Titre: Title:	Services de sécurité inter-locataire et multi-locataire pour les logiciels en tant que service
Auteur: Author:	Mohamed Yassin
Date:	2019
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Yassin, M. (2019). Services de sécurité inter-locataire et multi-locataire pour les logiciels en tant que service [Thèse de doctorat, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/3986/

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Services de sécurité inter-locataire et multi-locataire pour les logiciels en tant que service

MOHAMED YASSIN
Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie informatique

Août 2019

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

Services de sécurité inter-locataire et multi-locataire pour les logiciels en tant que service

présentée par **Mohamed YASSIN**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

Gabriela NICOLESCU, présidente
Hanifa BOUCHENEB, membre et directrice de recherche
Chamseddine TALHI, membre et codirecteur de recherche
Foutse KHOMH, membre
Abdelouaheb GHERBI, membre

REMERCIEMENTS

Je voudrais exprimer mes sentiments les plus sincères envers tous ceux qui, de près ou de loin, ont contribué à cette thèse.

Tout dabord, je tiens à remercier ma directrice de recherche Madame Hanifa Boucheneb, professeure à l'École Polytechnique Montréal, ainsi que mon codirecteur de recherche Monsieur Chamseddine Talhi, professeur à l'École de technologie supérieure pour leur aide scientifique et leur support financier. Leurs conseils ont été indispensables pour la réussite de ce projet de recherche.

J'adresse tous mes remerciements à Madame Gabriela Nicolescu, professeure à l'École Polytechnique Montréal, ainsi que Monsieur Abdelouaheb Gherbi, professeur à l'École de technologie supérieure, pour avoir accepté de participer à mon jury de thèse. Je tiens particulièrement à adresser mes remerciements à Monsieur Foutse Khomh, professeur à l'École Polytechnique Montréal, d'avoir participé à l'évaluation de mon examen de synthèse et à mon jury de thèse.

Merci à mes parents et à ma famille, et surtout, mon épouse Zeinab Wazneh pour son amour, son encouragement et ses sacrifices.

RÉSUMÉ

Récemment, l'infonuagique a joué un rôle essentiel dans l'évolution de la technologie d'informatique. Les logiciels en tant que service (SaaS for software as a service) sont parmi les services infonuagiques les plus attractifs qui ont suscité l'intérêt des fournisseurs et des consommateurs d'applications Web. D'une part, l'externalisation des ressources permet au fournisseur de déployer une application dans une infonuagique publique au lieu de gérer ses ressources sous-jacentes (machines physiques). D'autant plus, les ressources de cette application peuvent être dynamiquement et automatiquement mises à l'échelle en fonction de l'évolution de la clientèle et/ou de la quantité du trafic. D'autre part, la mutualisation (partage) des ressources permet au fournisseur une réduction significative des coûts d'infrastructure et de maintenance en partageant la même instance d'application entre plusieurs locataires, appelés tenants (en anglais). Un locataire peut s'abonner aux SaaS à la demande en payant à l'usage.

En dépit de leurs avantages, l'externalisation et la mutualisation des ressources entraînent de nouveaux défis et risques de sécurité qui doivent être inventoriés et résolus par le fournisseur d'un SaaS. Le locataire d'un SaaS ne peut pas déployer ses systèmes de détection d'intrusion (IDS for intrusion detection system) préférés puisqu'il ne contrôle ni le code source ni l'infrastructure de l'application (déployée par le fournisseur dans une infonuagique publique). Le fournisseur doit donc non seulement intégrer des IDS en tant que service dans son infrastructure infonuagique, mais aussi protéger chaque locataire selon ses propres exigences de sécurité. Dans un SaaS multi-locataire, les données des locataires, qui peuvent être des compétiteurs, sont stockées dans la même base de données. Le fournisseur doit donc détecter et prévenir les attaques réalisées par un locataire contre les données d'autres locataires.

Plusieurs recherches scientifiques proposent des IDS infonuagiques qui se focalisent sur l'infrastructure (réseaux virtuels, machines virtuelles, etc.). Cependant, ces IDS n'offrent pas une sécurité en tant que service au fournisseur et aux locataires d'un SaaS. D'autres recherches scientifiques et entreprises informatiques suggèrent des mécanismes d'isolation des données des locataires afin de réduire les risques d'attaques entre eux. Cependant, ces mécanismes ne sont pas automatisés et ne permettent pas de prévenir les attaques entre les locataires partageant la même base de données.

Cette thèse s'intéresse à l'intégration et la mutualisation d'IDS en tant que service dans une infonuagique publique ainsi qu'à la détection et à la prévention des attaques entre les locataires d'un SaaS multi-locataire. Elle s'articule autour de trois parties.

La première partie propose et évalue un cadre infonuagique, appelé SQLIIDaaS (for SQL Injection Intrusion Detection framework as a Service for SaaS providers), qui permet au fournisseur SaaS d'intégrer une contremesure d'injection SQL dans une infonuagique publique. Cette intégration est réalisée avec un niveau élevé de portabilité grâce à une corrélation entre les requêtes HTTP et SQL en fonction de leur temps d'occurrence et leur similarité. SQLIIDaaS est offert comme un service de sécurité en développant ses modules sous forme de machines virtuelles. SQLIIDaaS a été testé dans AWS (for Amazon Web Services) en intégrant un IDS qui détecte les attaques par injection SQL (SQLIA for SQL injection attack) visant une application Web. Les résultats d'expérimentation montrent que cette intégration a augmenté de 3.4% le temps moyen de réponse HTTP et de 5.61% l'utilisation moyenne des processeurs virtuels. Les taux de détection et de précision de l'IDS intégré sont 0.9735 et 1, respectivement.

La deuxième partie propose et évalue un cadre infonuagique, appelé ITADP (for an Inter-Tenant Attack Detection and Prevention framework for multi-tenant SaaS), afin de prévenir les attaques entre les locataires d'un SaaS qui partagent certaines tables d'une base de données relationnelle. Ce cadre analyse l'arbre syntaxique des requêtes SQL pour vérifier si une requête SQL permet à un locataire d'accéder aux données des autres locataires. L'optimisation des algorithmes proposés permet de réduire le temps d'exécution de cette vérification. ITADP a été testé dans AWS en fonction de deux SaaS multi-locataires. Les résultats d'expérimentation montrent que ITADP prévient les attaques entre les locataires d'un SaaS avec un taux moyen de détection de 0.981 et un taux moyen de précision de 0.967.

La troisième partie propose et évalue un cadre infonuagique, appelé MTIDaaS (for Multi-Tenant Intrusion Detection framework as a Service for SaaS), qui accorde au fournisseur l'opportunité d'intégrer plusieurs IDS applicatives en tant que service de sécurité dans une infonuagique publique. Afin de réduire son coût tout en respectant les exigences de sécurité de chaque locataire, nous avons appliqué la mutualisation des ressources sur ce cadre. MTIDaaS a été testé dans AWS en fonction d'un SaaS vulnérable et multi-locataire en intégrant deux IDS applicatives qui détectent les SQLIAs et scripts inter-site (XSS for Cross-site scripting). Les résultats d'expérimentation montrent les caractéristiques suivantes de MTIDaaS : (i) sa portabilité élevée au niveau des langages de programmation et des bases de données ; (ii) sa faible surcharge (5.1% de processeur virtuels) sur les machines virtuelles hébergeant les services Web ; (iii) ses taux de détection et de précision sont 0.964 et 0.982, respectivement ; (iv) son accessibilité en libre-service et à la demande par le fournisseur et chaque locataire ; (v) sa mise à l'échelle automatique en fonction de la quantité des requêtes HTTP ; (vi) son coût d'infrastructure optimisé grâce à la mise à l'échelle et à la mutualisation de ses ressources entre les locataires ; (vii) son paiement à l'usage pour le fournisseur et chaque

locataire grâce à deux modèles mathématiques de coût proposés ; (viii) sa haute disponibilité grâce à la distribution du trafic sur plusieurs machines virtuelles créées dans deux zones de disponibilité.

ABSTRACT

Recently, cloud computing plays a vital role in the evolution of computer technology. Software-as-a-Service (SaaS) is one of the cloud services that has attracted the providers and clients (tenants) of Web applications. On the one hand, outsourcing allows a SaaS provider to deploy an application in a public cloud instead of managing its underlying resources (physical machines). The resources of this application can be scaled dynamically and automatically according to the evolution of the customer and/or the amount of traffic. On the other hand, multi-tenancy (or resources pooling) enables a SaaS provider to significantly reduce the infrastructure and maintenance costs by sharing the same application and database instances among several tenants. A tenant can subscribe to SaaS on-demand and pay according to pay-per-use model.

However, the outsourcing and multi-tenancy bring new challenges and security risks that must be addressed by the SaaS provider. A tenant can not deploy its preferred intrusion detection systems (IDS) since it does not control the source code and the infrastructure of the application (deployed by the provider in a public cloud). Therefore, the provider must not only integrate IDS as a service into its cloud infrastructure, but also protect each tenant according to its own security requirements. In a multi-tenant SaaS, the data of tenants that can be competitors are stored in the same database. Therefore, the provider must detect and prevent attacks realized by a tenant (maliciously or accidentally) against the data of other tenants.

The cloud-based IDS proposed by scientific research focus on the infrastructure (e.g., virtual networks, virtual machines, etc.). However, they do not detect attacks between the tenants of SaaS and do not provide security as a service for both SaaS provider and tenant. Other scientific research and IT companies propose tenant data isolation mechanisms to reduce the risk of inter-tenant attacks. However, these mechanisms are not automated and do not prevent attacks between tenants sharing the same database.

This thesis focuses on the integration of multi-tenant IDS as a service in a public cloud and the prevention of the attacks among the tenants of a multi-tenant SaaS. Our research work is divided into three parts.

The first part proposes and evaluates a SQL injection intrusion detection framework as a service for SaaS providers (SQLIIDaaS) that allows the SaaS provider to integrate SQL injection countermeasures into a public cloud. This integration is realized with a high level of portability thanks to a correlation between the HTTP and SQL requests according to

their time of occurrence and their similarity. SQLIIDaaS is offered as security as a service by developing its modules as virtual machines. It is evaluated in Amazon web services (AWS) based on SQL injection countermeasure and vulnerable Web application. The experiment results show that this integration increases by 3.4 % the average HTTP response time and 5.61 % the average use of virtual processors. The detection and accuracy rates of the integrated IDS are 0.9735 and 1, respectively.

The second part proposes and evaluates an inter-tenant attack detection and prevention framework for multi-tenant SaaS (ITADP) that prevents attacks among the tenants of SaaS that share certain tables of a relational database. This framework analyzes the syntax tree of a given SQL query to verify if it allows a tenant to access the data of the other tenants at runtime. The execution time of this verification is reduced by optimizing the proposed algorithms. ITADP is evaluated in AWS based on two multi-tenant SaaS. The experimental results show that ITADP prevents attacks among SaaS tenants with a average detection rate of 0.981 and an average accuracy rate of 0.967.

The third part proposes and evaluates a multi-tenant intrusion detection framework as a service for SaaS (MTIDaaS) that allows the SaaS provider to integrate several application IDS as security as a service into a public cloud. We have applied the multi-tenancy concept on this framework to reduce its cost while meeting the security requirements of each tenant. MTIDaaS is evaluated in AWS according to a vulnerable multi-tenant SaaS by integrating two intrusion detection techniques that detect SQL injection and cross-site scripting attacks. The experimental results show the following characteristics of MTIDaaS: (i) its high portability on programming languages and databases; (ii) its low overhead (5.1 % of CPU) on virtual machines that host the web services; (iii) its detection and accuracy rates are 0.964 and 0.982, respectively; (iv) its on-demand self-service for both provider and tenant of SaaS; (v) its auto-scalability according to the amount of HTTP requests; (vi) its optimized infrastructure cost thanks to the auto-scalability and the resource sharing among the tenants; (vii) its pay-per-use for both provider and tenant of SaaS thanks to our two mathematical models of cost; (viii) its high availability which is ensured by distributing traffic across multiple virtual machines created in two availability zones.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

VM	Machine virtuelle (Virtual Machine)
VMM	Gestionnaire des machines virtuelles (Virtual Machine Manager)
IaaS	Infrastructure en tant que service (Infrastructure as a Service)
PaaS	Plateforme en tant que service (Platform as a Service)
SaaS	Logiciel en tant que service (Software as a Service)
DBMS	Système de gestion de base de données (Database Management System)
SQLIA	Attaque par injection SQL (SQL Injection Attack)
XSS	Script inter-site (Cross-site Scripting)
DoS	Attaque par déni de service (Denial of Service attack)
DDoS	Attaque par déni de service distribuée (Distributed Denial of Service attack)
XDoS	Attaque par déni de service XML (XML Denial of Service attack)
IDS	Système de détection d'intrusion (Intrusion Detection System)
HIDS	Système de détection d'intrusion hôte (Host IDS)
NIDS	Système de détection d'intrusion réseau (Network IDS)
TP	Vrais positifs (True Positives)
FP	Faux positifs (False Positives)
FN	Faux négatifs (False Negatives)
DR	Taux de détection (Detection Rate)
PR	Taux de précision (Precision Rate)
Snort	Free open source network intrusion detection system
SLA	Accord de niveau de service (Service Level Agreement)
AWS	Services Web Amazon (Amazon Web Services)
OS	Système d'exploitation (Operating system)

HTTP	Protocole de transfert hypertexte (Hypertext Transfer Protocol)
HTTPS	Protocole de transfert hypertexte sécurisé (HyperText Transfer Protocol Secure)
SOAP	Simple Object Access Protocol
REST	Representational State Transfer
WSDL	Web Services Description Language
UDDI	Universal Description Discovery and Integration
IIS	Microsoft Internet Information Services
RDS	Service de base de données relationnelle (Relational Database Service)
VPC	Infonuagique virtuellement privée (Virtual Private Cloud)
NAT	Traduction d'adresses réseau (Network Address Translation)
TLS	Sécurité de la couche de transport (Transport Layer Security)
SecaaS	Sécurité en tant que service (Security as a Service)

CHAPITRE 1 INTRODUCTION

Ce chapitre a pour objectif de présenter les concepts de base de l'infonuagique, les logiciels en tant que service et la détection d'intrusions. Il présente ensuite notre problématique et nos objectifs de recherche.

