demhttp.lstReqSql.Count;
24 |  This_Session.This_demhttp.lstReqSql.Insert(ReqSql);
25 end
```

Algorithme 4.4: Modélisation des requêtes SQL liées à une session expirée

Input : une session expirée (session).

Result : pour chaque (DemHttp) de la session, les requêtes SQL sont modélisées dans la base de connaissances (DB).

```
1  foreach demhttp in session.lstDemHttp do
2      foreach reqsql in demhttp.lstReqSql do
3          lstTemplate.Insert(reqsql.Template) ;
4      end
5      {lstTemplate_1, ..., lstTemplate_l} ← DB.GetlstTemplates(URL =
       demhttp.URL);
6      for i ← 1 to l do
7          for j ← 1 to lstTemplate.Count do
8              if lstTemplate_i.Template_j ! = lstTemplate.Template_j then
9                  Exit ;
10             end
11         end
12         if j = lstTemplate.Count then
13             Exit ;
14         end
15     end
16     if i = l and j = lstTemplate.Count then
17         DB.Insert(DemHttp.URL,lstTemplate);
18     end
19 end
```

4.4.1 Exemple

Supposons que l'utilisateur a saisi 'admin' et '123' dans les champs d'entrée du formulaire d'authentification (voir figure 2.2), la modélisation de ce cas s'exécute en trois étapes :

1. Le capteur HTTP intercepte la demande HTTP normale (voir listage 4.5)

sur laquelle l'algorithme 4.2 sera appliqué.

2. Le capteur SQL intercepte la requête SQL, illustrée dans la figure 4.3 :
   Select * from TblUsagers Where Usager='admin' and MotPasse='123" ;
   Les littéraux (admin et 123) sont les enfants des nœuds libellés "literal" et seront remplacés par ' ?' pour obtenir le modèle SQL suivant :
   Select * from TblUsagers Where Usager=' ?' and MotPasse=' ?'.
3. Le générateur de profils ajoute une liste des templates ne contenant que le template précédent aux listes des templates de l'URL représentant l'adresse de la page d'authentification.

```
GET /Default.aspx?Usager=admin&MotPasse=123 HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, ...
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 ...
Host: localhost
Cookie: ASP.NET_SessionId=xshitibm0r1nlpawjvwfzn55
Connection: Keep-Alive
```

Listage 4.5 Exemple d'une demande HTTP normale

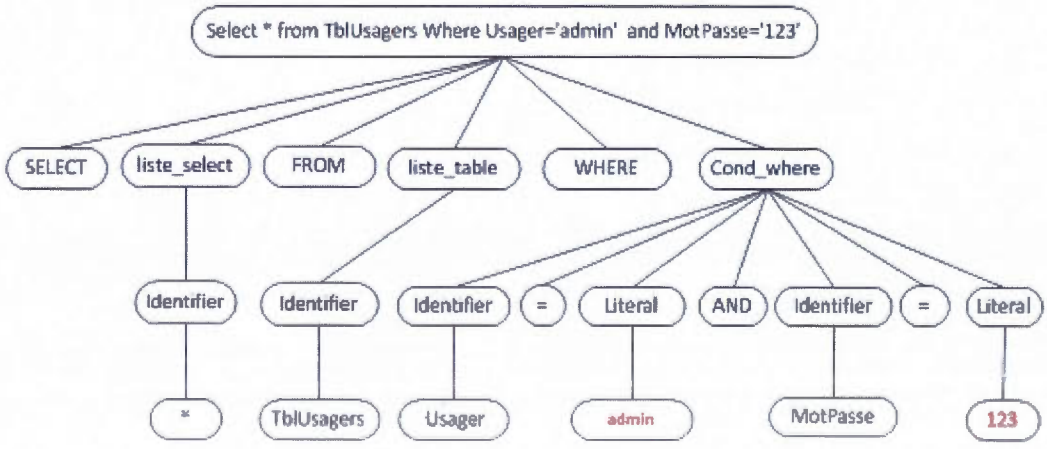

Figure 4.3 Arbre syntaxique d'une requête SQL normale

4.5 Identification et détection des attaques

Une manière de détecter des intrusions consiste à identifier des déviations dans le comportement des événements surveillés. Dans notre cas, les événements qui visent la base de l'application Web sont les requêtes SQL et les déviations sont celles qui n'ont pas été modélisées dans la base de connaissances (DB).

Selon l'analyse et les exemples du deuxième chapitre, il est impossible de perpétrer une attaque sans injecter des caractères dangereux (', –, /*.*/, etc.) dans les paramètres d'entrée puisque c'est la seule manière pour pouvoir changer la structure ou l'arbre syntaxique d'une requête SQL au moment de l'exécution. C'est pourquoi, pour une demande HTTP donnée, nous allons vérifier dans un premier temps si elle est contaminée (au moins un de ces paramètres d'entrée contient un caractère dangereux) afin de fournir un indice d'attaque. Si c'est le cas, une alerte sera levée lorsque nous trouvons une requête SQL, produite par la demande HTTP, qui n'a pas été modélisée dans la base de connaissances (DB).

L'identification et la détection des attaques lors de la phase de protection se font comme suit :

1. Interception et analyse d'une demande HTTP (DemHttp), selon l'algorithme 4.5 :
   (a) Si un des paramètres d'entrée de (DemHttp) contient un caractère dangereux, la demande sera classifiée comme contaminée (possibilité d'attaque).
   (b) Calculer le seuil d'acceptation et ajouter la demande HTTP (DemHttp) à la liste des demandes HTTP (lstDemHttp) de la session correspondante.
2. Interception et analyse d'une requête SQL (ReqSql), selon l'algorithme 4.3.
3. Identification et détection d'une attaque en vérifiant si une requête

SQL donnée (ReqSql) est malveillante (algorithme 4.6). Soit DemHttp la demande HTTP qui a produit (ReqSql) :

(a) Si DemHttp est normale alors ReqSql est aussi normale (lignes 1-3).

(b) Si DemHttp est contaminée (lignes 4-10) :

- Vérifier si le template de (ReqSql) n'a pas été modélisé dans la base de connaissances (DB) selon le même ordre et pour le même URL (ligne 5).
- Si c'est le cas, (ReqSql) sera classifiée comme malveillante et sera ajoutée dans la base d'attaques (lignes 6-9).

Algorithme 4.5: Analyse d'une demande HTTP lors de la phase de protection

Input : DemHttp =< URL ;TimeStamp ;Seuil ;PARAMS ;Type >.

Input : la session (session) durant laquelle la demande HTTP (DemHttp) a été lancée.

Result : la demande HTTP (DemHttp) est insérée sur la liste des demandes de la session correspondante.

```
1  Seuil ← 0;
2  Regex NormaleRegex = new Regex(@"[0-9a-zA-Z]");
3  for i ← 1 to n do
4  |   if not NormaleRegex.IsMatch(param_i) then
5  |   |   DemHttp.Type ← Contaminée;
6  |   end
7  |   Seuil=Seuil+param_i.Length ;
8  end
9  Seuil ← 0,6 × (Seuil/n) ;
10 session.lstDemHttp.Insert(DemHttp) ;
```

Algorithme 4.6: Identification et détection des attaques

```
Input : une requête SQL, ReqSql=< Requete ;Template ;Ordre ;TimeStamp ;LITERALS >.
Input : la demande HTTP (DemHttp) qui a produit ReqSql.
Input : la session (session) durant laquelle la demande HTTP (DemHttp) a été lancée.
Output : True : si la requête SQL (ReqSql) est malveillante. Sinon, False.
1  if DemHttp.Type = Normale then
2  |  return False;
3  end
4  if DemHttp.Type = Contaminee then
5  |  TEMPLATES ←
   |  DB.GetTemplates(URL=DemHttp.URL and Template=ReqSql.Template and Ordre=ReqSql.Ordre);
6  |  if TEMPLATES.Count = 0 then
7  |  |  AttacksDB.Insert(DemHttp.URL,ReqSql.Requete,session.IPSource,ReqSql.TimeStamp);
8  |  |  return True;
9  |  end
10 end
```

4.5.1 Exemple

Supposons qu'un attaquant a saisi (admin' or 1=1 - -) et (123) dans les champs d'entrée du formulaire d'authentification (voir figure 2.2) pour monter une tautologie. La détection de l'attaque s'exécute en trois étapes :

1. Le capteur HTTP intercepte la demande HTTP (voir listage 4.6) qui sera classifiée comme contaminée puisque le premier paramètre contient des caractères dangereux (' et - -).

2. Le capteur SQL intercepte la requête SQL, illustrée dans la figure 4.4 :
   Select * from TblUsagers Where Usager='admin' or 1=1 – and MotPasse='123' ;
   Les littéraux ('admin', 1 et 1) sont les enfants des nœuds libellés "literal" et seront remplacés par '?' et le code, après les deux caractères de commentaire (–), sera ignoré. Nous obtenons alors le template suivant :
   Select * from TblUsagers Where Usager='?' or ?=? .

3. Le détecteur d'attaques confirme que la requête SQL (voir figure 4.4) est malveillante puisque la demande HTTP est contaminée et le template précédent est différent de celui qui a été modélisé.