De nos jours, les services infonuagiques sont largement adoptés par les fournisseurs et les consommateurs grâce à leur accessibilité en libre-service, l'élasticité et la mutualisation de leurs ressources, le paiement à l'usage et leur haute disponibilité [1,2]. L'architecture infonuagique est composée de trois principaux niveaux ou modèles de services : une infrastructure en tant que service (IaaS for Infrastructure as a Service), une plateforme en tant que service (PaaS for Platform as a Service) et un logiciel en tant que service (SaaS for Software as a Service). L'IaaS est le niveau de service le plus bas qui consiste à offrir des ressources d'infrastructure via Internet telles que : réseaux et machines virtuels (VMs for Virtual Machines). Le PaaS est le niveau intermédiaire qui représente un système d'exploitation (OS for Operating System), un serveur Web, un système de gestion de base de données (DBMS for Database Management System) ou un environnement de développement et de déploiement des applications infonuagiques. Un SaaS est une application web orientée service et déployée dans une IaaS. En 2018, les deux tiers des dépenses infonuagiques, qui représentent 160 milliards de dollars, sont affectés aux SaaS [3].

Les SaaS sont de plus en plus populaires grâce à plusieurs bénéfices que l'externalisation et la mutualisation des ressources offrent aux fournisseurs et aux locataires (tenants). Pour un fournisseur, l'externalisation des ressources consiste à déployer l'application dans une infonuagique publique, ce qui permet une allocation automatique et progressive des ressources infonuagiques, selon la croissance de la clientèle ; la mutualisation des ressources consiste à partager la même instance d'application et/ou de bases de données entre plusieurs locataires, ce qui permet de réduire le coût d'infrastructure et de maintenance. Pour un locataire, l'externalisation consiste à s'abonner à l'application (offerte par le fournisseur) en libre-service en payant à l'usage au lieu d'implémenter et de déployer sa propre application (à l'interne).

En termes de sécurité, contrairement aux IaaS et PaaS qui sont gérés par des développeurs et administrateurs, les SaaS sont accessibles aux utilisateurs finaux (via des pages Web) qui peuvent être malintentionnés. Par conséquent, le modèle SaaS est le plus vulnérable (suivi par PaaS) de l'architecture infonuagique avec 39% des attaques comme les injections SQL (SQLIA for SQL Injection Attack), les scripts inter-site (XSS for Cross-site scripting) et les dénis de service XML (XDoS for XML Denial of Service) [4]. À cet égard, l'objectif de notre thèse est

d'améliorer la sécurité des SaaS qui communiquent avec des bases de données (relationnelles). On s'intéresse plus précisément aux problèmes, aux risques et aux défis de sécurité liés à la mutualisation et à l'externalisation des ressources telles que : la détection et la prévention des attaques entre les locataires, l'intégration des systèmes de détection d'intrusion (IDS for Intrusion Detection System) applicatives dans une infonuagique publique et la mutualisation d'un IDS entre plusieurs locataires d'un SaaS.

1.1 Définitions et concepts de base

Dans cette section, nous présentons quelques concepts relatifs à l'infonuagique, aux logiciels en tant que service et à la détection d'intrusions.

1.1.1 Infonuagique (Cloud Computing)

Selon le NIST (National Institute of Standards and Technology), l'infonuagique (Cloud Computing) est un ensemble de ressources informatiques virtuelles offertes comme des services par un fournisseur aux clients via Internet [5]. Un client peut facilement s'abonner et utiliser ces services sans se soucier de leur maintenance, leur gestion et leur implémentation. L'infonuagique se caractérise par son libre-service, son ouverture, la mutualisation et l'élasticité de ses ressources et son paiement à l'usage.

- Un libre-service permet au client d'accéder aux services à la demande sans une demande écrite au fournisseur.
- Une ouverture est l'accessibilité aux services par Internet à tout moment, par exemple, avec un ordinateur, un téléphone intelligent ou une tablette.
- Une mutualisation des ressources permet au fournisseur de partager des ressources hétérogènes (hôtes physiques, applications, bases de données, etc.) entre plusieurs clients afin de réduire les différents coûts (infrastructure, maintenance, gestion et implémentation).
- Une élasticité des ressources est la capacité des services à s'adapter rapidement et automatiquement aux demandes massives, dynamiques et fréquentes des clients en termes de ressources.
- Un paiement à l'usage permet à chaque client de payer en fonction de son utilisation (nombre des machines virtuelles, nombre de comptes d'utilisateur, etc.), ce qui exige une mesure systématique des ressources consommées.

Un service infonuagique peut être déployé dans une infonuagique publique, privée, hybride ou communautaire [5–7].

— Une infonuagique publique correspond à des services/infrastructures fournis au grand public via un réseau public (Internet). Par conséquent, elle n'est pas suffisamment sécurisée.
— Une infonuagique privée correspond à des services/infrastructures gérés par une entreprise (ou tierce partie) et accessibles en passant par le réseau privé de celle-ci. Elle est donc considérée comme la plus sécurisée.
— Une infonuagique hybride est constituée d'entités infonuagiques (privée ou publique) indépendantes, mais communicantes. Elle est considérée comme sécurisée, mais aussi flexible.
— Une infonuagique communautaire est partagée par plusieurs entreprises ou organisations qui ont des intérêts communs. Une tierce partie (ou les entreprises) peut gérer ce modèle de déploiement.

L'architecture infonuagique est composée de trois couches (ou modèles) de services : IaaS, PaaS et SaaS [5, 6].

— Un fournisseur IaaS offre à ses clients des réseaux, des machines et d'autres ressources virtuelles via internet. Ensuite, chaque client peut louer, gérer et utiliser ces services, par exemple, pour déployer ses applications en libre-service [8]. Amazon Web services (AWS) est le fournisseur IaaS le plus populaire [9, 10] ;
— Un fournisseur PaaS offre aux développeurs/administrateurs des plateformes (serveurs Web, DBMS, etc.) pour implémenter et déployer des logiciels/applications. Par exemple, Microsoft offre SQL Azure comme un service de gestion de bases de données dans l'infonuagique ;
— Un fournisseur SaaS offre à ses clients (ou locataires) une application Web orientée comme un service accessible via un navigateur Web. Ensuite, chaque locataire s'abonne à cette application en payant selon les modules utilisés et/ou le nombre de comptes utilisateur [9].

1.1.2 Logiciel en tant que service (SaaS)

L'architecture d'un SaaS comprend une application Web, des services Web (i.e. SOAP for Simple Object Access Protocol, REST for Representational State Transfer) et des bases de données [11–13]. Les applications et les services Web sont installés sur des serveurs Web (e.g., IIS for Internet Information Services, Apache, etc.) et les bases de données sont installées sur des serveurs de données (e.g., SQL Azure, RDS for Amazon Relational Database Service). Ces serveurs (Web et données) représentent la plateforme (PaaS) d'un SaaS et sont déployés sur

des VMs (IaaS). Afin d'assurer une haute disponibilité, des répartiteurs de charges peuvent être ajoutés au niveau d'application (requêtes HTTP), service (requêtes SOAP ou REST) ou base de données (requêtes SQL) pour distribuer le trafic sur des différentes VMs.

Un SaaS est multi-locataire lorsqu'il est partagé par plusieurs locataires avec un nom de domaine unique pour chacun d'eux. Cela permet au fournisseur de réduire les différents coûts (infrastructure, maintenance, etc.). Considérons, par exemple, un fournisseur qui développe et héberge une seule instance d'une application de boutique en ligne [14]. Ensuite, un locataire (ex : tenant1) loue et personnalise, par exemple, les icônes et les couleurs des interfaces de cette application partagée via son nom de domaine (ex : wwww.tenant1.SaaSApp.com).

Il existe trois modèles pour gérer les données des locataires d'un SaaS [12, 13, 15, 16] : (1) Base-par-locataire (2) Schéma-par-locataire et (3) Tables-partagées.

- Le modèle base-par-locataire consiste à dédier une base de données à chaque locataire. Il est très semblable au déploiement classique, mais ces bases sont installées sur une ou plusieurs machines virtuelles infonuagiques. Le niveau de sécurité de ce modèle est élevé, car il est difficile d'attaquer toutes les bases en passant par le domaine d'un locataire, et il est facile d'isoler la victime. Cependant, les coûts d'infrastructure, de maintenance et de gestion sont aussi élevés, car un nombre énorme de bases de données doivent être créées et gérées par les administrateurs.
- Le modèle schéma-par-locataire consiste à partager une base de données entre plusieurs locataires en créant un schéma spécifique (ensemble de tables) pour chacun d'eux. Dans ce modèle, la sécurité est assez difficile à assurer : un attaquant peut, par exemple, injecter des commandes malveillantes visant la base de données au complet et l'isolation d'un locataire n'est pas évidente. La gestion des données est assez complexe, car il va falloir gérer séparément les données de chaque locataire et un ensemble de tables doit être créé pour le nouveau locataire.
- Le modèle tables-partagées consiste à partager un ensemble de tables (schéma) d'une base de données entre plusieurs locataires. Les enregistrements des locataires sont distinguées par une colonne (Ex : TenantID). La majorité des fournisseurs SaaS, y compris SalesForce, préfèrent ce modèle qui permet de réduire au maximum le coût de maintenance et de gestion [12, 15, 17]. Cependant, ce modèle pose plusieurs défis en termes de sécurité comme le risque élevé des attaques internes entre les locataires. Par exemple, un locataire malintentionné peut utiliser son domaine pour compromettre les données d'autres locataires.

1.1.3 Détection d'intrusions

Les développeurs ne peuvent pas anticiper les futures attaques et ne prennent pas souvent en considération les attaques connues durant la phase de développement. Pour cela, les IDS sont apparus pour remédier à cette faiblesse. Le rôle principal d'un IDS est donc d'inspecter le trafic dirigé vers un réseau, un hôte ou une application dans le but d'envoyer des alertes suite à la détection d'attaques [18]. Un IDS peut être évalué en fonction des propriétés suivantes [18, 19] :

- Vrais positifs (TP for True Positives) est le nombre d'alertes levées qui correspondent à de vraies attaques. Un IDS est idéal s'il détecte toutes les attaques.
- Faux positifs (FP for False Positives) est le nombre d'alertes recensées qui correspondent à de fausses attaques. Un IDS est précis s'il entraîne un taux très bas de faux positifs.
- Faux négatifs (FN for False Negatives) est le nombre d'attaques non détectées. Un IDS est complet s'il entraîne un taux très bas de faux négatifs.
- Taux de détection (DR for Detection Rate) est égal au TP/(TP + FN).
- Taux de précision (PR for Precision Rate) est égal au TP/(TP+FP).

Approches de détection

Actuellement, il existe deux approches de détection : la détection par signatures (misuse detection en anglais) et la détection d'anomalies (anomaly detection en anglais) qui peuvent être parfois combinées pour construire une approche hybride [18, 20] :

- La détection par signatures consiste à bâtir une base de signatures d'attaques connue qui doit être mise à jour régulièrement par un administrateur. Les activités extraites, par exemple, à partir des fichiers log du serveur Web sont validées contre les signatures d'attaques. Toute activité qui correspond à une signature est considérée comme une attaque. En principe, cette approche ne produit pas un taux très élevé de faux positifs, car une alerte est émise si et seulement si les données extraites correspondent à une signature d'attaque. Cependant, elle exige qu'un administrateur maintienne régulièrement la base de signatures pour considérer les nouvelles attaques. Le taux de faux négatifs est par contre élevé, dans la mesure où les attaques récentes sont indétectables et un attaquant peut contourner la détection s'il maitrise bien les signatures qui sont souvent publiques.
- La détection d'anomalies se déroule en deux phases : la phase d'apprentissage qui modélise le comportement normal de l'entité à protéger (réseau, système, application, service, etc.) et la phase de protection qui envoie une alerte si le comportement observé

s'éloigne du comportement normal. Cette approche détecte de nouvelles attaques sans intervention humaine ni mise à jour de l'IDS. Le taux de faux négatifs est plus bas que l'approche par signatures, surtout si le comportement normal (modèle de référence) est complet. Cependant, elle possède une phase d'apprentissage et peut générer un taux élevé de faux positifs, car il est difficile de déterminer avec précision un comportement normal (taux d'utilisation normale du processeur/mémoire).

Source d'informations surveillées

Un IDS peut être classifié selon la source d'informations qu'il inspecte, à savoir un hôte (HIDS for Host IDS), un réseau (NIDS for Network IDS) ou une combinaison de HIDS et NIDS (HybridIDS for Hybrid IDS) [18, 20] :

- HIDS s'installe sur un hôte physique (ou une machine virtuelle) pour détecter les attaques contre les systèmes et les applications. Il est capable d'analyser les données d'un paquet qui sont en clair une fois reçues par cet hôte.
- NIDS s'installe sur des nœuds stratégiques pour surveiller les paquets dans un réseau. Il est efficace contre les attaques qui envoient un nombre incalculable de paquets pour inonder un réseau. Mais, il analyse l'entête et non pas les détails d'un paquet qui sont souvent chiffrés.
- HybridIDS améliore sa qualité de détection en combinant un NIDS qui possède une vision globale sur les paquets d'un réseau et un HIDS qui analyse les détails d'un paquet.