```
GET /Default.aspx?Usager=admin' or 1=1 --&MotPasse=123 HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, ...
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 ...
Host: localhost
Cookie: ASP.NET_SessionId=xshitibm0r1nlpawjvwfzn55
Connection: Keep-Alive
```

Listage 4.6 Exemple d'une demande HTTP contaminée

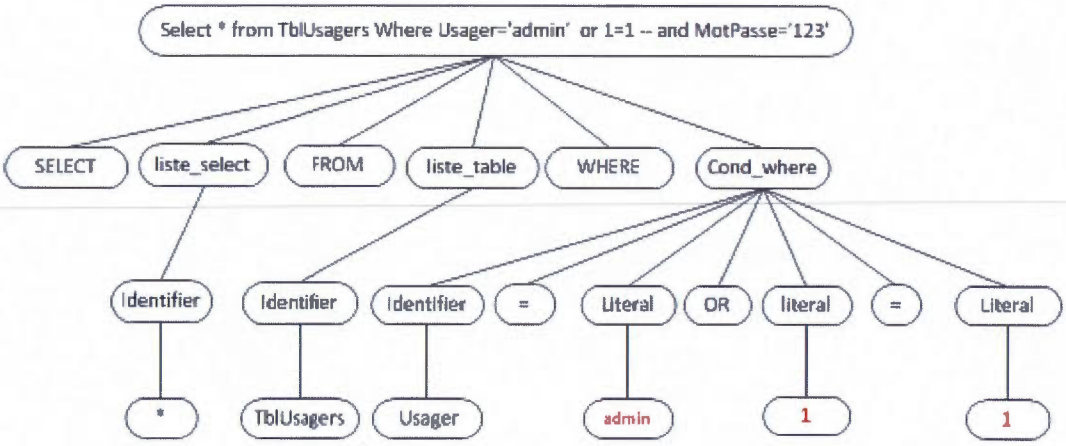

Figure 4.4 Arbre syntaxique d'une requête SQL malveillante

Conclusion

Dans notre proposition, nous avons essayé d'aller plus loin que l'approche implémentée dans AMNESIA en évitant d'accéder au code source. Cela nous a permis d'améliorer la portabilité et la facilité de mettre en œuvre de l'approche proposée. Cette approche, malgré ses avantages en termes de coût, de portabilité

et de facilité de mise en place, présente quelques limitations :

- Elle se limite aux attaques par injection SQL dans lesquelles l'attaquant tente de changer la structure des requêtes dynamiques. Elle est incapable, comme toutes les approches étudiées, de détecter les injections SQL à l'aveuglette.
- La durée d'apprentissage pourrait être longue dû au fait que toute demande HTTP contenant un caractère dangereux sera immédiatement annulée. Nous avons donc décidé de ralentir l'apprentissage pour augmenter le degré de précision.

CHAPITRE V

SQLIA-IDS :UNE MISE EN OEUVRE D'UN SYSTÈME DE DÉTECTION D'INTRUSIONS CONTRE L'ATTAQUE PAR INJECTION SQL

Nous appelons SQLIA-IDS notre mise en œuvre de l'approche proposée dans le chapitre précédent. Cette mise en œuvre nous permet donc d'évaluer pratiquement notre approche et de valider la faisabilité de ses algorithmes.

Ce chapitre est consacré à l'implémentation de SQLIA-IDS. Tout d'abord, nous introduisons le système en parlant de sa conception générale. Ensuite, nous présentons le fonctionnement et la mise en œuvre de chaque module de SQLIA-IDS. Finalement, nous terminons ce chapitre par une discussion sur la synchronisation de ces modules et des serveurs Web.

5.1 Survol du système

La présente section donne un aperçu général de la conception du système telle que le langage de programmation, les modules, la structure en mémoire, les bases de données et la configuration. SQLIA-IDS, dans sa version actuelle, supporte les applications Web développées en .NET qui accèdent à des bases de données MS SQL.

5.1.1 Choix du langage de programmation

Bien que le développement de SQLIA-IDS n'impose pas des contraintes techniques précises, nous avons finalement opté pour une implémentation faite avec le langage de programmation C# intégré dans l'environnement de développement (Microsoft Visual Studio 2010) de l'entreprise Microsoft car :

- il est complètement orienté objet et permet de concevoir facilement les listes dynamiques d'objets ;
- il gère automatiquement la mémoire en libérant l'espace alloué par des objets qui ne sont plus utilisés ;
- il permet une conception plus aisée des interfaces graphiques et des rapports.

5.1.2 Composants du système

SQLIA-IDS comporte cinq modules fonctionnels, une structure en mémoire, deux bases de données et une expression régulière. L'extrait simplifié du diagramme de classes de la figure 5.1 nous montre les modules fonctionnels et les classes principales de l'implémentation :

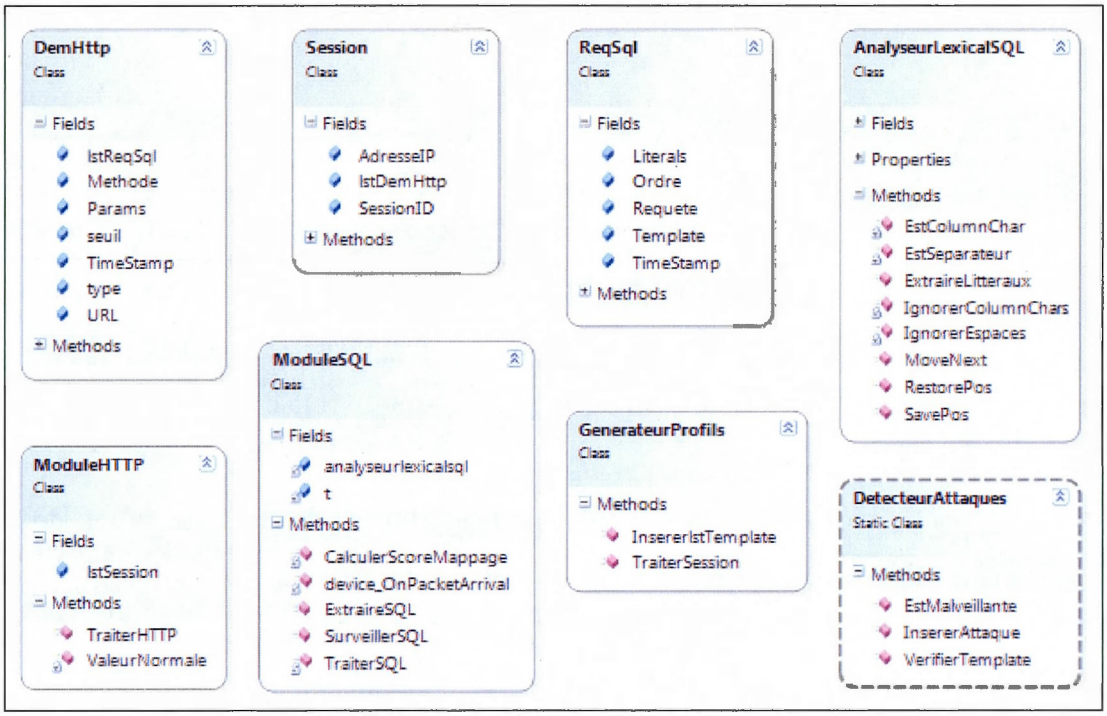

Figure 5.1 Extrait simplifié du diagramme de classes de SQLIA-IDS

Modules fonctionnels

SQLIA-IDS comprend cinq modules qui communiqueraient entre eux à travers une structure en mémoire et qui accéderaient à deux bases de données :

- Un module de surveillance HTTP, nommé ModuleHTTP, qui correspond au capteur HTTP du chapitre précédent.
- Un module de surveillance SQL, nommé ModuleSQL, qui correspond au capteur SQL du chapitre précédent.
- Un analyseur lexical SQL, nommé AnalyseurLexicalSQL, qui produit une séquence d'unités lexicales (jetons) représentant une instruction SQL.
- Un générateur de profils normaux, nommé GenerateurProfils, qui correspond au générateur de profils du chapitre précédent.
- Un détecteur d'attaques, nommé DetecteurAttaques, qui correspond au

détecteur d'attaques du chapitre précédent.

Structure en mémoire

Les cinq modules précédents partagent une structure temporelle en mémoire, illustrée par la figure 5.2 ; pour stocker les objets liés aux sessions, aux demandes HTTP et aux requêtes SQL. Ces objets sont instanciés respectivement de la classe Session, la classe DemHttp et la classe ReqSql (voir figure 5.1). Voici une description de ces classes :

- La classe Session est composée de certains attributs tels que l'identifiant de la session (SessionID), l'adresse IP du client (AdresseIP) et la liste ordonnée selon le TimeStamp des demandes HTTP (lstDemHttp). Le module ModuleHTTP construit un objet Session au démarrage d'une nouvelle session client.
- La classe DemHttp comprend certains attributs tels que la liste des requêtes SQL ordonnées selon le TimeStamp (lstReqSql), les paramètres d'entrée (Params), le seuil d'acceptation (seuil), la date et le temps d'occurrence (TimeStamp), le type (normale ou contaminée) et l'adresse de la page Web (URL). Le module ModuleHTTP construit un objet DemHttp à l'arrivée d'une nouvelle demande HTTP avec un ou plusieurs paramètres d'entrée.
- La classe ReqSql est composée de certains attributs tels que la liste des littéraux (Literals), l'ordre (Ordre), le code de la requête SQL (Requete), le modèle relatif à la requête SQL (Template) et la date et le temps d'occurrence (TimeStamp). Le module ModuleSQL construit les objets (ReqSql) au fur et à mesure de la réception des requêtes SQL.

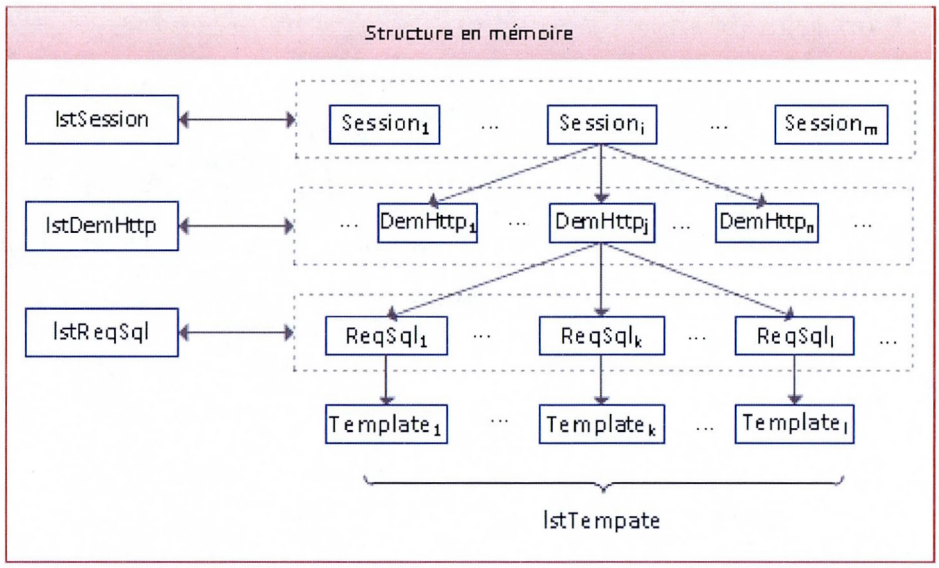

Figure 5.2 Structure en mémoire manipulée par SQLIA-IDS

Bases de données

SQLIA-IDS stocke les données persistantes dans deux bases :

- La base TemplatesDB, illustrée dans la figure C.1 de l'appendice C, correspond à la base de connaissances (DB) du chapitre précédent. Le module GenerateurProfils y accède en lecture et en écriture et le module DetecteurAttaques y accède en lecture. Une fois la phase d'apprentissage terminée, cette base est remplie avec les URLs et les modèles SQL.
- La base AttacksDB, illustrée dans la figure C.2 de l'appendice C, est constituée d'une seule table dans laquelle le module DetecteurAttaques enregistre les informations qui correspondent aux attaques détectées durant la phase de protection.

Expression régulière

Pour identifier les demandes HTTP normales ou celles qui peuvent être anormales, nous profitons de la technique des expressions régulières [AP02]. L'expression régulière, détaillée dans le tableau 5.1, valide si une chaîne de caractères ne contient que des chiffres ou des lettres.
Soit RegExStringNormale=@"[0-9a-zA-Z]".

RegExStringNormale=@"[0-9a-zA-Z]"	
sous-expression	caractère accepté
[0-9]	chiffre
[a-z]	lettre minuscule
[A-Z]	lettre majuscule

Tableau 5.1 L'expression régulière RegExStringNormale

5.1.3 Configuration

La figure 5.3 montre le formulaire qui permet à l'administrateur de configurer le système SQLIA-IDS et de générer des rapports de sécurité. Les éléments à configurer sont stockés dans le fichier de configuration (Web.config) de SQLIA-IDS et sont regroupés en quatre sections :

1. La section "Database Server" dans laquelle le serveur de données est configuré :
   - **Port number** : définit le numéro de port par lequel passe le trafic visant le serveur de données de l'application Web.
   - **IP address** : définit l'adresse IP du serveur de données de l'application Web.
2. La section "Web Server" dans laquelle le serveur Web est configuré :

- **Network Interface** : définit le dispositif du serveur Web pour se connecter au réseau.
- **IP address** : définit l'adresse IP du serveur Web sur lequel l'application Web a été installée.

3. La section "SQLIA-IDS database server" dans laquelle le serveur de données de SQLIA-IDS est configuré :
   - **Server name** : définit le nom du serveur de données sur lequel les deux bases TemplatesDb et AttacksDB sont installées.
4. La section "Training results/Phase" dans laquelle les résultats d'apprentissage s'affichent et où la phase d'exécution est configurée :
   - **Grille de résultats d'apprentissage** : cette grille affiche les résultats d'apprentissage, semaine par semaine, en montrant le nombre de comparaisons et le nombre de modèles SQL insérés. L'administrateur peut donc savoir si le système a atteint une stabilité d'apprentissage.
   - **Phase** : définit le mode de fonctionnement du système (apprentissage ou protection). L'administrateur règle le mode en se basant sur les résultats d'apprentissage.

Pour finir, le bouton (Security Report) offre la possibilité à l'administrateur d'afficher à l'écran un rapport détaillant les attaques par injection SQL détectées.

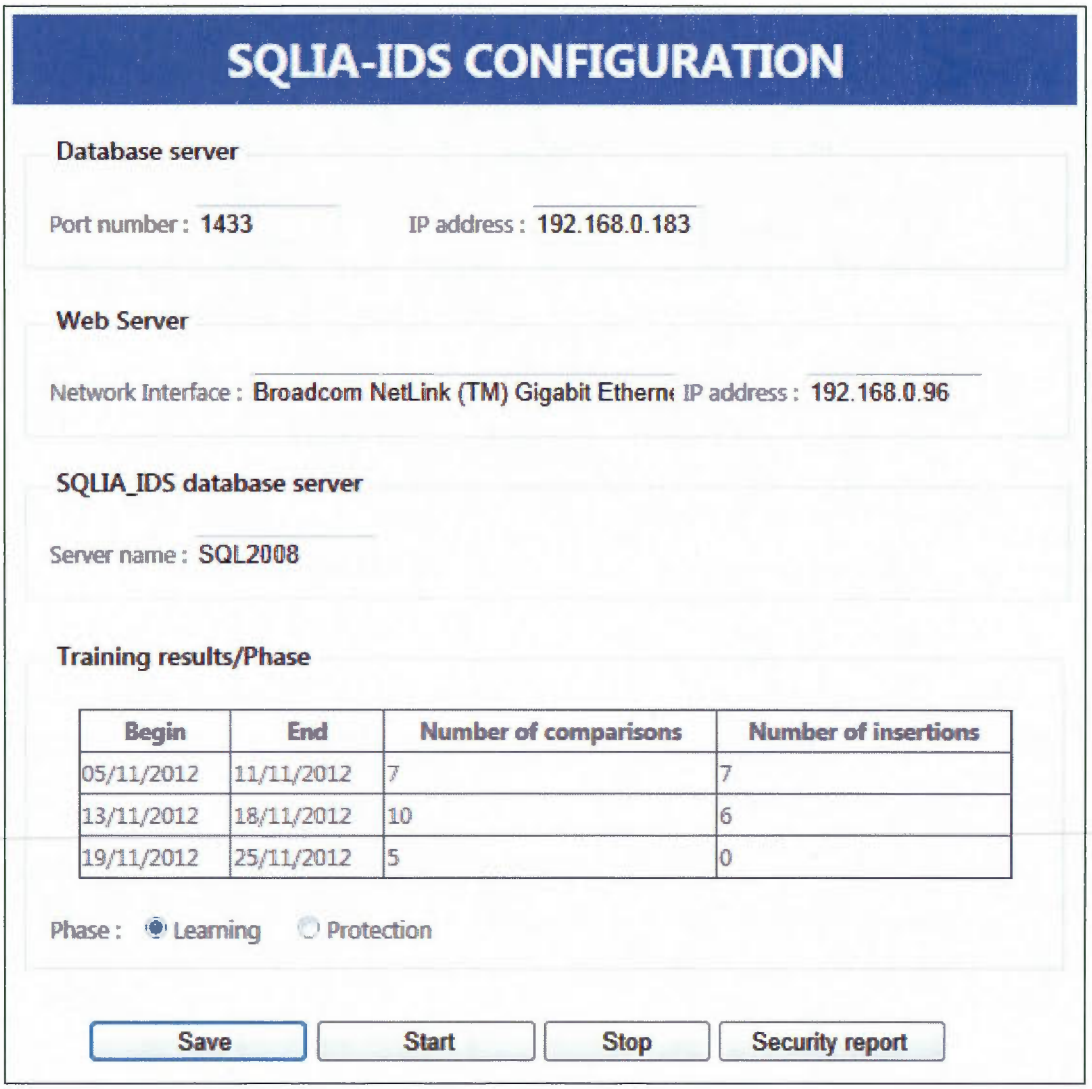

Figure 5.3 Formulaire de configuration de SQLIA-IDS

5.2 Module de surveillance HTTP

Le module de surveillance HTTP (ModuleHTTP) est celui qui récolte et qui traite les demandes HTTP afin de créer les objets associés à celles-ci et aux sessions clients. Dès que l'application Web reçoit une demande HTTP émise par le client, Global.aspx de .NET l'intercepte et l'envoie au ModuleHTTP qui

déclenche systématiquement le processus du traitement de la demande HTTP reçue. À titre de référence, nous fournissons dans le listage B.2 de l'appendice B le code source de ce module. La figure 5.4 montre l'architecture générale du ModuleHTTP.

Une fois que ModuleHTTP reçoit une demande HTTP avec au moins un paramètre d'entrée, il appelle la méthode (TraiterHTTP) qui fonctionne comme suit :

1. Elle extrait les paramètres d'entrée et vérifie à l'aide de l'expression régulière RegExStringNormale si l'un de ces paramètres contient un caractère dangereux. En ce cas :
   - Si le système fonctionne en phase d'apprentissage, la demande HTTP sera forcément abandonnée.
   - Si le système fonctionne en phase de protection, la demande HTTP sera considérée comme contaminée.
2. Elle extrait la session (SessionID), l'URL et le TimeStamp de la demande HTTP.
3. Elle calcule le seuil d'acceptation (seuil) qui est égal à 60% de la taille moyenne des paramètres.
4. Elle construit un objet DemHttp avec les informations extraites et le seuil calculé.
5. Si la session existe déjà, elle ajoute l'objet DemHttp à la liste des demandes HTTP (lstDemHttp) de cette session. Sinon, elle construit l'objet Session pour la nouvelle session à laquelle l'objet DemHttp sera ajouté.

Dans l'implémentation actuelle, ModuleHTTP dépend partiellement de la plateforme de développement adoptée (.NET) dans sa partie d'interception puisqu'il reçoit les demandes HTTP de la part du module Global.aspx de .NET.

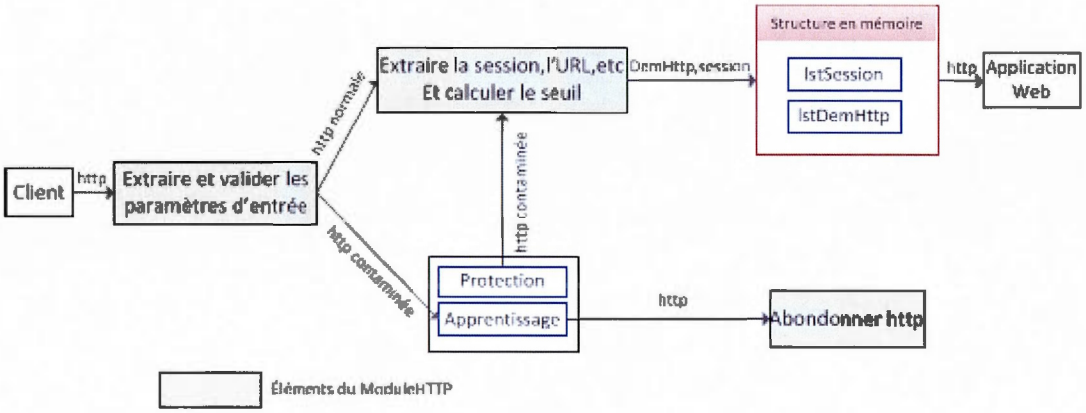

Figure 5.4 Architecture du module de surveillance HTTP

5.3 Module de surveillance SQL

Le module de surveillance SQL (ModuleSQL) est celui qui traite les requêtes SQL dynamiques. Il a été implémenté indépendamment de la plateforme de développement de l'application Web et du type de serveur Web pour pouvoir assurer la généricité. À titre de référence, nous fournissons dans le listage B.3 de l'appendice B le code source de ce module. La figure 5.5 montre l'architecture générale du ModuleSQL.

La capture des trames est effectuée à l'aide de la bibliothèque Winpcap [GVB12]. Comme le montre la figure 5.3, le système doit être préalablement configuré pour que ModuleSQL n'intercepte que les trames passant du serveur Web au serveur de données par un port spécifique selon le SGBD (1433 pour MS SQL).

Lorsque l'application Web démarre, la méthode (SurveillerSQL) est invoquée pour mettre ModuleSQL en attente jusqu'à l'arrivée d'une nouvelle trame liée au trafic SQL qui déclenche systématiquement l'exécution de la méthode (de-

vice_OnPacketArrival). En ce cas, ModuleSQL appelle la méthode (ExtraireSQL) qui récupère l'instruction SQL (strSQL) incluse dans la trame interceptée. Ensuite, la fonction (TraiterSQL) prend en paramètre (strSQL) et procède comme suit :

1. Elle passe la chaîne de caractères (strSQL) à l'analyseur lexical SQL afin d'extraire les littéraux. Si cette chaîne contient au moins un littéral, le modèle SQL relatif à l'instruction SQL (strSQL) sera généré en remplaçant les littéraux par le caractère ' ?'. Sinon, elle ne fait rien.
2. Elle sélectionne la demande HTTP (DemHttp) la plus similaire à la requête SQL :
   - Une boucle est alors utilisée pour itérer sur la liste des sessions (lstSession).
   - Pour une session consultée, nous calculons le score de mappage entre la requête SQL et les demandes HTTP de cette session en appelant la fonction (CalculerScoreMappage). Nous sélectionnons la demande HTTP qui correspond au score de mappage minimal, à condition que celui-ci soit plus petit que le seuil d'acceptation (seuil). Sinon, nous concluons que la requête SQL ne dépend pas des paramètres d'entrée et, par conséquent, elle sera ignorée.
3. À la suite de la sélection de (DemHttp), elle réagit en fonction de la phase d'exécution :
   - Dans la phase d'apprentissage, elle ajoute l'objet ReqSql à la liste des requêtes SQL (lstReqSql) de la demande HTTP (DemHttp).
   - Dans la phase de protection, elle envoie les deux objet ReqSql et DemHttp au détecteur d'attaques (DetecteurAttaques) qui déterminera ensuite si la requête SQL est malveillante.

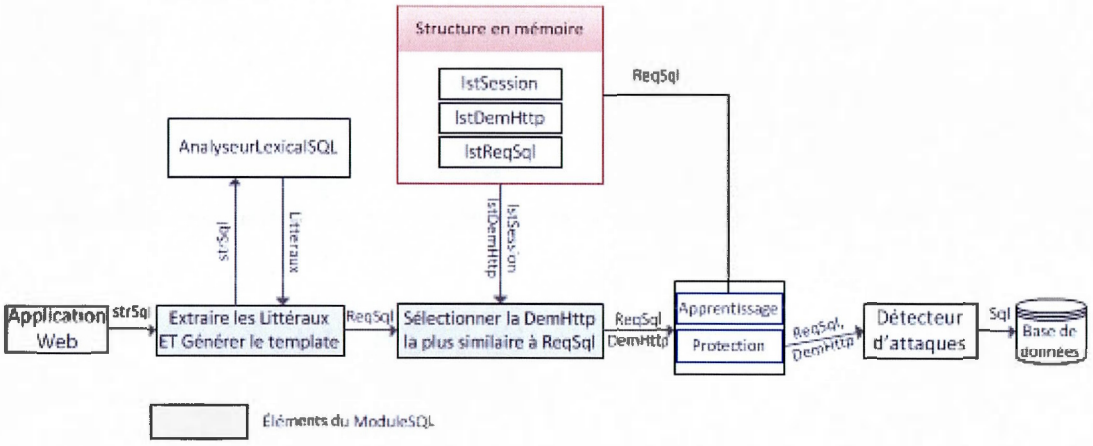

Figure 5.5 Architecture du module de surveillance SQL

5.4 Analyseur lexical SQL

SQLParser [Sof12] est un des meilleurs analyseurs lexicaux SQL. Il faut le payer et il dépend du SGBD ainsi que de l'environnement de développement. Ces caractéristiques entrent en contradiction avec notre exigence de portabilité (voir section 4.1). De plus, SQLParser fait d'un seul coup l'analyse syntaxique, lexicale et sémantique d'une requête SQL. Cela a une influence certaine sur la performance puisque SQLParser effectue des traitements inutiles au sein de notre système. Notons que notre besoin est d'extraire les littéraux d'une instruction SQL donnée.

Nous nous sommes donc inspirés du code implémenté dans [kod12] en développant un analyseur lexical simplifié et en l'adaptant à notre besoin. Cet analyseur rend notre système plus autonome, plus optimisé, plus contrôlable et indépendant du SGBD ainsi que de l'environnement de développement.

À titre de référence, nous fournissons dans le listage B.4 de l'appendice B le code source de ce module. Brièvement, notre analyseur lexical SQL (AnalyseurLexicalSQL) sert à prendre en entrée une instruction SQL sous forme d'une chaîne de caractères pour extraire les littéraux selon les règles lexicales suivantes :

- Les jetons sont des mots séparés par un opérateur (+, =, -, etc.) ou un séparateur (' ;', '.', etc.).
- Un littéral de type chaîne de caractères est un jeton qui débute par (') ou (") et qui se termine par un (') ou (").
- Un littéral de type numérique est un jeton qui ne contient que des chiffres.

Considérons par exemple la requête SQL suivante :
Select * from TblUsagers Where NomUsager='admin' and MotPasse='123'
Notre analyseur classifie 'admin' et '123' comme deux littéraux de type chaîne de caractères.

5.5 Générateur de profils normaux

Le générateur de profils normaux (GenerateurProfils) s'exécute uniquement pendant la phase d'apprentissage pour traiter les sessions expirées et modéliser les modèles SQL associés à celles-ci. À titre de référence, nous fournissons dans le listage B.5 de l'appendice B le code source de ce module. La figure 5.6 montre l'architecture générale du GenerateurProfils.

Dès que le module (GenerateurProfils) reçoit une session expirée de la part de l'application Web, il exécute la méthode (TraiterSession) qui consulte la liste des demandes HTTP (lstDemHttp) de cette session en traitant chacune de ces demandes HTTP (DemHttp) comme suit :

1. Elle construit la liste des modèles SQL (lstTemplate) produite par

(DemHttp) :

- Une boucle est utilisée pour itérer sur les objets (ReqSql) associés à (DemHttp) afin de récupérer les templates déjà générés et stockés par ModuleSQL.
- À la fin de la boucle, nous obtenons la liste ordonnée des templates (lstTemplate).

2. Elle exécute la fonction InsererlstTemplate qui fait appel à la procédure stockée InsererlstTemplate (voir listage B.6 de l'appendice B) qui à son tour vérifie si lstTemplate existe déjà dans la base TemplatesDB pour le même URL (DemHttp.URL). Sinon, elle enregistre la liste (lstTemplate) dans la base TemplatesDB.

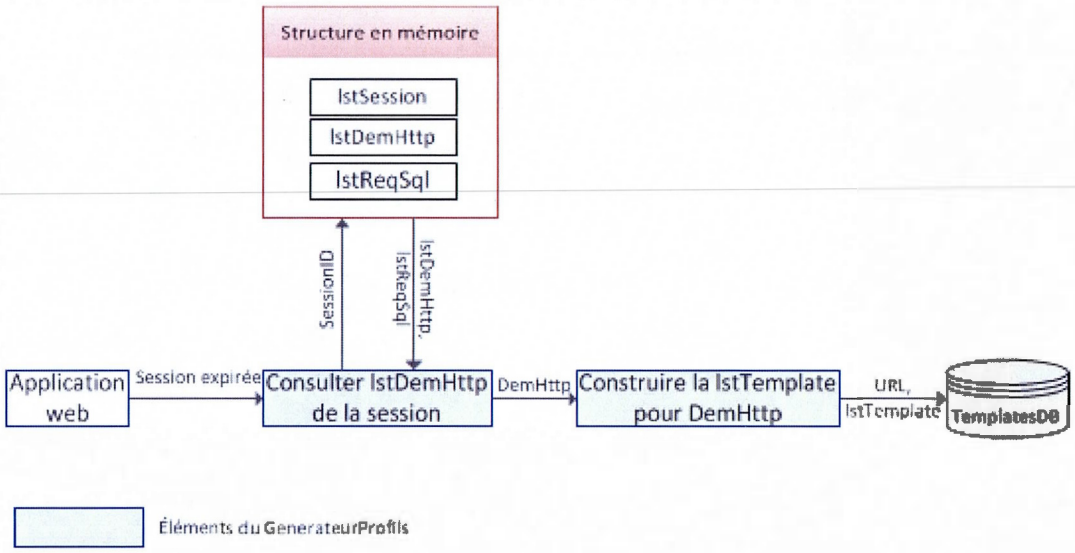

Figure 5.6 Architecture du générateur de profils normaux

5.6 Détecteur d'attaques

Le détecteur d'attaques (DetecteurAttaques) s'exécute pendant la phase de protection pour valider si une requête SQL est malveillante. À titre de référence,

nous fournissons dans le listage B.7 de l'appendice B le code source de ce module. La figure 5.7 montre l'architecture générale du DetecteurAttaques.

Dès que le module (DetecteurAttaques) reçoit une requête SQL de la part du ModuleSQL, il appelle la fonction (EstMalveillante) qui prend en paramètre les deux objets ReqSql et DemHttp et procède selon le type de DemHttp comme suit :

1. Si le type de DemHttp (DemHttp.type) est normal, elle retourne (False).
2. Si le type de DemHttp (DemHttp.type) est contaminé, elle exécute la procédure stockée VerifierTemplate (voir listage B.8 de l'appendice B) qui vérifie si le template (ReqSql.Template) existe dans le même ordre pour l'URL (DemHttp.URL) dans la base TemplatesDB :
   - Si c'est le cas, elle retourne (False).
   - Sinon, elle ajoute une attaque qui correspond à la requête SQL dans la base AttacksDB en faisant appel à la procédure stockée InsererAttaque (voir listage B.9 de l'appendice B) et retourne (True).

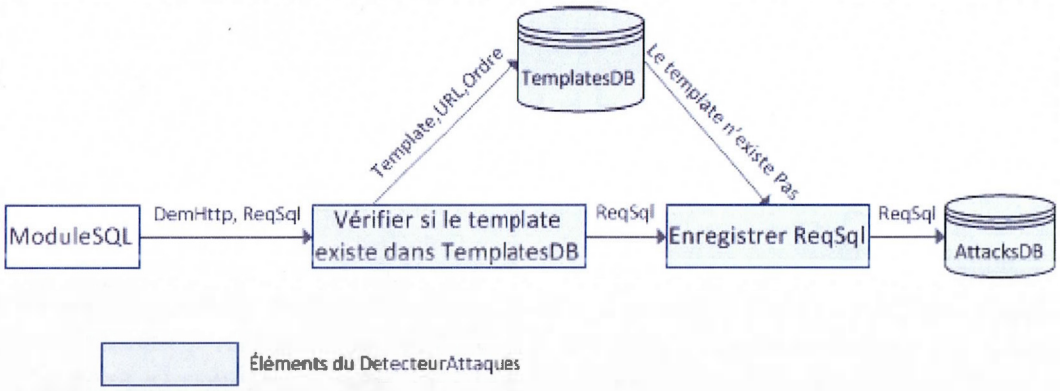

Figure 5.7 Architecture du détecteur d'attaques

5.7 Synchronisation des modules et des serveurs Web

Dans cette section, nous discutons deux aspects de synchronisation dans le cadre de SQLIA-IDS : la synchronisation des modules fonctionnels en termes d'accès aux objets en mémoire et la désynchronisation des horloges des serveurs Web.

5.7.1 Synchronisation des modules

SQLIA-IDS fonctionne dans un environnement multithread (synchronisation) puisqu'il déclenche plusieurs threads pour paralléliser l'exécution des modules sans trop ralentir les serveurs Web et les réponses aux demandes clients. En fait, nous ne voulons pas que deux threads (ou modules) différents puissent, en même temps, modifier les mêmes objets sans avoir à mettre en place des mécanismes de synchronisation (exclusion mutuelle, sémaphore, etc.) qui ne sont pas toujours efficaces.

Afin de comprendre comment les modules partagent les objets en mémoire, nous allons nous intéresser au tableau 5.2.

Objets / Modules	Session	DemHttp	ReqSql
ModuleHTTP	Lecture/Écriture	Lecture/Écriture	N/A
ModuleSQL	Lecture	Lecture	Écriture
GenerateurProfils	Lecture	Lecture	Lecture
DetecteurAttaques	Lecture	Lecture	Lecture

Tableau 5.2 Accès des modules de SQLIA-IDS aux objets en mémoire

D'après le tableau 5.2, chaque objet en mémoire est accessible en écriture par un seul module. Par contre, deux modules peuvent accéder en écriture et en lecture au même objet, mais pas en même temps. Autrement dit, une fois que

le flot d'exécution est rendu au GenerateurProfils, il accède en lecture à des objets Session et DemHttp associés à des demandes HTTP qui ont déjà été traitées préalablement par ModuleHTTP. Nous pouvons donc conclure que deux modules n'accèdent jamais en écriture et en lecture simultanément au même objet en mémoire. Par conséquent, les cinq modules de SQLIA-IDS accèdent à la structure en mémoire sans aucun problème de synchronisation.

5.7.2 Désynchronisation des horloges des serveurs Web

La désynchronisation représente un défi important pour les IDS qui se basent sur le temps pour corréler des événements si les horloges des nœuds d'un réseau ou des serveurs ne sont pas synchronisées. Étant donné que notre IDS et l'application Web se situent sur le même serveur Web (ou plusieurs serveurs Web), l'interception des demandes HTTP et des paquets contenant des requêtes SQL se fait pour une seule application Web selon l'horloge d'un seul serveur Web. Dans le cas de plusieurs serveurs Web, notre IDS ainsi que l'application Web seront installés sur tous ces serveurs et chaque serveur Web traitera les demandes des clients précis. Par conséquent, la désynchronisation des horloges ne cause pas de problèmes pour notre IDS même s'il utilise une méthode de corrélation temporelle.

Conclusion

Ce chapitre nous a permis d'expliciter l'implémentation de SQLIA-IDS, en termes de conception générale et en termes de modules. À présent, nous avons détaillé notre solution proposée et sa mise en œuvre. Nous allons présenter dans le chapitre suivant l'expérimentation et l'auto-évaluation de notre IDS.

CHAPITRE VI

RÉSULTATS EXPÉRIMENTAUX ET ÉVALUATION

L'objectif de ce chapitre est d'évaluer concrètement l'exhaustivité (taux de faux négatifs), la précision (taux de faux positifs) et la performance (consommation CPU) de notre IDS. Dans la première section, nous présentons l'environnement de test. Dans la deuxième section, nous détaillons les scénarios et les résultats expérimentaux. Dans la troisième section, nous évaluons notre approche selon les résultats obtenus en suivant les critères d'évaluation d'un IDS (voir section 1.3.1).

6.1 Environnement de test

Notre environnement de test est relativement simple et est constitué des composants suivants :

- Une application Web (en VB.NET) qui accède à une base de données (MS SQL) sans exécuter des procédures stockées.
- Un serveur Windows7 sur lequel nous avons installé un serveur Web IIS7, l'application Web, notre IDS (SQLIA-IDS) et un analyseur de performances (Microsoft) pour mesurer la consommation CPU.
- Un serveur de données (MS SQL) contenant la base de l'application Web.
- Un autre serveur de données (MS SQL) contenant les deux bases TemplatesDB

et AttacksDB de SQLIA-IDS.

- Un ordinateur client qui permet d'accéder à l'application Web par Internet Explorer 7.

6.2 Scénarios d'attaques et résultats

Pour évaluer SQLIA-IDS, nous avons lancé une centaine de demandes HTTP accédant, pour la plupart, à la base de données par des champs d'entrée des pages Web et contenant le nombre d'attaques présenté dans le tableau 6.1. Ces pages Web permettent l'authentification, la recherche et la génération dynamiques des formulaires à partir des informations existantes dans une base de données. Ces cas de figure nous permettent de mesurer la précision, l'exhaustivité et la performance de SQLIA-IDS.

Tautologie	Union de requêtes	Requête incorrecte	Commande SQL spécifique	Taux d'attaques
8	8	8	8	$\approx 32\%$

Tableau 6.1 Quantité d'attaques effectuées

Par la suite, nous allons tenter avec notre environnement de test de simuler trois scénarios d'attaques afin d'expérimenter l'exhaustivité, la précision, la performance et la mise à l'échelle de notre IDS. L'intérêt de ces scénarios est de déterminer les faiblesses de notre approche.

Il est important de noter que nous avons supposé dans tous nos tests que nous n'avons aucune connaissance préalable du code source de l'application Web.

6.2.1 Scénario d'une seule session sans IDS