1.2 Éléments de la problématique

Un SaaS hérite les vulnérabilités des services/applications Web qui sont victimes des attaques applicatives. Les attaques les plus courantes qui ciblent la couche SaaS sont les XDoS, SQLIAs et XSS [21]. Les conséquences de ces attaques sont plus critiques sur le SaaS que sur le modèle traditionnel [4,21,22]. Par exemple, la confidentialité des données de tous les locataires est violée, si un attaquant compromet une base de données partagée. Cela affecte de façon significative la réputation du fournisseur, compromet sa confiance avec ses locataires et ralentit l'adoption du modèle SaaS. Malgré l'existence de plusieurs IDS/contremesures [23–33] qui détectent ces attaques, aucun IDS n'est intégré dans un environnement infonuagique en tant que service pour protéger un SaaS. Cependant, une telle intégration est une préoccupation cruciale pour le fournisseur qui doit convaincre ses locataires que leurs données sont bien protégées. Cela encourage ces locataires à s'abonner aux SaaS. Cette intégration est une tâche complexe surtout qu'elle doit se faire avec un degré élevé de portabilité tout en

respectant les principales caractéristiques d'un service de sécurité (libre-service, élasticité des ressources, mutualisation des ressources, paiement à l'usage et haute disponibilité). Cette portabilité représente la capacité d'intégrer un IDS indépendamment de son approche de détection et du trafic surveillé (HTTP, SOAP ou SQL) sans aucune modification au code source d'application/service.

Dans le contexte d'un SaaS, l'externalisation des ressources isole les locataires de la complexité de la mise en œuvre d'un SaaS et la mutualisation des ressources permet au fournisseur SaaS de réduire les différents coûts (infrastructure, opérationnels et maintenance) [12, 17]. Cependant, cette externalisation empêche un locataire d'intégrer son propre IDS, car il ne contrôle ni le code source ni l'infrastructure d'un SaaS. Un fournisseur, qui associe un IDS à des instances partagées d'un SaaS (mutualisation), protège tous les locataires de la même manière. Toutefois, un locataire possède ses propres règles/exigences de sécurité [21,22,34–36] qui peuvent être en conflit avec un autre locataire [35]. Selon Cloud Security Alliance (CSA) [37, 38], la mutualisation impose des implications de sécurité critiques et complexes au fournisseur qui doit empêcher un locataire d'accéder aux données des autres locataires. Cette mutualisation, considérée comme le problème majeur de sécurité [4, 34, 35, 39], est à l'origine des attaques entre les locataires qui partagent la même application, service ou base de données [4, 21, 22, 34, 35, 37–49]. D'où la nécessité de nouveaux mécanismes permettant de s'assurer qu'un locataire ne peut pas compromettre les données des autres locataires, surtout lorsque ces derniers partagent certaines tables de la même base de données.

Afin de résoudre ces problèmes de sécurité causés par la mutualisation et l'externalisation des ressources, une nouvelle approche est nécessaire pour prévenir les attaques entre les tenants d'un SaaS multi-locataire avec un impact acceptable sur le temps de réponse aux requêtes HTTP. Un nouveau type d'IDS est aussi nécessaire, à savoir, l'IDS multi-locataire (multi-tenant IDS en anglais). Bien que cet IDS soit partagé, en tant que service, par plusieurs locataires, chacun d'eux doit être en mesure de souscrire à la demande et de personnaliser cet IDS en fonction de certains facteurs (sensibilité des données, budget, etc.). Par exemple, chaque locataire doit être capable de se protéger contre certains types d'attaques. Plusieurs facteurs et défis compliquent la conception d'un IDS multi-locataire pour les SaaS dont : (i) le grand nombre de locataires ; (ii) la grande quantité de trafic qui varie d'un locataire à un autre ; (iii) la variation des exigences/politiques de sécurité entre les locataires qui partagent certaines tables d'une base de données ; (iv) le regroupement du trafic par locataire en temps réel ; (v) la distribution adéquate du coût entre les locataires pour assurer un paiement à l'usage pour chacun d'eux ; (vi) l'intégration de cet IDS dans une infonuagique publique pour protéger un SaaS sans accéder à son code source afin d'assurer d'un niveau élevé de portabilité ; (vii) l'identification des locataires (malintentionnés ou coûteux) qui consomment

plus les ressources de cet IDS.

Dans le contexte de la problématique décrite précédemment, cette thèse s'intéresse aux trois questions de recherche suivantes :

1. Comment permettre au fournisseur SaaS d'intégrer un IDS dans son infrastructure louée (IaaS) en tant que service (libre-service, élasticité des ressources, mutualisation des ressources, paiement à l'usage et haute disponibilité), tout en assurant un degré élevé de portabilité ?
2. Comment permettre au fournisseur SaaS de détecter et/ou prévenir les attaques menées par un locataire contre les données des autres locataires d'un SaaS multi-locataire ?
3. Comment un fournisseur SaaS peut partager un IDS entre plusieurs locataires tout en permettant à chacun de personnaliser cet IDS partagé en tant que service (libre-service et paiement à l'usage) ?

1.3 Objectifs de recherche

L'objectif principal de cette thèse est de proposer de nouveaux mécanismes pour détecter et/ou prévenir les attaques qui visent les applications SaaS. Ces mécanismes sont offerts comme services de sécurité au fournisseur et à ses locataires. Plus spécifiquement, cette thèse vise à :

1. Permettre au fournisseur SaaS d'intégrer un IDS existant comme service dans son infrastructure infonuagique (IaaS) pour détecter les attaques qui visent les bases de données d'un SaaS, comme les injections SQL. Ce cadre a été conçu sous forme de VMs afin de respecter les caractéristiques d'un service (libre-service, élasticité des ressources, mutualisation des ressources, paiement à l'usage et haute disponibilité). Il est ensuite validé dans une infonuagique publique (AWS) afin de montrer sa faisabilité et d'évaluer son surcoût (processeur et mémoire) et son impact sur le temps de réponse aux requêtes HTTP.
2. Offrir aux fournisseurs SaaS une approche de détection et de prévention des attaques entre les locataires d'un SaaS multi-locataire avec très peu d'impact sur le temps d'exécution des requêtes HTTP. Cette approche a été intégrée dans AWS selon une double architecture de détection et de prévention. Ces architectures ont été validées et évaluées comparativement selon certains critères (coût, impact sur le temps de réponse HTTP, portabilité, compatibilité, disponibilité, facilité d'intégration et paiement à l'usage) dans AWS.

3. Permettre au fournisseur SaaS de partager un IDS (intégré dans infonuagique publique) entre plusieurs locataires, ce qui réduit le coût d'infrastructure, d'opérations et de maintenance. En plus, chaque locataire doit personnaliser cet IDS selon ses propres exigences de sécurité et son budget en libre-service tout en payant à l'usage. Afin de résoudre le problème de paiement à l'usage, deux modèles mathématiques ont été proposés pour quantifier le coût de ce service de sécurité payé par le fournisseur et chaque locataire.

1.4 Plan de la thèse

Dans ce chapitre, nous avons présenté les concepts de base, les éléments de la problématique et les objectifs de notre recherche. Dans le deuxième chapitre, nous abordons une revue de littérature sur la sécurité infonuagique, l'isolation des données des locataires, les contremesures des applications et services web et les IDS infonuagiques. Dans le troisième chapitre, nous décrivons la démarche de l'ensemble du travail de recherche. Nous présentons nos trois articles scientifiques du quatrième au sixième chapitre. Dans le septième chapitre, nous réalisons une discussion générale sur les résultats obtenus de notre étude selon certains critères. Enfin, dans le huitième chapitre, nous présentons la synthèse et les limites des solutions proposées ainsi qu'un aperçu de nos futurs travaux.

CHAPITRE 2 REVUE CRITIQUE DE LA LITTÉRATURE

Ce chapitre aborde la sécurité de l'infonuagique. Il présente ensuite une revue de littérature sur les mécanismes d'isolation des données des locataires, les contremesures des applications Web et les IDS infonuagiques.

2.1 Sécurité de l'infonuagique

Cette section présente les sources des vulnérabilités et les objectifs de sécurité dans l'infonuagique.

2.1.1 Sources des vulnérabilités

L'externalisation, la mutualisation des ressources et la complexité représentent les principales sources de la majorité des vulnérabilités spécifiques de l'infonuagique [35, 50].

V1 : Externalisation

L'acceptation d'une organisation d'externaliser ses infrastructures, données ou applications chez une tierce partie implique plusieurs défis et risques liés notamment à la perte de contrôle, à la localisation de données et à la vérification de la confidentialité, de l'intégrité et de la disponibilité [35]. Contrairement au fournisseur, un client perd plus de contrôle (et de responsabilité) sur ses ressources/services loués en allant du IaaS au PaaS et du PaaS au SaaS [50, 51].

V2 : Mutualisation des ressources

La virtualisation permet aux clients de partager les ressources hébergées dans une infrastructure publique. Cela réduit le coût du matériel et de la maintenance et augmente la surface d'attaques visant les trois modèles de services infonuagiques (IaaS, PaaS et SaaS). D'où la nécessité d'une séparation physique ou virtuelle entre les locataires [35]. Pour mieux illustrer nos propos, considérons, par exemple, le modèle IaaS où un seul hôte physique contient les machines virtuelles des différents locataires, dans ce cas, un locataire malintentionné peut réaliser des attaques contre les machines virtuelles d'autres locataires [35, 50, 51].

V3 : Complexité

L'allocation dynamique et rapide des ressources ainsi que la quantité massive des données réduisent la capacité en termes de performances des mécanismes traditionnels de sécurité [35]. Cette complexité empêche de juger si une consommation énorme de ressources (processeur ou mémoire) est normale (pour répondre à l'élasticité des ressources) ou anormale (causée par une DoS ou XDoS).

2.1.2 Objectifs de sécurité

Les objectifs de la sécurité infonuagique diffèrent des autres environnements par l'impact des vulnérabilités présentées dans la section précédente [35, 50] :

- Confidentialité est la garantie de manière que les données/ressources d'un locataire ne soient pas accessibles par le fournisseur et les autres locataires [35]. Dans l'infonuagique, cette propriété peut être violée par un fournisseur (externalisation) ou par un locataire (mutualisation).
- Intégrité est la capacité de détecter la perte/modification des données ou la violation de l'exécution normale d'un programme par un malware, un fournisseur ou par un locataire [35]. Il est difficile pour un locataire de valider l'intégrité des données sur des serveurs infonuagiques gérés par un fournisseur qui ne dévoile pas souvent les incidents de sécurité. La quantité massive de données complique énormément cette tâche, surtout avec le nombre élevé des parties prenantes (administrateurs/usagers) qui peuvent violer cette intégrité.
- Disponibilité est la garantie qu'un service infonuagique soit disponible à la demande, en respectant le niveau de la qualité de service défini dans un SLA (for Service Level Agreement) [35]. L'indisponibilité partielle a des répercussions sur l'adoption de l'infonuagique qui se caractérise par la haute disponibilité d'un service à la demande.
- Responsabilité (Accountability en anglais) est la capacité de déterminer le responsable (fournisseur ou locataire) d'une activité [35]. Cette propriété est extrêmement importante dans une infonuagique publique dans laquelle coexistent plusieurs parties prenantes (administrateurs du fournisseur, locataires, etc.). L'assurance de cette propriété est difficile à garantir avec la présence d'une quantité massive de ressources informatiques, d'un nombre important de machines virtuelles sur plusieurs hôtes physiques et des parties prenantes multiples. Cette complexité multiplie donc les risques d'erreurs humaines et logicielles ainsi que les violations du SLA.

Positionnement des travaux

Dans cette thèse, nous tentons de réduire les vulnérabilités et couvrir les objectifs de sécurité précédents dans le contexte du SaaS. En effet, l'externalisation permet au fournisseur SaaS de migrer ou déployer ses applications dans une infrastructure infonuagique (voir Section 1.1.2). Ensuite, un locataire loue un domaine au lieu de développer et déployer des applications à l'interne. Par conséquent, le fournisseur n'a pas le choix d'adapter ou d'intégrer des nouvelles mesures de sécurité dans l'infonuagique. Il peut aussi réduire largement les coûts d'infrastructure et de maintenance en mutualisant l'application, services et bases de données entre plusieurs locataires. Mais, cela peut conduire un locataire à compromettre accidentellement ou malicieusement la confidentialité, l'intégrité et la disponibilité des données d'un autre locataire. Afin de résoudre les problèmes générés par l'externalisation et la mutualisation, nous proposons des approches prometteuses aux fournisseurs afin de surveiller et protéger les services web et les bases de données des SaaS en tenant en compte les exigences de sécurité de chaque locataire.

2.2 Attaques et risques de sécurité du SaaS

Un SaaS est basé sur une architecture web orientée service. Il hérite donc des vulnérabilités et des risques de sécurité des applications et des services Web. Il représente une cible d'attaques qui visent la couche application [21]. Selon le projet OWASP (Open Web Application Security Project) [52], les dix principaux risques de sécurité des applications Web en 2017 sont (plus au moins critique) :

- A1 : l'attaque par injection SQL (SQLIA) qui se produit lorsqu'un attaquant injecte des mots clés SQL (ex : OR, AND) ou caractères spéciaux (ex : –,') dans les entrées d'utilisateur afin de générer des requêtes SQL malveillantes (imprévues) lors de l'exécution ;
- A2 : l'implémentation incorrecte des mécanismes d'authentification conduit un usager malveillant à accéder aux mots de passe, aux clés ou aux sessions d'autres utilisateurs ;
- A3 : l'absence de chiffrage des données qui permet à un attaquant de voler des données sensibles (carte de crédit) qui sont manipulées par les services ou applications Web ;
- A4 : l'attaque par déni de service XML (XDoS) qui se produit lorsque les processeurs XML ne valident pas les entités externes injectées par un attaquant ;
- A5 : l'application incorrecte des restrictions/privilèges de sécurité qui accorde à un attaquant l'accès à des fonctionnalités et/ou des données illégitimes ;
- A6 : la mauvaise configuration de sécurité (ex : en-têtes HTTP/SOAP) qui ouvre la voie à un attaquant de compromettre l'application, les services et les données ;

— A7 : l'attaque Cross-Site Scripting (XSS) qui simplifie l'exécution d'un script malveillant (HTML ou JavaScript) dans le navigateur d'une victime pour pirater les sessions utilisateur ou rediriger l'utilisateur vers des sites malveillants ;
— A8 : la désérialisation non sécurisée (sans contrôle d'intégrité) des objets peut mener à l'exécution de code malveillant ;
— A9 : l'utilisation de composants (ex : bibliothèques, cadres, etc.) contenant des vulnérabilités connues peut être facilement exploitée par des attaquants ;
— A10 : la surveillance insuffisante de l'application ou des services aide un attaquant à extraire ou à détruire des données. Nous distinguons deux catégories principales de sécurité pour une application SaaS : la sécurité des services Web qu'elle invoque et la sécurité des données qu'elle manipule.