Ce scénario a été réalisé sans l'activation de notre IDS. Nous avons exécuté environ 100 demandes HTTP, avec la quantité d'attaques du tableau 6.1, dans les pages Web de l'application et durant une seule session. Cela conduit, d'une part, à vérifier expérimentalement la vulnérabilité de l'application Web ; d'autre part, cela nous permet de mesurer la consommation CPU sur laquelle nous allons nous baser pour mesurer le coût de performance de notre IDS.

La grande majorité des attaques ont réussi. Cela confirme que l'application Web accède à la base de données en exécutant des requêtes dynamiques sans valider correctement les entrées d'utilisateur. Le serveur Web a consommé 14,8% du CPU pour pouvoir traiter les demandes HTTP visant l'application Web.

6.2.2 Scénario d'une seule session avec IDS

Dans ce test, nous avons rejoué le même jeu de trafic décrit précédemment, mais en activant notre IDS. Toutes les demandes HTTP ont été lancées durant une seule session client.

Comme le montre le tableau 6.2, la majorité des attaques ont été détectées. Notre IDS n'a pas détecté trois attaques et a engendré deux fausses alertes. Le serveur Web a consommé 17,4% du CPU pour pouvoir traiter les demandes HTTP, c'est-à-dire que la consommation CPU est passée de 14,8% à 17,4% en activant l'IDS.

6.2.3 Scénario de plusieurs sessions avec IDS

Nous avons rejoué le même trafic HTTP décrit précédemment durant trois sessions qui s'exécutent simultanément au lieu d'une session unique et en activant

Type d'attaque \ Nombres	Attaques	Attaques détectées	Faux négatifs	Faux positifs
Tautologie	8	8	0	0
Union de requêtes	8	7	1	1
Requête incorrecte	8	6	2	1
Commande SQL spécifique	8	8	0	0
Total	32	29	3	2

Tableau 6.2 Résultats de détection : une seule session avec IDS

notre IDS. Ce scénario est censé évaluer la mise à l'échelle de l'approche concernant la performance (consommation CPU) et la précision de notre méthode de corrélation.

Comme le montre le tableau 6.3, la majorité des attaques ont été détectées. Notre IDS n'a pas détecté quatre attaques et a engendré trois fausses alertes. Le serveur Web a consommé 17,7% du CPU pour pouvoir traiter les demandes HTTP, c'est-à-dire la consommation CPU est passée de 17,4% à 17,7% en répartissant le trafic HTTP sur plusieurs sessions.

Type d'attaque \ Nombres	Attaques	Attaques détectées	Faux négatifs	Faux positifs
Tautologie	8	7	1	1
Union de requêtes	8	7	1	1
Requête incorrecte	8	6	2	1
Commande SQL spécifique	8	8	0	0
Total	32	28	4	3

Tableau 6.3 Résultats de détection : plusieurs sessions avec IDS

6.3 Évaluation

Maintenant que nous avons vu les différentes parties de l'implémentation (chapitre 5) et les résultats expérimentaux, nous allons évaluer l'exhaustivité, la précision, la performance, la facilité de mise en place, la portabilité et l'autonomie de notre IDS.

Exhaustivité

Comme le montre les deux tableaux 6.2 et 6.3, l'exhaustivité de notre IDS est bonne puisque le taux de détection a varié entre 90,6% (une seule session) et 87,5% (plusieurs sessions).

Dans nos tests d'exhaustivité, l'IDS n'a pas réussi à détecter un pourcentage d'attaques (entre 9,4% et 12,5%). Cela est causé par l'analyseur lexical SQL qui n'a pas pu traiter tous les séparateurs (et tous les caractères) et qui a généré des erreurs d'exécution sans être capable d'extraire les littéraux de certaines requêtes SQL. Ces erreurs ont empêché les autres modules de continuer le traitement pour détecter certaines attaques. Une amélioration de l'analyseur lexical pourrait donc contribuer à diminuer le taux de faux négatifs.

Précision

Les résultats de nos tests de précision (voir tableaux 6.2 et 6.3) ont montré que l'IDS a engendré de faux positifs : 2% (une seule session) et 3% (plusieurs sessions). Ce taux de faux positifs nous semble acceptable puisque tous les IDS engendrent de faux positifs.

La comparaison entre les deux tableaux 6.2 et 6.3 montre que le taux de faux positifs a été augmenté proportionnellement au taux de faux négatifs, en

répartissant le trafic HTTP sur plusieurs sessions. Cela revient au fait que notre IDS n'a pas pu associer une ou plusieurs requêtes SQL aux bonnes demandes HTTP. Une amélioration de notre méthode de corrélation pourrait donc réduire le taux de faux positifs et négatifs.

Performance

Notre IDS introduit certains délais supplémentaires dans le traitement des requêtes. L'analyse comparative de la consommation CPU, illustrée dans le tableau 6.4, montre que la consommation CPU est passée de 14,8% à 17,4% dans le cas d'une session unique (avec IDS) et de 14,8% à 17,7% dans le cas de plusieurs sessions (avec IDS). Cela est acceptable en tenant compte qu'un IDS utilise généralement une quantité de CPU. Nos mesures sur la consommation CPU nous amènent à conclure que l'augmentation du nombre de sessions ne fait pas trop augmenter la consommation CPU.

	Consommation CPU moyenne (%)	Consommation CPU introduit (%)
Une seule session sans IDS	14,8	-
Une seule session avec IDS	17,4	2,6
Plusieurs sessions avec IDS	17,7	2,9

Tableau 6.4 Moyennes de consommation CPU

Facilité de mise en place

Nous pensons que l'avantage majeur de SQLIA-IDS versus les autres approches est sa facilité de mise en place grâce aux caractéristiques suivantes :

- il n'exige aucune modification ni lecture ni analyse du code source de l'application Web ;
- il est indépendant de la compétence du programmeur ;
- il est relativement simple à configurer par le formulaire 5.3 ;
- il ne nécessite aucune infrastructure supplémentaire (parser, serveur mandataire, gestion des clés, etc). Il suffit de télécharger gratuitement DotNetFrameWork 4.0 et de l'installer sur le serveur Web.

Portabilité

Le tableau 6.5 résume l'évaluation de la portabilité de chaque module de SQLIA-IDS par rapport aux plateformes telles que l'environnement de développement (.NET, J2EE, etc.), le fournisseur de données (MS SQL, MySQL, etc.) et le serveur Web (Apache, IIS, etc.). La valeur "Tous" signifie que le module est portable quelle que soit la plateforme.

Plateformes / Modules	Environnement de développement	Fournisseur de données	Serveur Web
ModuleHTTP	.NET(VB, C#, etc.)	Tous	Tous
ModuleSQL	Tous	MS SQL	Tous
AnalyseurLexicalSQL	Tous	Tous	Tous
GenerateurProfiles	Tous	Tous	Tous
DetecteurAttaques	Tous	Tous	Tous
SQLIA-IDS	.NET(VB, C#, etc.)	MS SQL	Tous

Tableau 6.5 Évaluation de la portabilité de SQLIA-IDS

Par conséquent, la version actuelle de SQLIA-IDS protège les applications Web développées en .NET (C#, VB, etc.), qui accèdent à des bases de données MS SQL et qui sont installées sur n'importe quel type du serveur Web.

D'après le tableau 6.5, nous remarquons que les trois modules Analyseur-LexicalSQL, GenerateurProfiles et DetecteurAttaques sont indépendants des plateformes. D'un autre côté, ModuleHTTP dépend de l'environnement de développement et ModuleSQL dépend de SGBD dans la partie d'interception. Cela nous amène à conclure que le coût nécessaire pour adapter SQLIA-IDS à toutes les plateformes est bas.

Pour rendre ModuleHTTP portable sur tous les environnements de développement, deux solutions peuvent être envisagées :

- La première solution est d'intercepter les paquets contenant les demandes HTTP (comme dans le cas des requêtes SQL). Cette solution est générique, mais l'inconvénient est qu'il va falloir filtrer les paquets liés au trafic HTTP qui visent l'application Web à protéger et à extraire les informations incluses dans ces paquets.
- La deuxième solution est de développer un module d'interception pour chaque environnement (J2EE, PHP, etc.). Cette solution est moins générique et plus performante. Par exemple, dans le cas de J2EE, le servlet Java intercepte et envoie les demandes HTTP au ModuleHTTP.

Pour que ModuleSQL puisse supporter tous les SGBDs, il suffit de savoir pour chaque SGBD la position de l'instruction SQL dans les paquets interceptés (54 pour MS SQL). Le numéro de port et l'adresse IP du serveur de données devront être préalablement configurés par le formulaire 5.3.

Autonomie

SQLIA-IDS est autonome puisqu'il a été implanté à la base de la détection d'anomalies. Il est donc capable de réaliser automatiquement (sans intervention humaine) l'apprentissage de comportement normal et la détection des attaques.

CONCLUSION

Au cours de ce mémoire, nous avons étudié en profondeur les différents types d'attaques par injection SQL. Nous avons proposé une solution automatisée et fondée sur la détection d'anomalies pour détecter ces attaques. Nous avons également implémenté et évalué, SQLIA-IDS, une mise en œuvre de notre proposition, qui est certainement facile à mettre en place, puisqu'elle n'exige aucun accès ou modification au code source de l'application Web à protéger. Bien que nous n'ayons pu faire fonctionner SQLIA-IDS avec tous les environnements et tous les SGBDs, elle pourrait être aisément adaptée à tous les environnements et à tous les SGBDs. Parmi les autres avantages de notre approche, notons :

- La capacité de détecter les attaques internes : les tentatives pour obtenir des informations confidentielles peuvent être réalisées non par un intrus extérieur, mais par un utilisateur déclaré d'une application Web. Notre IDS est capable de détecter ces types d'attaques.
- La détection des quatre types de SQLIA et toutes sortes d'attaques visant à modifier la structure des requêtes SQL dynamiques parce que nous modélisons les requêtes SQL normales produites par une application Web, et toute déviation sera considérée comme une attaque.
- Contrairement aux approches nécessitant une comparaison entre les arbres syntaxiques des requêtes SQL qui nous semble coûteuse en termes de performance, notre approche compare deux modèles SQL comme deux chaînes de caractères.

Contributions

L'apport principal de la présente recherche au domaine de la sécurité des applications Web est d'amener et de mettre en œuvre une solution automatisée face aux attaques par injection SQL. Parmi les contributions secondaires, nous pouvons citer :

- L'implémentation d'une plateforme qui peut être adaptable pour détecter d'autres attaques Web.
- La proposition d'une méthode de mise en relation entre le trafic HTTP et le trafic SQL. Cette méthode pourrait être réutilisable au sein d'autres algorithmes et pour d'autres applications de sécurité ainsi que pour identifier les utilisateurs finaux d'une application Web.
- La modélisation des requêtes dynamiques dans une base de données peut fournir un audit automatique du code source des applications Web.
- La démonstration concrète que les systèmes de détection d'intrusion dédiés aux applications Web, peu répandus, peuvent s'avérer utiles.

À l'heure actuelle, bien que nous n'ayons pas eu le temps de réaliser tout ce que nous avions prévu de faire, notre IDS est fonctionnel et viable. Plusieurs améliorations peuvent alors être envisagées. D'une part, il serait intéressant de mieux évaluer notre IDS en effectuant des expérimentations plus diversifiées, sur des applications Web plus sophistiquées et durant un temps plus long. D'autre part, il serait souhaitable de faire fonctionner notre IDS sous des environnements plus variés que .NET et MS SQL.

Plusieurs travaux futurs peuvent être envisagés pour compléter notre travail. Nous pensons particulièrement à la possibilité d'appliquer les techniques de fouille de données (clustering) afin de regrouper les requêtes SQL dynamiques pour chaque demande HTTP. Pour ce faire, notre mesure de similarité peut être

adaptée pour mesurer les ressemblances entre les requêtes SQL dynamiques selon les littéraux et les TimeStamps de celles-ci. L'intérêt soulevé précédemment vise à éviter l'interception et le traitement des demandes HTTP, ce qui rend notre IDS indépendant de tout environnement de développement.

APPENDICE A

DIAGRAMME DE CLASSES DE SQLIA-IDS

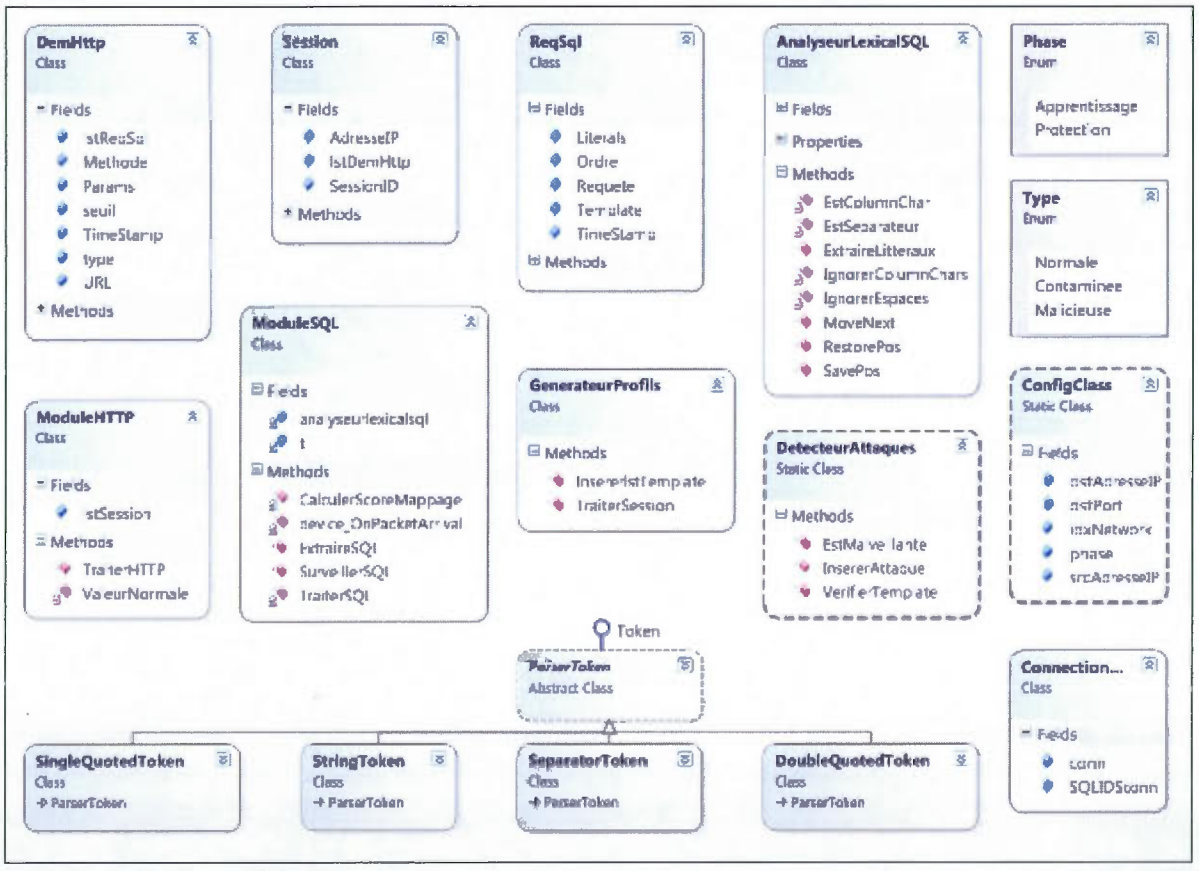

Figure A.1 DIAGRAMME DE CLASSES DE SQLIA-IDS

APPENDICE B

CODE SOURCE DE SQLIA-IDS