2.2.1 Sécurité des services Web

Les services infonuagiques, y compris les SaaS, sont offerts par les services Web [53]. Une application SaaS appelle donc des services Web qui sont souvent des cibles d'attaques. Les concepteurs ont donc proposé des standards ou spécifications pour assurer la sécurité de ces messages à savoir, XML-Encryption pour la confidentialité et XML-Signature pour l'intégrité et l'authenticité [54–56]. Cependant, aucune spécification n'a été proposée pour assurer la disponibilité des services Web.

Une application SaaS, qui invoque des services Web, doit rester disponible malgré les attaques effectuées. L'indisponibilité d'un service Web est donc le risque le plus critique qui affecte plusieurs locataires partageant la même instance (ou utilisant différentes instances) d'une application SaaS. L'une des plus graves menaces pour la disponibilité des services infonuagiques est l'attaque par déni de service XML [26, 53].

L'attaque par déni de service basée XML

L'attaque par déni de service (DoS for Denial of Service) consiste à rendre une ressource informatique indisponible pour les usagers légitimes [57]. La recherche, l'invocation et l'exécution des services Web utilisent le protocole SOAP. Un attaquant peut profiter de l'ouverture et de la richesse du langage XML pour envoyer un long message dont le traitement nécessite beaucoup de ressources système. Le but de ce message est d'accaparer et épuiser les ressources du serveur Web. Ce type d'attaques, connu aussi sous l'acronyme XDoS, cible les analyseurs XML (XML parsers en anglais) [58].

Dans l'infonuagique, une EDoS (for Economic Denial of Sustainability) sert à provoquer une consommation des ressources supplémentaires (payées par un consommateur) durant la détection d'une XDoS sans la nécessité d'arrêter le service cible [57]. De ce fait, une EDoS peut donc nuire à la réputation du fournisseur, provoquer une violation du SLA et détériorer la confiance entre le fournisseur et le consommateur [35, 57]. Les quatre principaux types de XDoS (et EDoS) sont les suivants [58] :

— Coercive parsing : l'attaquant envoie un message SOAP avec des nœuds imbriqués et cycliques. Si l'analyseur XML traite ce message sans validation, l'attaquant provoque un traitement sans fin d'appels par le processeur.
— Oversize Payload : l'attaquant envoie un message SOAP avec énormément de données et d'éléments. Si aucune contrainte n'est exprimée (ex : MaxOccurs='Unbound') et l'analyseur XML charge ce message au complet sous forme d'arbre dans la mémoire, un grand espace mémoire est consommé.
— WSDL Flooding : l'annuaire UDDI est accessible (parfois sans authentification) au public. Un attaquant peut donc inonder cet annuaire en envoyant un nombre énorme de requêtes WSDL.
— Malformed external Schema Referencing : la syntaxe XML accepte d'intégrer un lien vers un schéma XML externe. Un attaquant peut donc injecter un ou plusieurs liens vers certains schémas XML externes malveillants dans le message SOAP original. La surcharge de l'analyseur XML augmente considérablement lorsqu'il reçoit et traite ces schémas malveillants.

2.2.2 Sécurité des données

Le fournisseur d'un SaaS mutualise un serveur (modèle base-par-locataire), une base (modèle schéma-par-locataire) ou certaines tables (modèle tables-partagées) de données entre plusieurs locataires (voir la section 1.1.2). Il héberge aussi ces données dans une infonuagique publique (externalisation). Cette mutualisation et externalisation des données compliquent la protection des données des locataires contre les attaques réalisées par les usagers/administrateurs malintentionnés. Cela pose certains risques et défis supplémentaires liés à la disponibilité, à la confidentialité, à l'intégrité et à la localisation des données [4, 50, 59] :

— Disponibilité : les serveurs de données (PaaS) doivent rester disponibles pour assurer aux usagers légitimes l'accès à leurs données. Le fournisseur SaaS est responsable d'arrêter les attaques qui visent la disponibilité de ces serveurs.
— Confidentialité et intégrité : un locataire (ou administrateur) malveillant peut extraire (confidentialité) ou altérer (intégrité) les données des autres locataires. Un des défis est de vérifier l'intégrité des données externalisées et gérées par une tierce partie

(fournisseur SaaS).

— Localisation : les données d'une application SaaS peuvent être situées dans différents pays qui ont des lois différentes par rapport à la confidentialité, à la vie privée et au transfert des données [4].

Pratiquement, les attaquants exécutent différents types de SQLIA pour menacer la disponibilité, la confidentialité et l'intégrité des données manipulées par les applications SaaS : les vulnérabilités les plus répandues dans le modèle SaaS sont les injections SQL [4] qui représentent 97% des attaques contre les données [60].

L'attaque par injection SQL

L'attaque par injection SQL (SQLIA) est l'attaque la plus critique qui viole la disponibilité, la confidentialité et l'intégrité des données d'une application SaaS. Pour réaliser une SQLIA, des caractères dangereux[1] (', /, –, etc.) et des mots clés SQL (union, drop, etc.) sont injectés par les entrées d'utilisateur, les cookies, les variables du serveur et les adresses URL [61–63] dans le but de changer la structure (ou la sémantique) des requêtes SQL [61,62]. Un exemple classique de vulnérabilité face à cette attaque est le non vérification de l'existence de ces caractères dangereux lors de la construction des requêtes SQL dynamiques. Nous expliquons, ci-après, les principaux types de SQLIA [63] à l'aide d'une requête dynamique d'authentification basée sur l'identifiant d'un locataire (pTenantID) et les deux champs d'entrée : « txtUsager » et « txtMotPasse » comme suit :

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

— Requête syntaxiquement incorrecte : le but ici est de provoquer des messages d'erreurs significatifs sur l'application Web, le serveur Web, le DBMS et la structure des données. Par exemple, un attaquant peut saisir (') dans txtUsager et (123) dans txtMotPasse pour produire la requête SQL suivante :

```
Select * from TblUsagers Where Usager='' ' and
MotPasse='123' and TenantID =1
```

L'attaquant a donc pu provoquer le message d'erreur significatif suivant :

```
102, State 1, Line 3 Incorrect syntax near '123'.
```

Le numéro du message (102) permet de reconnaitre que le DBMS de l'application Web

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.

(ou SaaS) est MS SQL (ou MS SQL Azure).

— Tautologie : le but ici est d'éviter l'authentification pour se connecter comme un utilisateur normal (ou même comme un administrateur) de l'application. Pour ce faire, l'attaquant du tenant1 injecte des caractères et des opérations de sorte que la requête demandée retourne toujours la valeur booléenne (vrai), par exemple, en saisissant (`Admin' OR 1=1-`) dans txtUsager et n'importe quelle valeur dans txtMotPasse pour produire la requête SQL suivante :

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

La condition (1=1) est toujours vérifiée et le reste de la commande est mis en commentaire grâce à la chaîne de caractères « – ». L'attaquant a donc réussi à s'authentifier en tant qu'administrateur.

— Union de requêtes : le but ici est d'injecter une requête supplémentaire pour qu'elle soit concaténée avec la requête existante afin de retourner des résultats recherchés par l'attaquant, par exemple, en saisissant (`' UNION select * from TblUsagers -`) dans txtUsager et (`123`) dans txtMotPasse pour produire la requête SQL suivante :

```
Select * from TblUsagers Where Usager=''
UNION
select * from TblUsagers -' and MotPasse='123' and TenantID =1
```

La première requête ne retourne aucune ligne, mais la seconde retourne toutes les lignes de la table TblUsagers. L'attaquant a donc réussi à obtenir les coordonnées des usagers de tous les clients de l'application SaaS.

— Commande SQL spécifique : le but ici est d'exécuter des commandes malveillantes SQL (ou système) sur le serveur de données, par exemple, en saisissant (`' OR 1=1 ; Drop Table TblUsagers -`) dans txtUsager et (`123`) dans txtMotPasse pour produire la requête SQL suivante :

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

L'attaquant a donc réussi à détruire la table (TblUsagers), qui contient les informations des usagers de tous les locataires d'une application SaaS, à l'aide de la commande (`Drop Table`). Il peut aussi mettre le DBMS hors service en injectant la commande (`Shut Down`) au lieu de (`Drop Table`).

Comme nous l'avons vu à l'aide des exemples, les conséquences des vulnérabilités et attaques précédentes sont plus graves sur les SaaS que sur les applications web classiques [4, 21, 22]. Par exemple, les données de tous les locataires partageant la même base de données peuvent

être compromises lorsqu'un attaquant réalise une SQLIA. De plus, si un attaquant met hors service un service web via une XDoS, les données traitées par ce service victime seront indisponibles pour tous les locataires.

Positionnement des travaux

Dans cette thèse, nous permettons au fournisseur de surveiller automatiquement les SaaS pour détecter les attaques applicatives en intégrant plusieurs techniques de détection d'intrusion existantes. Cependant, la mutualisation des ressources (web services et bases de données) est la source de nouveaux risques de sécurité, en particulier, les attaques entre les locataires qui partagent la même application et/ou base de données. Par exemple, un locataire peut accéder (intentionnellement ou accidentellement) aux données d'autres locataires en lecture ou écriture en combinant plusieurs types d'attaques [4, 21, 22, 34, 35, 37–49].

2.3 Isolation des données de locataires

Certains fournisseurs IaaS suggèrent des mécanismes d'isolation complète entre les clients d'un SaaS. Par exemple, AWS propose que le fournisseur SaaS déploie l'application et la base de données de chaque client dans un compte ou sous-réseau séparé [64]. Cependant, cette isolation demeure coûteuse en termes de ressources et ne respecte pas l'architecture d'un SaaS multi-locataire où les locataires partagent la même base de données.

Quelques approches de chiffrage assurent une certaine confidentialité des données en attribuant une clé spécifique à chaque locataire [49, 65]. Même si ces approches empêchent un locataire malveillant de lire les données chiffrées des autres locataires, elles n'assurent pas l'intégrité des données. Par exemple, un locataire peut exécuter une requête d'insertion, de suppression ou de mise à jour (insert, delete ou update) non-autorisée contre d'autres locataires. De plus, seulement 9,4% de fournisseurs chiffrent les données des locataires dans l'infonuagique [66].

Les développeurs du SaaS peuvent appliquer certains patrons de conception pour s'assurer que les données d'un locataire ne seront pas manipulées par un autre locataire [67, 68]. Un de ces patrons consiste à implémenter les requêtes SQL avec un filtrage sur le locataire correspond (l'invocateur de la requête). Par exemple, la requête SQL suivante est créée dans le code source d'une application ou un service :

```
"Select * from Orders Where TenantID=" + pTenantID
```

Au niveau de la base de données, les administrateurs peuvent gérer des droits d'accès pour ré-

duire le risque des accès illégitimes entre les différents locataires. Dans le modèle des données (schéma-par-locataire), un compte peut être crée pour chaque locataire avec des droits d'accès à ses tables. Cette gestion est malgré tout très coûteuse en termes de temps et d'espace, notamment, dans le cas de centaines voire de milliers de locataires. Cependant, il n'existe pas des contrôles d'accès sur les enregistrements d'une table partagée par plusieurs locataires (tables-partagées). Ces patrons sont manuellement implémentés par les développeurs qui peuvent être incompétents. D'où la nécessité des outils de vérification pour valider leur efficacité avant la mise en production des SaaS. De plus, un attaquant peut contourner ces patrons au moment de l'exécution des requêtes SQL. Par exemple, un usager malveillant d'un locataire peut injecter `1' OR 1=1 -` dans le paramètre (ptenantID) pour générer la requête suivante qui retourne les ordres de tous les locataires.

```
select * from Orders where TenantID=1' OR 1=1-
```

Très peu d'articles scientifiques traitent le problème d'isolation des données des locataires d'un SaaS de point de vue sécurité [40, 69]. Les auteurs de [69] proposent d'installer un proxy HTTP dans le réseau interne de chaque locataire afin de signer les requêtes HTTP à l'aide d'une clé fournie par le fournisseur SaaS. Cette proposition est inappropriée aux nouveaux SaaS où les serveurs web et les données sont déployés dans l'infrastructure du fournisseur. Deux algorithmes tentent de réduire le risque des attaques entre les locataires en évitant (autant que possible) de placer les données de deux locataires compétiteurs dans la même base de données [40]. Les données du nouveau locataire sont aléatoirement placées dans une base de données si toutes les bases de données contiennent au moins les données d'un locataire compétiteur. Il existe donc une forte possibilité que les données de plusieurs locataires compétiteurs soient dans la même base de données. Sinon, il va falloir dédier une base de données pour chaque locataire (base-par-locataire).

Positionnement des travaux

Dans cette thèse, nous proposons une nouvelle approche pour détecter et prévenir les attaques (et accès illégitimes) aux données d'un locataire par l'usager d'un autre locataire d'un SaaS. Nous privilégions le modèle de données qui consiste à enregistrer les données de plusieurs locataires (qui peuvent être des compétiteurs) dans certaines tables de la même base de données, car il est le moins coûteux mais aussi le moins sécuritaire. Contrairement aux approches, reposant sur une politique de contrôle d'accès, une détection d'intrusion par signatures ou une analyse statique du code source, notre approche repose sur l'analyse dynamique de l'arbre syntaxique de requêtes SQL au moment de l'exécution. Cela permet de détecter les attaques inconnues d'une manière automatique sans accéder au code source ou

aux bases de données.

2.4 Contremesures des applications/services Web

Afin d'éliminer certains risques applicatifs de sécurité (Section 2.2), les développeurs et administrateurs des SaaS peuvent implémenter des mesures de protection comme l'authentification (A2), le chiffrement (A3), l'amélioration des restrictions (A5) ou la bonne configuration (A6). Cependant, ces mesures ne protègent pas les applications/services contre toutes les attaques (comme SQLIA, XSS et XDoS) qui peuvent s'exécuter via des requêtes HTTP, HTTPS (chiffrées avec TLS) et SQL. C'est pour cela que plusieurs IDS ou contre-mesures ont été proposés pour détecter ces attaques. Cette section présente brièvement quelques contremesures des attaques les plus courantes qui ciblent les SaaS : SQLIA et XDoS [21].

2.4.1 SQLIA

Il existe de nombreuses contremesures du SQLIA qui s'appuient sur l'analyse statique et/ou dynamique des requêtes HTTP/SOAP et / ou des requêtes SQL [23–25,28–31]. Un IDS qui se base sur la détection d'anomalies [24] combine l'analyse du trafic HTTP/SOAP et SQL pour réduire les erreurs et les faux positifs. Une bibliothèque anti-SQLIA [25] permet de générer et valider l'arbre syntaxique de chaque requête SQL contre les modèles d'attaques. Cette bibliothèque ne traite pas les requêtes HTTP/SOAP. Dans [28], un automate est généré pour chaque requête SQL en analysant le code source d'une application Web PHP : une requête SQL est considérée comme SQLIA (en exécution), si elle n'est pas conforme à l'automate généré. L'outil SQLCheck [29] vérifie la conformité des requêtes SQL aux modèles de requêtes légales. SQLProb [30] propose une approche d'apprentissage qui valide l'entrée utilisateur dans la requête SQL générée pour apprendre la structure de requête normale de manière dynamique (au moment de l'exécution). L'outil CANDID [31] compare l'arbre syntaxique de chaque requête SQL dynamique avec et sans les entrées d'utilisateur. Toute divergence est considérée comme une SQLIA.

2.4.2 XDoS

L'article [70] traite un type de XDoS, appelé Stealth (en anglais), qui consiste à envoyer des messages XML avec un nombre dynamique de balises imbriquées et une fréquence aléatoire afin d'éviter les approches de détection avec des seuils fixes. L'identification de cette attaque se fait en observant les symptômes d'attaque (le type, la répartition et le nombre des balises imbriquées) dans les messages SOAP et l'utilisation du processeur. Ensuite, le système annule

tous les messages avec un nombre de balises imbriquées supérieur à un seuil T (déterminé durant la phase d'apprentissage). Ensuite, le système diminue dynamiquement T, de sorte qu'il rejette aussi les messages de l'attaque avec un faible nombre de balises imbriquées, jusqu'à ce que l'utilisation du processeur tombe au-dessous d'un niveau de gravité. Cette approche a été validée en fonction du nombre d'interactions Web traitées par seconde (WIPS) durant trois fenêtres de temps : opérations normales, sous l'attaque et après la réaction. Le WIPS a diminué jusqu'à 93% sous l'attaque et a augmenté jusqu'à 12% pour un T fixe et de 69% pour un T dynamique après la réaction (filtrage des messages). Le seuil dynamique, la phase de reprise ainsi que la corrélation entre les symptômes d'attaque et l'utilisation du processeur ont largement amélioré la détection de la XDoS de type (coercive parsing en anglais), en réduisant la période d'indisponibilité du service Web victime. Cependant, l'approche a été validée avec très peu des messages SOAP malveillants.

L'article [57] évalue le frais provoqués par une EDoS (voir Section 2.2.1) dans AWS. L'environnement du test est constitué d'un serveur de 4 VMs qui contiennent un service Web ciblé par 1800 requêtes HTTP-SOAP malveillantes. Ces requêtes ont générées des frais supplémentaires d'environ 120 USD. La solution proposée dans cet article utilise la cryptographie à clé publique et installe un pare-feu qui intercepte et valide les demandes client en se basant sur deux listes d'adresses IP (noire et blanche). Chaque client doit donc résoudre un problème mathématique envoyé par le serveur pour s'authentifier et accéder au service demandé. Sinon, il sera considéré comme un attaquant et son IP sera ajouté à la liste noire. Cet article aborde un nouveau type d'attaque spécifique à l'infonuagique tout en montrant ses conséquences. Cependant, la solution proposée est très simple, hérite des points faibles de la cryptographie et n'a pas été concrètement évaluée.

Positionnement des travaux

Certaines contremesures applicatives possèdent une bonne qualité de détection (taux élevé de détection, taux faible de faux positifs et négatifs). Néanmoins, aucune de ces contremesures n'est intégrée dans une infonuagique publique pour protéger les SaaS. Dans cette thèse, nous proposons un cadre de surveillance qui fournit les informations nécessaires (requêtes HTTP et SQL) pour l'intégration des contremesures applicatives.

2.5 IDS infonuagiques

Afin de protéger les applications Web déployées dans une infonuagique publique, certaines entreprises ont migré leurs solutions de sécurité surtout vers AWS. Prevoty [71] fournit des

plug-ins pour certaines environnements de développement (e.g., J2EE, .NET) des applications Web. Imperva [72] offre un WAF (Web Application Firewall) qui peut être associé à une application Web déployée dans une infonuagique publique. D'une part, ces solutions ne dévoilent pas des détails concernant l'approche de détection utilisée. D'autre part, elles ne distinguent pas entre les locataires d'un SaaS et n'offrent pas un service de sécurité adapté aux exigences de sécurité de chacun d'eux.

Les IDS infonuagiques existants peuvent être installés au niveau d'hyperviseur (VMMIDS for Virtual Machine Manager IDS) [73, 74], sur un réseau infonuagique (NIDS for Network IDS) [75–78], sur une machine virtuelle (VMIDS for Virtual Machine IDS) [79–83]. Plusieurs IDS peuvent collaborer (CIDS for Collaboratif IDS) pour améliorer la détection [84–88]. Les IDS qui se basent sur la modification d'hyperviseur sont inappropriés aux SaaS, car les fournisseurs, qui louent des machines virtuelles, sont incapables d'accéder à l'hyperviseur pour installer un IDS. Dans les sections suivantes, nous analysons les NIDS, HIDS et CIDS en tenant compte du type de trafic surveillé et du modèle infonuagique (SaaS, PaaS ou IaaS) couvert.

2.5.1 IDS réseau (NIDS)

Un NIDS est déployé sur certains nœuds stratégiques d'un réseau dans le but d'inspecter les entêtes des paquets. Ces IDS sont capables de détecter ou prévenir certains types d'attaques (DoS, DDoS, balayage de ports, spoofing, etc.).

Dans [75], un modèle de sécurité est offert au fournisseur IaaS et aux locataires des VMs. Le fournisseur peut protéger son infrastructure contre les locataires malintentionnés. Le locataire peut protéger ses VMs contre les attaques internes réalisées par d'autres locataires et les administrateurs du fournisseur. Trois types d'attaques peuvent être détectés : (1) Une DoS exécutée à partir d'une VM d'un locataire contre les VMs des autres locataires ; (2) L'installation des applications illégitimes ou la désactivation des outils de sécurité sur la VM d'un locataire ; (3) Les accès illégitimes réalisés par les administrateurs du fournisseur contre les VMs des locataires. Ce modèle est flexible, car le fournisseur offre au locataire une protection de base (contre les locataires et les administrateurs malveillants) et des services additionnels de sécurité (contre les usagers malintentionnés) à condition que le locataire paie et accepte de dévoiler ses outils et ses applications. Cependant, ce modèle détecte les attaques internes et non pas les attaques provenant, par exemple, des usagers finaux des sites Web.

Dans [76], un NIDS à base de signatures (extension du Snort) est déployé dans une infonuagique virtuellement privée (VPC for Virtual Private Cloud) d'AWS. Les machines virtuelles (IaaS) d'un client peuvent être protégées contre les attaques provenant de l'intérieur et de

l'extérieur de l'infonuagique. Le surcoût de ce NIDS a été évalué selon le temps de réponse, en envoyant des requêtes HTTP malveillantes, qui accèdent à des pages Web non-autorisées, et des fichiers suspects au serveur Web victime. Ce temps a été augmenté de 9.27% (à l'intérieur de l'infonuagique) et de 10.6% (à l'extérieur de l'infonuagique) pour les FTP ainsi que de 3.57% (à l'intérieur de l'infonuagique) et de 8.58% (à l'extérieur de l'infonuagique) pour les HTTP. Malgré cela, le taux de détection, les faux positifs et les faux négatifs n'ont pas été évalués. Cet IDS est flexible, car le fournisseur/client IaaS peut définir ses propres règles/signatures. Par contre, sa portabilité est faible car une application doit être implémentée sous forme d'une VMI (Virtual Machine Instance) pour qu'elle soit protégée. De plus, il utilise la détection par signatures, ce qui exige une intervention humaine fréquente.

Les auteurs de [77] proposent un NIDS pour sécuriser les réseaux des services infonuagiques. Cet IDS peut être personnalisé par les utilisateurs d'infonuagique en saisissant certaines règles de sécurité. Un commutateur (switch en anglais) virtuel est utilisé pour acheminer les paquets à travers plusieurs pare-feu à l'aide de deux tables de routage. Les règles de sécurité sont vérifiées sur chaque pare-feu.

Dans [78], un NIDS, qui se base sur les signatures du Snort, est intégré dans la couche réseau avec la capacité d'ajouter de nouvelles signatures dans le fichier de configuration du Snort pour améliorer l'efficacité de la détection. Ce NIDS ne peut détecter que les attaques connues et certaines attaques dérivantes de celles-ci.

2.5.2 IDS machine (VMIDS)

Un VMIDS surveille et analyse le trafic entrant/sortant des machines virtuelles des locataires. Un VMIDS peut inspecter les détails chiffrés des paquets puisqu'il est installé sur une VM ; Cependant, il peut être victime d'attaques comme les DOS.

Un IDS hybride (Signature et anomalie) [79] est déployé sur une VM pour détecter les attaques réseau (DoS, DDoS, phishing, inondation, etc.). Cet IDS utilise les signatures du Snort et certains classificateurs (bayésien, règle associative et arbre de décision) pour la détection d'anomalies.

Un IDS [80], basé sur la détection d'anomalies, a été proposé pour détecter les programmes malveillants ciblant les VMs d'un environnement infonuagique. Les appels système normaux sont générés pendant la phase d'apprentissage et considérés comme des références pour la détection des programmes malveillants durant la phase de détection.

Certaines intrusions systèmes/réseaux connues (par exemple, DDoS) sont détectées en déployant les signatures de Snort sur chaque VM de l'infonuagique (privé et public) [81].

Un IDS est installé entre le routeur et l'hôte virtuel pour analyser les TCP/IP des paquets à destination de chaque VM [82]. Cette analyse est effectuée en fonction des signatures (Cloud Data Sets) qui doivent être manuellement préparées en fonction des ports ouverts (TCP/IP).

Un IDS hybride [83] consiste à créer un profil (processeur, mémoire, etc.) pour chaque VM, à comparer le débit des paquets entrants avec un seuil et à analyser périodiquement les vulnérabilités. Toutefois, cet IDS n'est ni implémenté ni évalué.

2.5.3 IDS collaboratif (CIDS)

Une approche collaborative d'IDS [84] consiste à déployer une instance d'un NIDS dans chaque région infonuagique. Ces NIDS échangent les alertes pour réduire les risques des DoS (et DDoS). Si l'IDS attaché à une région détecte une DoS, il avise les IDS des autres régions en envoyant les informations pertinentes (IP, etc.). L'IDS récepteur réalise un vote : une alerte est acceptée, si elle est envoyée par la moitié d'IDS (émetteurs). Techniquement, les modules de communication et de coopération Inter-IDS ont été implémentés et intégrés dans Snort. Cette approche détecte les attaques provenant d'une même source (IP). Mais, le taux total de détection a diminué de (0.2%) et l'agent coopératif de chaque IDS envoie les alertes à tous les autres agents, ce qui augmente largement les échanges et peut saturer le trafic réseau. De plus, l'expérimentation se limite à deux réseaux qui envoient la même alerte à un troisième réseau.

Les auteurs de [85] proposent de déployer un NIDS (Snort) sur un commutateur réseau virtuel pour surveiller les paquets réseaux entrants et un HIDS sur chaque machine virtuelle. Un coordinateur central collecte les alertes générées par ces deux IDS. Cependant, cette approche n'a pas été implémentée ni expérimentée, le coordinateur central représente un point de défaillance unique et un usager doit analyser manuellement les alertes.

Dans [86], une architecture d'un IDS à base de signatures, hybride (NIDS et HIDS) et collaboratif divise l'infrastructure (IaaS) en trois couches logiques : réseau, hôte et globale. Les routeurs sont équipés par des NIDS. Les hôtes physiques de chaque fournisseur sont regroupés en clusters collaboratifs selon la localisation physique. Chaque cluster manipule trois bases locales (logs, signatures et intrusions) qui se synchronisent périodiquement avec les trois bases globales de chaque fournisseur. Chaque hôte physique est équipé d'un HIDS au niveau du système d'exploitation et d'un hyperviseur qui synthétise les machines virtuelles des clients. Chaque VM contient un IDS (VM-IDS) configurable via l'hyperviseur et qui doit obligatoirement envoyer des alertes au HIDS pour mettre à jour les bases locales. Cette architecture permet une collaboration entre les fournisseurs infonuagiques et les clients d'un fournisseur. Un client peut configurer son IDS (VM-IDS), mais il est obligé de partager ses alertes, ce qui

réduit la flexibilité. Cependant, c'est une architecture à haut niveau qui ne propose aucune implémentation et ne donne pas de détails sur les attaques détectées. D'où l'impossibilité de valider pratiquement son efficacité.