```
public static int Compute(string s, string t)
    {
        int n = s.Length;
        int m = t.Length;
        int[,] d = new int[n + 1, m + 1];
        // Step 1
        if (n == 0) return m;
        if (m == 0) return n;
        // Step 2
        for (int i = 0; i <= n; d[i, 0] = i++){ }
        for (int j = 0; j <= m; d[0, j] = j++){ }
        // Step 3
        for (int i = 1; i <= n; i++)
        {
            //Step 4
            for (int j = 1; j <= m; j++)
            {
                // Step 5
                int cost = (t[j - 1] == s[i - 1]) ? 0 : 1;
                // Step 6
                d[i, j] = Math.Min(
                    Math.Min(d[i - 1, j] + 1, d[i, j - 1] + 1),
                    d[i - 1, j - 1] + cost);
            }
        }
        // Step 7
        return d[n, m];
    }
```

Listage B.1 Algorithme de la distance d'édition en C#(tiré de [dot11])

```
using System;
using System.Collections.Generic;
using System.Web;
using System.Text.RegularExpressions;
namespace SQLIAIDS
{
  public enum Phase { Apprentissage, Protection};
  public enum Type { Normale, Contaminee };
  public class ModuleHTTP
  {
        /// <summary>
    /// Takes a http request and traits it.
    /// </summary>
   public static List<Session> lstSession = new List<Session>();
   public bool TraiterHTTP(HttpRequest p_http,HttpContext p_context )
   {
      if (p_http.Form.Count >= 4)
       {
            DemHttp tmphttp = new DemHttp();
        List<string> tmplstParams = new List<string>();
        for (int i = 2; i < p_http.Form.Count-1; i++)
        {
                    if (p_http.Form.Keys[i].ToString().Substring(0, 1) != "_")
                    {
                        if (!ValeurNormale(p_http.Form[i].ToString()))
                        {
                            if (ConfigClass.phase == Phase.Apprentissage) return false;
                            else tmphttp.type = Type.Contaminee;
                        }
                        tmplstParams.Add(p_http.Form[i].ToString());
                        tmphttp.seuil = tmphttp.seuil + p_http.Form[i].Length;
                    }
        }
        tmphttp.URL = p_http.Url.AbsoluteUri.Split('?')[0].ToString();
        tmphttp.TimeStamp = p_context.Timestamp;
        tmphttp.Params = tmplstParams;
        tmphttp.Methode = p_http.HttpMethod.ToString();
        tmphttp.seuil = (float) 0.6 * (tmphttp.seuil / tmplstParams.Count);
        Session result = lstSession.Find(item => item.SessionID == p_http.Params["ASP.NET_SessionId"]);
        if (result == null)
        {
                    result = new Session();
```