Les auteurs de [87] proposent un cadre de collaboration entre les NIDS infonuagiques. La précision de la détection est améliorée en combinant les signatures de Snort avec une approche d'anomalie (Decision Tree Classifier). Afin de prévenir des attaques coordonnées contre une infrastructure infonuagique, une unité de corrélation reçoit les alertes de tous les NIDS.

Dans le but d'optimiser le délai de détection des DDoS, un IDS coopératif a été proposé dans [88]. Cet IDS collecte et corrèle les informations (adresses IP source et destination) reçues de toutes les VMs du réseau. Il a été évalué dans une infonuagique privée en intégrant deux IDS. Il a réussi à réduire le délai de détection, mais il a augmenté les taux de faux négatifs.

Positionnement des travaux

Selon notre revue de littérature, les IDS infonuagiques existants se concentrent sur la couche IaaS (réseaux et machines virtuelles) plutôt que sur la couche SaaS. D'une part, ils n'analysent pas les requêtes HTTP et SQL envoyées sous forme de données encryptées dans les paquets, et pourtant l'analyse de ces requêtes est nécessaire pour la détection des attaques qui ciblent la couche SaaS (applications Web, services Web et bases de données). D'autre part, très peu d'IDS infonuagiques [75, 76] sont offerts en tant que service de sécurité, mais pour les fournisseurs IaaS. De plus, ces IDS n'assurent pas toutes les caractéristiques de l'infonuagique (libre-service, élasticité des ressources, mutualisation des ressources, paiement à l'usage et haute disponibilité). Dans cette thèse, nous tentons de combler cette lacune de recherche en permettant aux fournisseurs SaaS de combiner plusieurs contremesures applicatives (Section 2.4) au sein d'un seul IDS infonuagique. Celui-ci est offert en tant que service de sécurité pour le fournisseur et les différents locataires d'un SaaS. Selon notre constat, la mutualisation des ressources n'a pas été explorée dans le contexte des IDS infonuagiques. Par conséquent, cette thèse vise aussi à partager un IDS infonuagique qui protège la couche SaaS entre les locataires d'un SaaS pour réduire le coût des ressources et de la maintenance.

CHAPITRE 3 OBJECTIFS DE RECHERCHE ET CONTRIBUTIONS ASSOCIÉES

L'objectif principal de cette thèse est de résoudre des problèmes infonuagiques, tels que les risques et les défis de sécurité dans le contexte du SaaS. Dans ce chapitre, nous présentons les démarches qui nous permettent d'atteindre nos objectifs de recherche présentés dans le chapitre d'introduction.

3.1 Premier objectif de recherche

Le premier objectif de notre recherche est la conception d'un cadre infonuagique pour permettre aux fournisseurs SaaS d'intégrer des contremesures de SQLIA. Ce cadre est détaillé dans l'article du quatrième chapitre qui s'intitule « SQLIIDaaS : A SQL injection intrusion detection framework as a service for SaaS providers ». Afin de détecter les SQLIAs visant un SaaS sans accéder à son code source, nous proposons d'extraire les requêtes HTTP et SQL à partir des paquets. Ensuite, chaque SQL est associée à la HTTP correspondante grâce à un algorithme de corrélation qui se base sur la similarité entre les littéraux des requêtes SQL et les paramètres des requêtes HTTP. Ce cadre est développé sous forme des images de VMs, ce qui permet de respecter les caractéristiques d'un service de sécurité telles que : libre-service, élasticité des ressources, mutualisation des ressources et paiement à l'usage.

Ce cadre permet de détecter les attaques par injection SQL qui consistent à modifier la syntaxe des requêtes SQL au moment de l'exécution dans le contexte d'une base de données relationnelle. Par exemple, il ne détecte pas les injections SQL à l'aveugle qui consistent à deviner des informations en se basant sur des valeurs booléennes (vrai ou faux) retournées par une requête SQL ou son temps d'exécution. Il permet d'intégrer des contremesures des injections SQL qui se basent sur l'analyse dynamique des requêtes HTTP et/ou SQL.

Bien que ce cadre permette d'intégrer les différents types des contremesures de SQLIA avec une haute portabilité par rapport aux langages de programmation, il présente certaines faiblesses énumérées comme suit :

- Il ne supporte que le modèle de données qui consiste à créer une base de données pour chaque locataire d'un SaaS.
- Il exige l'acheminement des requêtes HTTP et SQL vers une NAT VM (NATVM for Network Address Translation Virtual Machine), qui représentera un point de défaillance unique.

— Il est incompatible avec le protocole HTTPS, car il ne peut pas extraire les détails des requêtes HTTP qui sont encryptées sur la NATVM.

Nous avons remédié à ces faiblesses dans notre troisième objectif de recherche.

3.2 Deuxième objectif de recherche

Le deuxième objectif de notre recherche est la conception d'un cadre infonuagique pour détecter et prévenir les attaques entre les locataires d'un SaaS qui partagent certaines tables d'une base de données relationnelle. Ce cadre est détaillé dans l'article du cinquième chapitre qui s'intitule « ITADP : an inter-tenant attack detection and prevention framework for multi-tenant SaaS ». Dans cet article, nous définissons une grammaire SQL pour une base de données multi-locataire. Cette grammaire est employée pour vérifier si une requête SQL, exécutée à partir du domaine d'un locataire (www.tenant1.SaaSApp.com), manipule les enregistrements d'autres locataires (tenant2, tenant3, etc.) sans la nécessité d'accéder aux tables partagées de données. Certains algorithmes ont été proposés pour analyser l'arbre syntaxique d'une requête SQL (au moment de l'exécution) dans le but de vérifier sa conformité vis-à-vis de certaines règles. La violation de ces règles est considérée comme une potentielle attaque entre les locataires. Les algorithmes proposés sont optimisés afin de réduire le temps de la détection des attaques entre locataires.

3.3 Troisième objectif de recherche

Le troisième objectif de notre recherche est la conception d'un cadre infonuagique pour intégrer plusieurs IDS applicatives tout en assurant les caractéristiques d'un service de sécurité pour les fournisseurs et locataires d'un SaaS. Ce cadre est détaillé dans l'article du sixième chapitre qui s'intitule « Multi-tenant intrusion detection framework as a service for SaaS ». Dans cet article, nous avons pallié aux lacunes du cadre conçu durant la réalisation du premier objectif comme suit :

— L'analyse des requêtes SQL a été adaptée aux deux autres modèles de données (schéma-par-tenant et tables-partagées).

— Les requêtes HTTP et SQL sont interceptées et corrélées sur la VM qui contient les services Web. Cela rend le cadre compatible avec le protocole HTTPS, tout en évitant d'avoir NATVM comme un point de défaillance unique. Cette amélioration permet aussi d'assurer une haute disponibilité et de réduire l'impact sur le temps de réponse aux requêtes HTTP, puisqu'il n'est plus nécessaire d'acheminer les requêtes vers NATVM.

— Une élasticité des ressources est assurée en diminuant et augmentant (automatiquement) le nombre des VMs en fonction de l'utilisation du processeur. Cela permet aussi de réduire le coût global du cadre.
— Un paiement à l'usage est assuré grâce à la proposition d'un modèle mathématique de coût fournisseur.

Afin d'offrir ce cadre en tant que service de sécurité multi-locataire, les requêtes HTTP et SQL sont regroupées par locataire (en temps réel). Ensuite, chaque locataire peut spécifier et modifier ses exigences de sécurité, par exemple, pour se protéger contre certaines attaques en libre-service. Il paie aussi à l'usage grâce à la proposition d'un modèle mathématique de coût locataire qui prend en considération la quantité du trafic et les exigences de sécurité de chaque locataire.

3.4 Conclusion

Dans ce chapitre, nous avons présenté notre démarche scientifique qui trace le cheminement choisi pour atteindre l'objectif principale de cette thèse. Les trois chapitres suivants détaillent les trois articles scientifiques qui nous permettent d'étayer la réalisation de nos trois objectifs de recherche.

CHAPITRE 4 ARTICLE 1 : SQLIIDaaS : A SQL INJECTION INTRUSION DETECTION FRAMEWORK AS A SERVICE FOR SAAS PROVIDERS

Mohamed Yassin, Hakima Ould-Slimane, Chamseddine Talhi and Hanifa Boucheneb
2017 IEEE 4th International Conference on Cyber Security and Cloud Computing

Abstract

Recently, we are attending to the proliferation of Cloud Computing (CC) as the new trending internet-based-Platform. Thanks to the outsourcing paradigm, CC is enabling many services. Software as a Service (SaaS) is one of those cloud-based-services. Indeed, SaaS model allows providers to reduce the cost of maintenance and management by transferring traditional on premise deployment to public Cloud. Clients can subscribe, in self-service, to SaaS services based on a pay-per-use model. However, since user data are outsourced to the Cloud, serious security breaches are rising and could harm the reputation of providers and slow down the subscription of clients. SQL injection attack (SQLIA) is one of the most critical SaaS vulnerabilities that allows attackers to violate the availability, confidentiality and integrity of user data.

In this paper, we propose SQL injection intrusion detection framework as a service for SaaS providers. Our framework allows a SaaS provider to detect SQLIAs targeting several SaaS applications without reading, analyzing or modifying the source code. To achieve SQL query/HTTP request mapping, we propose an event correlation based on the similarity between literals in SQL queries and parameters in HTTP requests. Our framework is integrated and validated in Amazon Web Services (AWS). A SaaS provider can subscribe to this framework and launch its own set of virtual machines, which holds on-demand self-service, resource pooling, rapid elasticity, and measured service properties.

4.1 Introduction

Cloud Computing (CC) refers to an infrastructure as a Service (IaaS), platform as a Service (PaaS), or Software as a Service (SaaS) offered by a provider through the Internet [5]. Several customers share and use these services according to self-service, on-demand and pay-per-use characteristics [5]. A SaaS is a service-oriented Web application hosted in a CC environment [11]. The platform (PaaS) of an SaaS application consists of Web and database servers that should be installed on virtual machines (IaaS). A SaaS provider can reduce maintenance and infrastructure-related management costs in traditional on-promise environments as well as

start-up costs by gradually hiring compute resources in accordance with customer growth [89]. A client or tenant benefits from the SaaS model as a subscription, and not in terms of a license, and pays only in relation to usage (e.g., number of users) [89]. According to Gartner, the 2012 revenue for SaaS applications was approximately $14.5 billion, with a growth of 18.5% in comparison with 2011 [90].

A SaaS application inherits the vulnerabilities of Web applications and the service-oriented architecture (SOA) that have increasingly become victims of SQLIA. About 97% of attacks against data are still due to SQLIAs [60], which can allow an attacker to gain complete control of a database [91]. The SaaS/application (39% of attacks) is the most vulnerable service model, followed by the PaaS/platform/OS (representing 23% of attacks) [4]. SQLIAs are the most common vulnerabilities in the SaaS layer [4] and are consistently at the top of the OWASP's list of the most critical application security risks [61].

AWS, the most popular IaaS provider, offers significant protection against traditional network security issues such as IP spoofing and port scanning [4]. However, it does not provide security measures for SaaS applications deployed in its infrastructure. Force.com, the most popular PaaS provider, gives coding tips to protect against SQLIAs without providing automatic solutions [92]. However, the implementation of IDS, specifically for SaaS applications , enables SaaS providers to increase their customers by convincing them that the confidentiality, integrity, and availability of data are well insured.

An intrusion detection system (IDS) is used to detect suspicious or unauthorized activities within an information processing system [93, 94]. Two approaches are commonly used for intrusion detection : misuse and anomaly detection [93]. The misuse approach is to build signatures of known attacks that must be updated regularly by an administrator. Any activity that matches a signature is considered as an attack. The anomaly approach is in two phases : the learning phase that models the normal behavior of the entity to protect (e.g., network, system, application, service) and the detection phase that generates an alert when a deviation from the normal behavior is observed. An IDS can be classified according to the source of information that is inspected : host (HIDS), network (NIDS) [93,94]. The HIDS is installed on a physical host or virtual machine (VM) to analyze data from packets. The NIDS is installed on strategic nodes in a network to analyze the header of packets.

The most of proposed SQLIA countermeasures [28,29,31,95,96] require costly analysis and/or modifications of the source code and consequently they are valid for a specific language or environment (e.g., J2EE, PHP). Many proposed SQLIA countermeasures analyze the HTTP and/or SQL traffic [19, 23, 24, 30, 97]. However, all these approaches are not available or evaluated in a public Cloud to protect the SaaS applications. Recently, many Cloud IDS

[76,84,86,98] are proposed to allow an IaaS provider to protect the infrastructure IaaS (e.g., VM Monitor) against network attacks (e.g., DoS and DDoS). These Cloud IDS focus on the infrastructure layer without ensuring the security of SaaS applications against SQLIAs, since they do not monitor the input (HTTP) and/or output (SQL) of these applications.

In this paper, we propose a framework, SQLIIDaaS, which detects SQL injection targeting SaaS applications deployed in public Clouds (e.g., AWS). SQLIIDaaS provides the HTTP traffic (e.g., HTTP request, Web session), SQL traffic (e.g., SQL query), and the correlation between HTTP requests and SQL queries. This information is required for the detection of SQLIAs that target SaaS applications without modifying or analyzing their source code. SQLIIDaaS is integrated in AWS, which allows a SaaS provider to reduce maintenance and infrastructure costs by pooling computing resources to serve multiple clients or applications simultaneously (resource pooling). SQLIIDaaS is offered as a set of pre-built Amazon Machine Images (AMI). In this way, a SaaS provider can subscribe to this framework on-demand and the resources of framework (VMs) can be scaled (rapid elasticity) and measured (pay-per-use).

The rest of this paper is organized as follows : Section 4.2 defines the different types of SQLIA. Sections 4.3 presents a brief state of the art on SQLIA countermeasures and Cloud IDS. Section 4.4 describes our proposed HTTP and SQL correlation algorithm. Section 4.5 provides the architecture, integration, and implementation of SQLIIDaaS. Section 4.6 describes the scenarios of test and results of evaluation of SQLIIDaaS. Section 4.7 discusses the advantages and drawbacks of SQLIIDaaS. Section 4.8 presents a short summary and some future works.

4.2 SQL Injection Attacks

An SQLIA is a kind of code injection attack in which a rogue SQL command is crafted by entering special characters into the input fields used in creating dynamic SQL queries [28,61,91]. By using one or more special characters (e.g., `', /, --`) together with key words of the SQL language (e.g., `union, drop`), it is possible to modify the structure and/or the semantics of a dynamically constructed SQL query in such a way that it executes in ways unforeseen by the developer.

We illustrate different types of SQLIA starting from the authentication Web page (e.g., hacmebank.com/LogIn.aspx) of a real-world web services-enabled online banking application, which was built with SALIA vulnerabilities [99]. The following dynamic SQL query is constructed with two input fields (`txtUser` and `txtPassWord`) in the source code of LogIn Web page.

```
‘‘Select * from TblUsers Where User=‘’’ + txtUser.text ‘‘’
and PassW=‘’’ + txtPassWord.text + ‘‘’ ’’
```

The following HTTP request and SQL query will be generated by the application (at run-time), when a user logs into the application :

```
POST /LogIn.aspx HTTP/1.1
userID=admin
password=123
Cookie: ASP.NET_SessionId=xsh...

Select * from TblUsers Where User=‘admin’ and PassW=‘123’
```

We assume that an attacker tries to execute the following classical SQLIA types from the LogIn Web page :

Syntactically Incorrect Queries

The attacker wants to generate an error message from the application. Inputting `’` into `txtUser` and `123` into `txtPassWord`, the following SQL query is generated at run-time :

```
Select * from TblUsers Where User=‘’’ and PassW=‘123’
```

This query gives rise to a meaningful error message (e.g., information about data structure).

Tautologies

The attacker wishes to bypass authentication to access database information. The fields entered create a condition that is always true. For instance, typing `Administrator’ OR 1=1 --` into `txtUser` and anything into `txtPassWord` will generate the following query at run-time :

```
Select * from TblUsers Where User=‘Admin’ OR 1=1 --’ and PassW=‘123’
```

where the condition `1=1` is always true. The remainder of the statement is commented out by the two dashes `--`. Hence, the attacker can access to the application.

Union Queries

A rogue query is combined with an existing query in such a way that results are always returned. For instance, typing `’ UNION select * from TblUsers --` into `txtUser` and `123` into `txtPassWord` will generate the following query at run-time :

```
Select * from TblUsers Where User=‘’
UNION
select * from TblUsers --’ and PassW=‘123’
```

The first select query returns nothing, but the second one returns the users.

PiggyBacked Queries

This attack makes use of the possibility of executing SQL or even system commands on the database server. For instance, typing `’ OR 1=1 ; Drop Table TblUsers --` into `txtUser` and `123` into `txtPassWord` will generate the following query at run-time :

```
Select * from TblUsers Where User=‘’ OR 1=1;Drop Table TblUsers --‘ and PassW=’123’’
```

Hence, the attacker can successfully delete the users by executing `Drop Table` SQL command.

4.3 Related Work

The SQLIA countermeasures can be classified into two categories :

— **Static/dynamic analysis :** AMNESIA [28] analyses the source code of a PHP Web application to generate an automaton for every dynamic query. In the execution phase, if a query does not conform to the corresponding automaton, it will be identified as malicious. SQLCheck [29] and JDBC-checker [95] verify queries at run time to make sure that they conform to models of expected legal queries. SQLGuard [96] and CANDID [31] compare systematically syntactic trees generated for every dynamic SQL queries with and without user input : any discrepancy indicates an attack. These countermeasures require costly analysis and/or modifications of the source code and then they are valid for a specific language (or environment).

— **HTTP/SQL analysis :** Security Gateway [97] and ModSecurity validate the HTTP parameters according to security rules established by developers or signatures of attacks. SQLProb [30] and [23] propose learning-based approaches that validate the user input within the generated query to learn the normal query structure dynamically (at run-time). The HTTP analysis-based techniques [97] require human intervention and maintenance and can produce high level of false alerts. The SQL analysis-based approaches [23, 30] are unable to determine the source of malicious SQL queries (Web session, URL, IP address), since they do not deal with HTTP requests. The authors of [19, 24] propose anomaly-based IDS that combine the analysis of HTTP and SQL traffic in order to reduce the errors and false positive during the detection.

Recently, many Cloud IDS have been proposed [76, 84, 86, 98]. In [84], a NIDS is deployed in different regions of the Cloud to reduce the risk of a distributed denial of service attack (DoS) by exchanging alerts. A security model has been proposed to allow an IaaS provider to protect its infrastructure (e.g., VM Monitor) and the VMs of its clients against the DoS and the installation of illegal tools [98]. The authors of [86] propose a high-level architecture of signature-based, distributed, and collaborative IDS to protect the infrastructure (IaaS). Snort (signature-based NIDS) is installed in AWS to detect unauthorized access and suspicious FTP files [76].

4.4 Correlating SQL queries with HTTP requests

The correlation between HTTP requests and SQL queries is required to determine the HTTP request which generates a given query. Otherwise, some functions should be added in the code to have this information. This correlation allows the SQL analysis approaches to identify the source (e.g., Web session, IP address) of SQLIAs and then realize further prevention measures (e.g., add malicious IP into black list). It is also needed for the approaches that combine the analysis of HTTP requests and SQL queries.

Technically, there is no straightforward and completely accurate way to automatically associate an SQL query to its originating HTTP request using only the limited information available in packet details. Therefore, we developed our own correlation method according on two types of information : 1) the similarity between literals in SQL queries and parameters in HTTP requests ; 2) the time relationship between HTTP requests and SQL queries. The similarity of character strings (parameters and literals) is evaluated using string distance computation algorithm [100] and the time relationship is enforced by a simple time comparison.

Distance Score

Before explaining the distance score, we introduce two definitions :

1. A *Param* is an input parameter (e.g., input fields, cookies) of an HTTP request.
2. A *literal* is any part of an SQL statement that is neither an identifier (key word, table name, field name, ...) nor an operator of the SQL language.

Consider that an HTTP=$\{Param_1, \ldots, Param_n\}$ and an SQL=$\{Literal_1, \ldots, Literal_m\}$. For example, we have HTTP={admin,123} and an SQL={admin,123}, when logs through LogIn.aspx URL. In typical programming, one single parameter from the HTTP is used for

generating each literal in a dynamic SQL. We thus compute the minimal string distance between each parameter $Param_i$ of an HTTP and all the literals of the SQL. The distance score between an HTTP and an SQL is evaluated as the average, over all the parameters of the HTTP, of the minimum distances :

$$Score(HTTP, SQL) = \frac{1}{n}\sum_{i=1}^{n}\min_{j=1}^{m} Dist(Param_i, Litteral_j)$$

where n is the number of HTTP parameters, m is the number of SQL literals.
$Dist(Param_i, Litteral_j)$ is the edit distance between $Param_i$ and $Litteral_j$, which is basically the number of basic edit operations needed to transform one string ($Param_i$) into the other ($Litteral_j$). We selected this measure as it applies to strings of different lengths and is consistent with transformations that may be applied in producing a dynamic query from an HTTP (e.g., concatenating strings, extracting sub-strings).

In order to enforce the causality condition between HTTPs and SQLs, we limit the distance evaluations to pairs (HTTP,SQL) such that

$$\text{timestamp}_{SQL} > \text{timestamp}_{HTTP} \text{ AND } \text{timestamp}_{SQL} < \text{timestamp}_{HTTP} + (2 * t)$$

where t provides a time window to account for the time required by the SaaS application for generating SQL queries for each HTTP request. This value is multiplied by 2 to ensure that all possible (candidate) HTTP requests are covered. The value of t is 1 second at the beginning of the execution. In order to reduce the number of scores to calculate, it will be the average of ($Timestamp_{SQL} - Timestamp_{HTTP}$) calculated after the correlation of each (HTTP, SQL) pair.

Selecting an HTTP Request

Among the HTTP requests compared, the HTTP request that obtains the lowest score is considered as the generator of the SQL query, provided that its score is below a pre-established acceptance threshold. The threshold is set relative to the average size of the HTTP parameters :

$$Threshold(HTTP) = \frac{r_{th}}{n}\sum_{i=1}^{n}(Param_i.length)$$

where $Param_i.length$ is the size of parameter i. The coefficient r_{th} must be adjusted empirically. After a number of experiments using different values for r_{th}, it was found that a value of $r_{th} = 0.6$ provides the best overall results.

4.5 SQLIIDaaS : SQL injection intrusion detection framework as a service for SaaS providers

This section presents the architecture, integration and implementation of SQLIIDaaS.

4.5.1 Architecture

As Shown in Fig. 4.1, the architecture of SQLIIDaaS consists of six main components to detect SQLIAs that target SaaS applications.

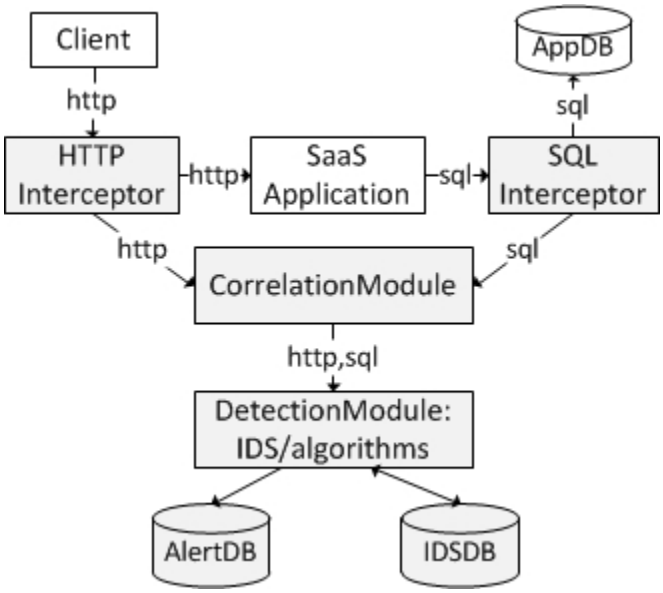

Figure 4.1 Architecture of SQLIIDaaS

HTTPInterceptor

Monitors HTTP traffic that targets the application and checks the parameters for each HTTP request. If it contains at least one parameter, relevant information including URL, parameters' list, Timestamp and Web session identifier will be extracted and sent to CorrelationModule.

SQLInterceptor

Monitors SQL traffic generated by the application in response to the HTTP requests. It sends relevant information including SQL statement, Timestamp to CorrelationModule.

CorrelationModule

Receives the details of HTTP requests and SQL queries. Then, it matches each SQL query into HTTP request by executing the correlation algorithm (see section 4.4).

DetectionModule

Analyses the HTTP and/or SQL to detect SQLIAs. For instance, we added an existed SQL injection anomaly-based IDS [19] that works as follows :

— The learning phase aims at creating the models of normal SQL queries originated from HTTP requests in which no input parameter contains character other than a letter or a number. The appropriate model (template) is generated by replacing all literals with the token character (?).

— In the detection phase, if the template of a given SQL query is not modelled during the learning phase, this SQL is considered as a SQLIA.

IDSDB/AlertDB

DetectionModule stores the normal behavior (e.g., templates of normal SQL queries) or signatures of attacks in IDSDB. It stores the relevant information about detected attacks (e.g., SQL statement, IP source and date/time of the attack) in AlertDB.

An illustrative example : Let us consider a user who tries to authenticate via LogIn URL (the HTTP and SQL are illustrated in the section 4.2). The HTTPIntercepor and SQLInterceptor send the following HTTP and SQL to the CorrelationModule :

HTTP=(LogIn.aspx,(admin,123),WebSession,$Timestamp_{HTTP}$)

SQL=(Select * from TblUsers Where User='admin' and PassW='123',$Timestamp_{SQL}$)

The CorrelationModule calculates the score(HTTP,SQL) which is equal to 0. Thus, it matches and sends the HTTP and SQL to the DetectioModule which works as follows :

— The learning phase associates the following template to LogIn.aspx URL (in IDSDB) :
`select * from TblUsers Where User='?' and PassW='?'`

— The detection phase considers the SQL queries of section 4.2 as attacks, since their models (shown below) are not modelled for LogIn URL.