```
            result.SessionID = p_http.Params["ASP.NET_SessionId"];
            result.AdresseIP = p_http.UserHostAddress;
            result.lstDemHttp = new List<DemHttp>();
            result.lstDemHttp.Add(tmphttp);
            lstSession.Add(result);
          }
          else result.lstDemHttp.Add(tmphttp);
        }
        return true;
    }
        private bool ValeurNormale(string valeurparam)
    {
        System.Text.RegularExpressions.Regex RegExStringNormale = new Regex(@"^[a-zA-Z0-9]*$");
        return RegExStringNormale.IsMatch(valeurparam);
    }
  }
}
```

Listage B.2 Code source du ModuleHTTP

```
using System;
using SharpPcap;
using PacketDotNet;
namespace SQLIAIDS
{
    public class ModuleSQL
    {
        /// <summary>
        /// Intercept a packet contains a SQL Statement and traits it.
        /// </summary>
        static int t = 1000;
        AnalyseurLexicalSQL analyseurlexicalsql = new AnalyseurLexicalSQL();
        public void SurveillerSQL()
        {
            // Retrieve the device list
            var devices = CaptureDeviceList.Instance;
            // If no devices were found print an error
            if (devices.Count < 1) return;
            string capFile = "C:\\temp\\4.cap";
            var device = devices[ConfigClass.idxNetwork];
            // Register our handler function to the 'packet arrival' event
            device.OnPacketArrival += new PacketArrivalEventHandler(device_OnPacketArrival);
            // Open the device for capturing
            device.Open();
            // tcpdump filter to capture only TCP/TDS packets
            device.Filter = "dst port " + ConfigClass.dstPort + " and src " + ConfigClass.srcAdresseIP;
            // Open or create a capture output file
            if (!device.DumpOpened)
            {
                device.DumpOpen(capFile);
            }
            // Start capture 'INFINTE' number of packets
            device.Capture();
            // Close the pcap device
            device.DumpFlush();
            device.DumpClose();
            device.Close();
        }
        // Capture ant treat each received TDS packet
        private void device_OnPacketArrival(object sender, CaptureEventArgs e)
        {
            string strSQL = "";
```

```
    var device = (ICaptureDevice)sender;
    //if device has a dump file opened
    if (device.DumpOpened)
    {
        //dump the packet to the file
        device.Dump(e.Packet);
        int idx = 54;
        strSQL=ExtraireSQL(e.Packet.Data, ref idx);
        if (strSQL != "") TraiterSQL(strSQL,e.Packet.Timeval);
    }
}

public string ExtraireSQL(byte[] PacketData, ref int Index)
{
    string TmpSQL = "";
    byte type;

    type = PacketData[Index++];
    if (type == 1)
    {
       TmpSQL = "";
       int i = 84;
       while (i < PacketData.Length)
       {
           TmpSQL = TmpSQL + Convert.ToChar(PacketData[i]);
           i = i + 2;
       }
     }
  return TmpSQL;
}

// treat a SQL Statement and insert the SQL object
private void TraiterSQL(string p_strquery,PosixTimeval p_timeval)
{
    float MinScore = float.MaxValue ;
    float ScoreMapp=0;
    int idxsession = -1;
    int idxhttp = -1;
    ReqSql sql = new ReqSql();
    float t_sql;
    float t_http;
    double Num;
```

```
sql.Requete = p_strquery;
sql.Literals = analyseurlexicalsql.ExtraireLitteraux(p_strquery);
//Generate the template
sql.Template = p_strquery;
for (int i = 0; i < sql.Literals.Count; i++)
{
    if (double.TryParse(sql.Literals[i].ToString(), out Num))
    {
        sql.Template = sql.Template.Replace(sql.Literals[i].ToString(), "?");
    }
    else if (sql.Literals[i].ToString().Substring(0, 1) == "'" &&
        sql.Literals[i].ToString().Substring(sql.Literals[i].ToString().Length - 1, 1) == "'")
    {
        sql.Template = sql.Template.Replace(sql.Literals[i].ToString(), "'?'");
    }
}

for (int i=0;i<ModuleHTTP.lstSession.Count ;i++)
{
  for (int j = 0; j < ModuleHTTP.lstSession[i].lstDemHttp.Count; j++)
  {
    t_sql=Convert.ToInt16(p_timeval.MicroSeconds / 1000);
    t_http = Convert.ToInt16(ModuleHTTP.lstSession[i].lstDemHttp[j].TimeStamp.Millisecond);
    if (t_sql > t_http && t_sql < t_http + t)
    {
     ScoreMapp = CalculerScoreMappage(ModuleHTTP.lstSession[i].lstDemHttp[j], sql);
     if (ScoreMapp<MinScore && ScoreMapp <=ModuleHTTP.lstSession[i].lstDemHttp[j].seuil)
       {
            MinScore = ScoreMapp;
            idxsession = i;
            idxhttp = j;
       }
    }
    else break;
  }
}
if (idxsession >= 0 || idxhttp >= 0)
{
    if (MinScore < ModuleHTTP.lstSession[idxsession].lstDemHttp[idxhttp].seuil)
    {
        if (ConfigClass.phase == Phase.Apprentissage)
        {
```

```
                    ModuleHTTP.lstSession[idxsession].lstDemHttp[idxhttp].lstReqSql.Add(sql);
                }
                else
                {
                    DetecteurAttaques.EstMalveillante(ModuleHTTP.lstSession[idxsession],
                        ModuleHTTP.lstSession[idxsession].lstDemHttp[idxhttp], sql);
                }
            }
        }
    }
    // calculate the score of mapping between a http request and a SQL Statement
    private static float CalculerScoreMappage(DemHttp p_http, ReqSql p_sql)
    {
        float Min = float.MaxValue ;
        float Score =0;
        float temp;
        for (int i = 0; i < p_http.Params.Count ;i++)
        {
            for (int j = 0; j < p_sql.Literals.Count; j++)
            {
                temp = EditDistance.Compute(p_http.Params[i].ToUpper(), p_sql.Literals[j].ToUpper());
                if (temp < Min) Min = temp;
            }
            Score = Score + Min;
            Min = float.MaxValue;
        }
        return (Score / p_http.Params.Count);
    }
  }
}
```

Listage B.3 Code source du ModuleSQL

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;
using SQLIAIDS.Parser;
using SQLIAIDS.Parser.Tokens;
namespace SQLIAIDS
{
    public class AnalyseurLexicalSQL
    {
        /// <summary>
        /// Takes a SQL statement and converts it to tokens.
        /// </summary>
        private static readonly char[] Separateurs = { '(', ')', ',', '.', ' ', '<', '>', '=', '?', ';', '+', '!','#' };
        private readonly List<Token> _tokens = new List<Token>();
        private int _index;
        private Queue<int> _savedPositions = new Queue<int>();
        /// <summary>
        /// Save current position in the string
        /// </summary>
        public void SavePos()
        {
            _savedPositions.Enqueue(_index);
        }
        /// <summary>
        /// Restore the last saved position
        /// </summary>
        public void RestorePos()
        {
            _index = _savedPositions.Dequeue();
        }
        /// <summary>
        /// End of string (file)
        /// </summary>
        public bool EOF
        {
            get { return _index >= _tokens.Count; }
        }
        /// <summary>
        /// Peek at the next token
        /// </summary>
        /// <remarks>returns null if no more tokens exist</remarks>
```

```
public Token Peek
{
    get
    {
        if (_index + 1 < _tokens.Count) return _tokens[_index + 1];
        else return null;
    }
}
/// <summary>
/// Current token
/// </summary>
public Token Current
{
    get
    {
        if (_index < _tokens.Count) return _tokens[_index];
        else return null;
    }
}
/// <summary>
/// Move to next token
/// </summary>
/// <returns>true if we could; false if there are no more tokens</returns>
public bool MoveNext()
{
    ++_index;
    if (_index < _tokens.Count) return true;
    return false;
}
public List<Token> tokens
{
    get
    {
        return _tokens;
    }
}
private static bool IgnorerColumnChars(string sql, ref int index, ref int charPos)
{
    for (; index < sql.Length; ++index)
    {
        ++charPos;
        if (!EstColumnChar(sql[index])) return true;
```

```
        }
        return false;
    }
    private static bool IgnorerEspaces(string sql, ref int index, ref int charPos, ref int lineNumber)
    {
        for (; index < sql.Length; ++index)
        {
            ++charPos;
            if (sql[index] == '\n')
            {
                ++lineNumber;
                charPos = 1;
            }
            if (!char.IsWhiteSpace(sql[index])) return true;
        }
        return false;
    }
    private static bool EstColumnChar(char ch)
    {
        return char.IsLetterOrDigit(ch) || ch == '_' || ch == '*';
    }
    private static bool EstSeparateur(char ch)
    {
        foreach (char c in Separateurs)
        {
            if (c == ch) return true;
        }
        return false;
    }
    /// <summary>
    /// Converts string to general purpose tokens.
    /// </summary>
    /// <param name="tokenString">sql statement to parse</param>
    public List<string> ExtraireLitteraux(string tokenString)
    {
        int index = 0;
        bool inSingleQuote = false;
        bool inDoubleQuote = false;
        int startPos = 0;
        int charPos = 0;
        int lineNumber = 0;
        double Num;
```

```
List<string> tmplstLiterals=new List<string>();
string strTemplate;
while (true)
{
    if (inSingleQuote || inDoubleQuote)
    {
        if (tokenString[index - 1] != '\\')
        {
            if (tokenString[index] == '\'')
            {
                _tokens.Add(new SingleQuotedToken(charPos, lineNumber,
                    tokenString.Substring(startPos, index - startPos + 1)));
                inSingleQuote = false;
            }
            else if (tokenString[index] == '\"')
            {
                _tokens.Add(new DoubleQuotedToken(charPos, lineNumber,
                    tokenString.Substring(startPos, index - startPos + 1)));
                inDoubleQuote = false;
            }
        }
        ++charPos;
        ++index;
        continue;
    }
    if (!IgnorerEspaces(tokenString, ref index, ref charPos, ref lineNumber))
        break;
    if (EstColumnChar(tokenString[index]))
    {
        startPos = index;
        int end;
        if (!IgnorerColumnChars(tokenString, ref index, ref charPos))
            end = tokenString.Length;
        else
            end = index;
        string word = tokenString.Substring(startPos, end - startPos);
        _tokens.Add(new StringToken(charPos, lineNumber, word));
        --index; // needed since we've moved to next token with IgnoreColumnChars
    }
    else if (EstSeparateur(tokenString[index]))
    {
        char ch = tokenString[index];
```

```
                _tokens.Add(new SeparatorToken(charPos, lineNumber, ch));
            }
            else if (tokenString[index] == '\'')
            {
                startPos = index;
                inSingleQuote = true;
            }
            else if (tokenString[index] == '"')
            {
                startPos = index;
                inDoubleQuote = true;
            }
            else if (tokenString[index] == '-')
            {
                break;
            }
            else
                throw new InvalidDataException("Unexpected char: " + tokenString[index]);
            ++charPos;
            ++index;
        }
        //Generate the literals
        for (int i = 0; i < tokens.Count; i++)
        {
            if (double.TryParse(tokens[i].ToString(), out Num))
            {
                tmplstLiterals.Add(tokens[i].ToString());
            }
            else if (tokens[i].ToString().Substring(0, 1) == "'" &&
                tokens[i].ToString().Substring(tokens[i].ToString().Length - 1, 1) == "'")
            {
                tmplstLiterals.Add(tokens[i].ToString());
            }
        }
        _tokens.Clear();
        return tmplstLiterals;
    }
  }
}
```

Listage B.4 Code source du AnalyseurLexicalSQL

```
using System;
using System.Collections.Generic;
using System.Data.SqlClient;
using System.Data;
namespace SQLIAIDS
{
  public class GenerateurProfiles
  {
      /// <summary>
      /// Takes a expired session and traits it.
      /// </summary>
    public bool TraiterSession(string p_SessionID)
    {
      string TemplateSet = "<params> ";
      for (int i = 0; i < ModuleHTTP.lstsession.Count; i++)
      {
        if (ModuleHTTP.lstsession[i].SessionID == p_SessionID)
        {
            for (int j = 0; j < ModuleHTTP.lstsession[i].lstDemHttp.Count; j++)
          {
              for (int k = 0; k < ModuleHTTP.lstsession[i].lstDemHttp[j].lstReqSql.Count; k++)
              {
                TemplateSet += "<p>" + ModuleHTTP.lstsession[i].lstDemHttp[j].lstReqSql[k].Template
                + "</p>";
              }
              TemplateSet = TemplateSet + "<params>";
              InsererlstTemplate(ModuleHTTP.lstsession[i].lstDemHttp[j].URL,
              ModuleHTTP.lstsession[i].lstDemHttp[j].Methode.ToString(), TemplateSet);
          }
          ModuleHTTP.lstsession.RemoveAt(i);
          return true;
        }
      }
      return false;
    }

    // Insert a list of template into DB (if it isn't alread existed )
    public bool InsererlstTemplate(string p_URL,string p_httpmethod, string p_TemplateSet)
    {
            ConnectionClass.SQLIDSconn.Open();
            SqlCommand sqlcomm = new SqlCommand();
            sqlcomm.Connection = ConnectionClass.SQLIDSconn;
```

```
            sqlcomm.CommandType = CommandType.StoredProcedure;
            sqlcomm.CommandText = "InsererlstTemplate";
            sqlcomm.Parameters.Add("@URL",SqlDbType.NVarChar,100 );
            sqlcomm.Parameters.Add("@Methode", SqlDbType.NVarChar,100);
            sqlcomm.Parameters.Add("@Templates", SqlDbType.Xml );
            sqlcomm.Parameters[0].Value = p_URL;
            sqlcomm.Parameters[1].Value = p_httpmethod;
            sqlcomm.Parameters[2].Value = p_TemplateSet;
            sqlcomm.ExecuteNonQuery();
            ConnectionClass.SQLIDSconn.Close();
            return true;
        }
    }
}
```

Listage B.5 Code source du GenerateurProfiles

```
CREATE procedure [dbo].[InsererlstTemplate]
 @URL nvarchar(100),
 @HttpMethod nvarchar(100),
 @Templates xml -- This will recevie a List of values
as
begin
 declare @IDLstTemplate int
 declare @IDURL int
 declare @Order int
 declare @Template nvarchar(max)
 CREATE TABLE #TempTable(Template nvarchar(max),[order] int)
 DECLARE db_cursor CURSOR FOR
 SELECT params.p.value('.','nvarchar(max)') FROM @Templates.nodes('/params/p') as params(p);
 OPEN db_cursor
 FETCH NEXT FROM db_cursor INTO @Template
 set @Order=1
 WHILE @@FETCH_STATUS = 0
 BEGIN
   INSERT INTO #TempTable (Template,[order]) VALUES(@Template,@Order)
   set @Order=@Order+1
   FETCH NEXT FROM db_cursor INTO @Template
 END
 CLOSE db_cursor
 DEALLOCATE db_cursor
 if not exists (select * from dbo.TbllstTemplates
  inner join dbo.TblTemplates on (dbo.TbllstTemplates.IDLstTemplate=dbo.TblTemplates.IDLstTemplate)
  inner join dbo.TblUrls on (dbo.TbllstTemplates.IDUrl=dbo.TblUrls.IDUrl)
  inner join #TempTable on (dbo.TblTemplates.Template=#TempTable.Template)
  where URL=@URL and HttpMethod=@HttpMethod)
 begin
        select @IDURL=IDUrl from dbo.TblUrls where URL=@URL
        if (@IDURL is null) insert into dbo.TblUrls values(@URL)
        select @IDURL=@@IDENTITY
        insert into dbo.TbllstTemplates values(@IDURL,@HttpMethod)
        select @IDLstTemplate=@@IDENTITY
        insert into TblTemplates select @IDLstTemplate,Template,[order],GETDATE() from #TempTable
 end
 DROP TABLE #TempTable
end
```

Listage B.6 Procédure stockée : InsererlstTemplate

```
using System;
using System.Collections.Generic;
using System.Data.SqlClient;
using System.Data;
namespace SQLIAIDS
{
    public static class DetecteurAttaques
  {
        public static bool EstMalveillante(Session p_session, DemHttp p_http, ReqSql p_sql)
        {
            if (p_http.type == Type.Normale) return false;
            else if (p_http.type == Type.Contaminee)
            {
                if (VerifierTemplate(p_http.URL, p_sql.Template, p_sql.Ordre))
                {
                    InsererAttaque(p_http.URL, p_sql.Requete, p_session.AdresseIP);
                    return true;
                }
            }
            return true;
        }
        public static bool VerifierTemplate(string p_URL, string p_template, int p_order)
        {
            ConnectionClass.SQLIDSconn.Open();
            SqlCommand sqlcomm = new SqlCommand();
            sqlcomm.Connection = ConnectionClass.SQLIDSconn;
            sqlcomm.CommandType = CommandType.StoredProcedure;
            sqlcomm.CommandText = "VerifierTemplate";
            sqlcomm.Parameters.Add("@URL", SqlDbType.NVarChar, 100);
            sqlcomm.Parameters.Add("@Template", SqlDbType.NVarChar, -1);
            sqlcomm.Parameters.Add("@Ordre", SqlDbType.Int);
            sqlcomm.Parameters[0].Value = p_URL;
            sqlcomm.Parameters[1].Value = p_template;
            sqlcomm.Parameters[2].Value = p_order;
            Int16 TemplateCount=Convert.ToInt16(sqlcomm.ExecuteScalar());
            ConnectionClass.SQLIDSconn.Close();
            if (TemplateCount == 0) return true;
            else return false;
        }

        public static bool InsererAttaque(string p_URL, string p_requete, string p_IPSource)
        {
```

```
            ConnectionClass.SQLIDSconn.Open();
            SqlCommand sqlcomm = new SqlCommand();
            sqlcomm.Connection = ConnectionClass.SQLIDSconn;
            sqlcomm.CommandType = CommandType.StoredProcedure;
            sqlcomm.CommandText = "InsererAttaque";
            sqlcomm.Parameters.Add("@URL", SqlDbType.NVarChar, 100);
            sqlcomm.Parameters.Add("@Requete", SqlDbType.NVarChar, -1);
            sqlcomm.Parameters.Add("@IPSource", SqlDbType.NVarChar,50 );
            sqlcomm.Parameters[0].Value = p_URL;
            sqlcomm.Parameters[1].Value = p_requete;
            sqlcomm.Parameters[2].Value = p_IPSource;
            sqlcomm.ExecuteNonQuery();
            ConnectionClass.SQLIDSconn.Close();
            return true;
        }
    }
}
```

Listage B.7 Code source du DetecteurAttaques

```
CREATE procedure [dbo].[VerifierTemplate]
    @URL nvarchar(100),
    @Template nvarchar(max) ,
    @Ordre int
as
begin
select COUNT(*) from dbo.TblTemplates
inner join dbo.TbllstTemplates on (dbo.TblTemplates.IDLstTemplate=dbo.TbllstTemplates.IDLstTemplate)
inner join dbo.TblUrls on (dbo.TbllstTemplates.IDUrl=dbo.TblUrls.IDUrl)
where dbo.TblUrls.URL = @URL and dbo.TblTemplates.Template=@Template
and dbo.TblTemplates.[Order]=@Ordre
end
```

Listage B.8 Procédure stockée : VerifierTemplate

```
CREATE procedure [dbo].[InsererAttaque]
    @URL nvarchar(100),
    @Requete nvarchar(max),
    @IPSource nvarchar(50)
as
begin
        insert into dbo.TblAttacks values(@URL,@Requete,@IPSource,GETDATE())
end
```

Listage B.9 Procédure stockée : InsererAttaque

APPENDICE C

STRUCTURES DES BASES DE DONNÉES

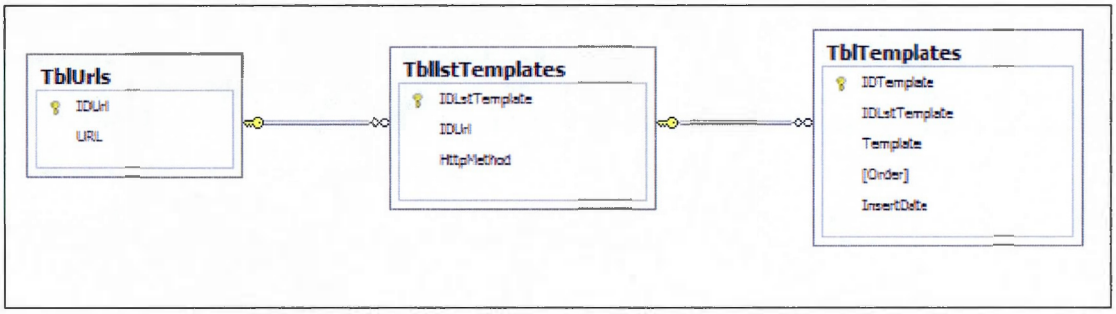

Figure C.1 STRUCTURE DE LA BASE DE MODÈLES SQL

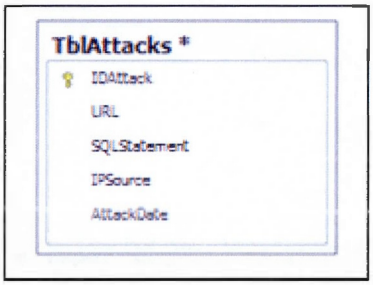

Figure C.2 STRUCTURE DE LA BASE D'ATTAQUES

BIBLIOGRAPHIE

[AP02] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*, pages 25–27. Cambridge University Press, second edition, 2002.

[BÉ10] Guy BÉGIN. Inf8750 – sécurité des systèmes informatiques. Notes de cours, Hiver 2010.

[BBMm07] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, and V.N. Venkatakrishnan. Candid : Preventing sql injection attacks using Dynamic candidate evaluations. *In Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, (October) :pages 12–24, 2007.

[BK04] S. Boyd and A. Keromytis. SQLrand : Preventing SQL Injection Attacks. *In Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, pages 292–302, June 2004.

[BWS05] Gregory T. Buehrer, Bruce W. Weide, and Paolo a. G. Sivilotti. SQLGuard :Using parse tree validation to prevent SQL injection attacks. *Proceedings of the 5th international workshop on Software engineering and middleware - SEM 205*, (September) :pages 106–113, 2005.

[CD12] Asma CHIKH and Amina DJENNANE. Sécurité d'une application Web à l'aide d'un système de détection d'intrusions comportementale. Mémoire de fin d'études, université Abou Bakr Belkaid– Tlemcen, Juillet 2012.

[Cer10] C. Cerrudo. Manipulating Microsoft SQL Server Using SQL Injection. Technical report, Application Security, INC., 2010.

[Deb05] Hervé Debar. An introduction to intrusion-detection systems. *IBM Research, Zurich Research Laboratory,Saumerstrasse 4, CH 8803 Ruschlikon*, pages 1–18, 2005.

[dot11] dotnetperls. Levenshtein. http ://www.dotnetperls.com/levenshtein, consulté le 25 Juillet 2011.

[GD04] C. Gould and P. Devanbu. JDBC checker : a static analysis tool for SQL/JDBC applications. *Proceedings. 26th International Conference on Software Engineering*, pages 697–698, 2004.

[Gro11] Network Working Group. Hypertext transfer protocol – http/1.1. http ://www.w3.org/Protocols/rfc2616/rfc2616.html, consulté le 15 Juillet 2011.

[GT07] Tyrone Grandison and Evimaria Terzi. Intrusion detection technology. *IBM Almaden Research Center*, (September) :pages 1–7, 2007.

[GVB12] Fulvio Risso Gianluca Varenni, Loris Degioanni and John Bruno. The industry-standard windows packet capture library. http ://www.winpcap.org/, consulté le 23 Juillet 2012.

[Ham12] Richard Hamming. Distance de hamming. http ://fr.wikipedia.org/wiki/Distance_de_Hamming, consulté le 23 Juin 2012.

[HO06] William G. J. Halfond and Alessandro Orso. Preventing SQL injection attacks using AMNESIA. *Proceeding of the 28th international conference on Software engineering - ICSE '06*, pages 1–4, 2006.

[Hol04] Ted Holland. Using IPS and IDS together for Defense in Depth. Technical report, 2004.

[HVO06] William G. J. Halfond, Jeremy Viegas, and Alessandro Orso. A classification of sql injection attacks and countermeasures. *Georgia Institute of Technology*, pages 1–11, 2006.

[JCmA12] Jeremiah Grossman Jason Coleman, Ofer Shezafi and Robert Auger. The web application security consortium. http ://projects.webappsec.org/page/13246995/ WebHackingIncidentDatabaseRealTimeStatistics, consulté le 2 Septembre 2012.

[JFM08] M. Vieira J. Fonseca and H. Madeira. Testing and comparing web vulnerability scanning tools for sql injection and xss attacks. *In Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*, (September) :pages 365–372, 2008.

[kod12] koders. Parserstatement. http ://www.koders.com/csharp/ fid4AE87D591015A6ECB85595CADEEBE40CD7D01000.aspx, consulté le 12 Septembre 2012.

[Lar11] Francois Larouche. http ://www.sqlpowerinjector.com/, June 2011.

[LD12] Vladimir Levenshtein and Frederick Damerau. Distance de damerau-levenshtein. http ://fr.wikipedia.org/wiki/Distance_de_Levenshtein, consulté le 23 Juin 2012.

[Mic11a] Microsoft. How to : Protect from injection attacks in asp.net, June 2011.

[Mic11b] Microsoft. Sql injection attack. http ://blogs.technet.com/sql-injection-attack.aspx, consulté le 2 octobre 2011.

[Nga09] Serge Rufin Ngansop. Étude et optimisation d'un nouveau mécanisme de sécurité pour protéger les applications Web contre les attaques par injection de code SQL. Rapport de Bibliographie, ENST Bretagne – INSA, 2009.

[OWA10] OWASP. OWASP : Top 10 - 2010. Technical report, 2010.

[OWA12] OWASP. Blind sql injection. https ://www.owasp.org/index.php/Blind_SQL_Injection, consulté le 15 Juin 2012.

[Saf] Gunadiz Safia. Algorithmes d'intelligence artificielle pour la classification d'attaques réseaux à partir de données tcp. Mémoire de magistère, Université Mohamed BOUGARA de Boumerdes, 2011.

[Sof12] Gudu Software. Professional sql engine for various databses. http ://www.sqlparser.com/, consulté le 23 Juillet 2012.

[SS] D. Scott and R. Sharp. Abstracting application-level web security. *In Proceedings of the 11th International Conference on the World Wide Web*, (January) :pages 396–407.

[SW] Zhendong Su. and Gary Wassermann. SQLCheck :The Essence of Command Injection Attacks in Web Applications. *Proceedings of the 5th international workshop on Software engineering and middleware - SEM 205*, (January).

[SWLL06] San-tsai Sun, Ting Han Wei, Stephen Liu, and Sheung Lau. Classification of sql injection attacks. Technical report, 2006.

[Wik11a] Wikipedia. Web application. http ://en.wikipedia.org/wiki/Web_application, consulté le 2 Juillet 2011.

[Wik11b] Wikipedia. Procédure stockée. http ://fr.wikipedia.org/wiki/Proc%C3%A9dure_stock%C3%A9e, consulté le 25 Juillet 2011.

[Wik12a] Wikipedia. Intrusion detection system. http ://en.wikipedia.org/wiki/Intrusion_detection_system, consulté le 20 Juillet 2012.

[Wik12b] Wikipedia. Sécurité des systèmes d'information. http ://fr.wikipedia.org/wiki/S%C3%A9curit%C3%A9_des_syst%C3%A8mes_d'information, consulté le 20 Octobre 2012.

[Wik12c] Wikipedia. Plus longue sous-séquence commune. http ://fr.wikipedia.org/wiki/Plus_longue_sous-s%C3%A9quence_commune, consulté le 23 Juin 2012.

[Wik13] Wikipedia. Système de détection d'intrusion. http ://fr.wikipedia.org/wiki/Syst%C3%A8me_de_d%C3%A9tection_d%27intrusion, consulté le 20 Octobre 2013.