1. `Select * from TblUsers Where User='?' OR ?=? --' and PassW='?'`
2. `Select * from TblUsers Where User=''' and PassW='?'`
3. `Select * from TblUsers Where User=''`
   `UNION`

```
Select * from TblUsers --' and PassW='?'
```

4. ```
   Select * from TblUsers Where User='' OR ?=?;
   Drop Table TblUsers --' and PassW='?''
   ```

4.5.2 Integration in AWS

The integration of our framework (see Fig. 4.2) in AWS consists of a virtual private Cloud (VPC) with a public subnet (IDSSubNet) and two private subnets (AppSubNet and DBSubNet). The WebAppVM1 and WebAppVM2 VMs are launched in the AppSubNet to install IIS Web servers and service-oriented Web applications. The AppDBVM VM is launched in the DBSubNet to install the MSSQL server and the database (App_DB) used by the applications. For instance, a SaaS provider can deploy these applications on WebAppVM1 and WebAppVM2 for two different clients (or tenants). However, he can launch additional WebAppVM and AppDBVM (on-demand) in the AppSubNet and DBSubNet to deploy other Web applications. The NATVM, IDSVM and IDSDBVM VMs are launched in IDSSubNet to install the components of our framework. An internet gateway (igw) is associated to the VPC to enable the communication with the internet. Table 6.1 shows the specifications of all VMs.

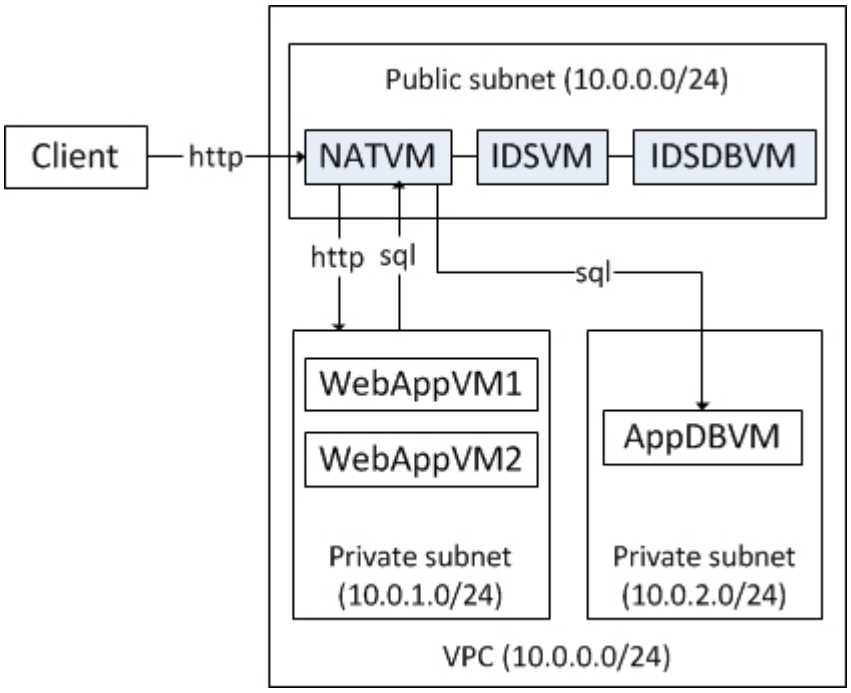

Figure 4.2 Integration of SQLIIDaaS in AWS
   ```

Table 4.1 Specifications of different VMs

Virtual machine	Subnet	Address IP	Instance type	Installation
WebAppVM1	10.0.1.0 Private	10.0.1.236	t2.small, 1vCPU, 2GiB RAM	IIS Web Server Web Application Web Services
WebAppVM2	10.0.1.0 Private	10.0.1.8	t2.small, 1vCPU, 2GiB RAM	IIS Web Server Web Application Web Services
AppDBVM	10.0.2.0 Private	10.0.2.5	t2.micro, 1vCPU, 1GiB RAM	MSSQL/App_DB
NATVM	10.0.0.0 Public	10.0.0.119 10.0.0.254	t2.medium, 2vCPU, 4GiB RAM	WinPcap, .Net-FrameWork, NATApp
IDSVM	10.0.0.0 Public	10.0.2.236	t2.small, 1vCPU, 2GiB RAM	.NetFrameWork IDSApp
IDSDBVM	10.0.0.0 Public	10.0.0.220	t2.micro, 1vCPU, 1GiB RAM	MSSQL/IDS-DB
AttackVM	11.0.0.0 Public	11.0.0.1	c4.2xlarge, 8vCPU, 15GiB RAM	Python/SQLMap

NATVM configuration : The NATVM is configured as a NAT (Network Address Translation) with RRAS (Routing and Remote Access) to enable communication between the VPC and the outside (Internet) via two elastic (public) IP addresses (e.g., 52.25.139.95 and 52.37.39.100) that should be associated with two private IP addresses (e.g.,10.0.0.119 and 10.0.0.254). Thus, the NATVM can send and receive HTTP traffic from WebAppVM1 and WebAppVM2 (launched in AppSubNet) via an interface network (WebAppNet). A routing table should be configured and attached to each subnet in the AWS. For instance, a DBSubNetRT is attached to a DBSubNet to route all traffic destined for the AppDBVM over the AppDBNet network interface of the NATVM. As a result, HTTP and SQL traffic can be captured on the NATVM without modifying the source code or installing additional tools/-programs (e.g.,Winpcap) on the WebAppVM1 and WebAppVM2.

SQLIIDaaS modules installation : In order to avoid having NATVM as a single point of failure and prevent NATVM from breaking, we only installed the HTTPInterceptor and SQLInterceptor on NATVM to capture and send HTTP and SQL traffic to IDSVM. In order to support HTTPS (for SSL) protocol, the HTTPInterceptor can be deployed on each WebAppVM where the HTTP traffic is decrypted. If the file of security keys are sent to WireShark/winpcap (installed on the NATVM), HTTP traffic can also be decrypted and analyzed by HTTPInterceptor on NATVM. The CorrelationModule and DetectionModule

are installed on IDSVM to receive, correlate and analyze the HTTP/SQL traffic. The IDSDB and AletrDB are installed on IDSDBVM.

Security groups configuration : In order to provide an additional layer of security and limit the attack surface, a security group (SG) is configured and associated with each VM (AppDBVM, NATVM, IDSVM and IDSDBVM) or set of VMs (WebAppVM1/WebAppVM2). The NATVM is the only node with public IP addresses and a SG open to the public. The WebAppVM1/WebappVM2 SG is configured to only send to and receive from the NATVM node. The IDSVM SG is configured to only receive from the NATVM in order to protect the IDS.

4.5.3 Implementation

A proof of concept of SQLIIDaaS has been implemented (in C#) as two applications NATApp and IDSApp that should be installed on NATVM and IDSVM (see Fig. 4.3). These applications communicate in asynchronous mode through a .NET message queue service (DotNetMQ) [101] that should be installed on the NATVM and IDSVM.

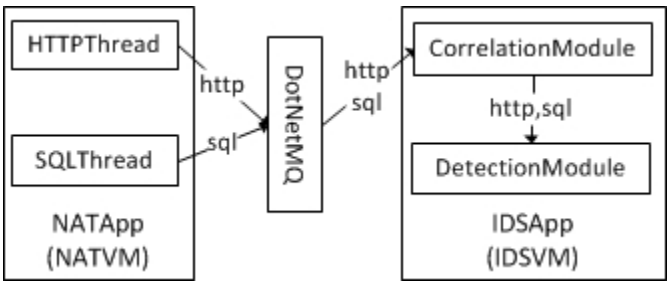

Figure 4.3 Implementation of SQLIIDaaS

The NAT application (NATApp) implements the HTTPInterceptor and SQLInterceptor modules as two threads, HTTPThread and SQLThread, to simultaneously capture HTTP and SQL traffic by using the Winpcap library [102]. The HTTPThread and SQLThread listen to traffic, respectively, on the TCP/HTTP port specific to the HTTP (e.g., port 80) and to the database (e.g., port 1443 for MS SQL) protocols. Then, HTTP and SQL details are extracted and sent to the IDS application (IDSApp).

The IDS application (IDSApp) implements CorrelationModule and DetectionModule. The CorrelationModule receives HTTP/SQL details and associates the SQL with the corresponding HTTP. In the learning phase, the DetectionModule deals with the analysis of the HTTP/SQL to generate and store the templates of normal SQL query for each URL in IDSDB

(installed on the IDSDBVM). In the detection phase, it detects SQLIAs by communicating with IDSDB and stores the alerts in AlertDB (installed on the IDSDBVM).

4.6 Scenarios of test and evaluation

Generally, an IDS can be evaluated by generating traffic from the signatures (e.g., of another IDS) or by using vulnerability exploitation tools to target a system with attacks [94]. We adopt the vulnerability exploitation tool approach to be able to evaluate the reaction of all VMs under attacks in real time, and not just the detection quality. Our test environment is implemented as follows :

Vulnerable applications : Hacmebank [99] is a real-world web services-enabled online banking application, which was built with SQLIA and developed by using ASP.NET and MSSQL. An instance of this application is deployed on the IIS Web server of each WebAppVM (WebAppVM1 and WebAppVM2). The databases of applications are installed on the MSSQL server of AppDBVM.

Vulnerability exploitation tool : An Attack VM (AttackVM) is launched in a VPC1 (different than VPC) with high performance (see Table 6.1) to send a big quantity of HTTPs during a small period of time. Note that the AttackVM can be outside of AWS. We used SQLMap [103], an open source penetration testing tool that automates the process of exploiting SQL injection flaws, as SQLIA exploitation tool. SQLMap is installed on the AttackVM to target the two applications with a large quantity of malicious (with SQLIA) HTTP POST requests. A batch file is created with a list of Python SQLMap commands for each application to automate and stress the VMs (WebAppVM1, WebAppVM2, NATVM, IDSVM and IDSDBVM) and two applications by increasing the amount of attacks.

In order to guarantee the same quantity and frequency of attacks, the same previous batch file is executed, from AttackVM, during three scenarios :

1. **Scenario 1 :** The WebAppVM1 and WebAppVM2 communicate with AppDbVM directly without routing traffic through NATVM. The goal of such scenario is to measure the HTTP response time and consumption of resources on VMs without adding SQLIIDaaS.
2. **Scenario 2 :** The HTTP/SQL traffic are routed through NATVM, but without starting IDSVM and IDSDBM (DotNetMQ is disabled). The HTTP response time during this scenario is compared with that of the previous one in order to evaluate the integration's impact of SQLIIDaaS.
3. **Scenario 3 :** The HTTP/SQL traffic are routed through NATVM. IDSVM/IDSDBM

are started, and DotNetMQ, NATApp, and IDSApp are enabled. The consumption of resources (CPU and memory) during scenario 2 and 3 are compared in order to evaluate the overhead of SQLIIDaaS.

4.6.1 Detection quality

The detection quality was evaluated by metrics including false positive (FP), false negative (FN), true positive (TP), detection rate (DR), and precision (PR) [94]. In the context of our evaluation, we define these metrics as follows :

1. TP (attack, alert) : The number of detected queries whose source is the IP address of the AttackVM that runs the SQLMap, which generates only malicious HTTPs (with SQLIA).
2. FN (attack, no alert) : The number of undetected queries (not existing in the table of alerts) whose source is the IP address of AttackVM. We determine this value by forming a relationship between the alerts table and the table of all SQL queries.
3. FP (no attack, alert) : The number of detected queries (in the table of alerts) whose source is the IP address generated by LoadImpact tool [104], which we configured to generate only benign HTTPs (input values without critical characters).
4. The detection rate (DR) and precision (PR) are TP/ (TP + FN) and TP/ (TP+FP) respectively.

The attack phase takes about 16 minutes to generate an 45663 HTTP POST with SQLIA. The TP, FN, FP, DR, and PR values are 44456, 1207, 0, 0.9735, and 1, respectively. According to these values, the true positive rate is acceptable (0.9735), since SQLMap uses other attack tools and evasion techniques. The detection rates of Amnesia [28] and SQLProb [30] are 1 and between 0.98 and 1 (depending on web applications) respectively .

4.6.2 HTTP response time

We used Wireshark and LoadImpact [104] tools to evaluate the impact of adding SQLIIDaaS and routing traffic through NATVM on HTTP response time. Table 4.2 shows the average values of the execution of these tools during the previous three scenarios :

1. Wireshark is used to collect packets that contain the HTTP responses of malicious HTTP requests (sent by SQLMap) : the average of response time values (based on time request since field) are 116.45, 120.39, and 120.91 milliseconds in scenario 1, scenario 2, and scenario 3, respectively.

2. LoadImpact, which generates a massive amount of HTTP traffic by simulating virtual users who are trying to load the web pages at the same time, is used to measure the performance of Web applications by sending a number of HTTP requests in 5 minutes : the average values (of 5 executions) of this number are 31485, 27968, and 26738 in scenario 1, scenario 2, and scenario 3, respectively.

Table 4.2 Impact evaluation of adding SQLIIDaaS on HTTP response time

Scenarios	Wireshark (ms)	LoadImpact (5 minutes)
(1)no traffic routing, no SQLIIDaaS	116.45	31485 HTTPs
(2)traffic routing,no SQLIIDaaS	120.39	27968 HTTPs
(3)traffic routing, SQLIIDaaS	120.91	26738 HTTPs

According to these results, the integration of SQLIIDaaS (from first to third scenarios) increases the HTTP response time about 5 ms and decreases to 4747 the number of HTTP requests executed by LoadImpact during 5 minutes. We obtained a low impact from the second and third scenarios, because NATApp and DotNetMQ consume few resources of NATVM.

4.6.3 Overhead

In this section, we measure CPU and memory usage on the VMs of SQLIIDaaS.

NATVM : Fig. 4.4.a demonstrates that the highest CPU and memory usage on NATVM are 2.46% (y-axis, top-left) and 17.5% (y-axis,bottom-left), respectively, during the period of attacks (for scenario 2). Thus, the traffic routing introduces an additional usage of 2.12% CPU and 0.2% memory (0.34% CPU and 17.3% memory are used before the start of attacks). Fig. 4.4.b demonstrates that the highest CPU and memory usage on NATVM are 6.86% (y-axis, top-right) and 22.1% (y-axis, bottom-right), respectively, during the period of attacks (for scenario 3). Thus, the traffic routing and the activation of NATApp/DotNetMQ introduce an additional usage of 5.61% CPU and 0.4% memory to route, capture, extract, and send the HTTP and SQL. This overhead is acceptable (for scenario 2 and scenario 3) considering the massive amount of HTTP requests (45663) during a short time (16 minutes).

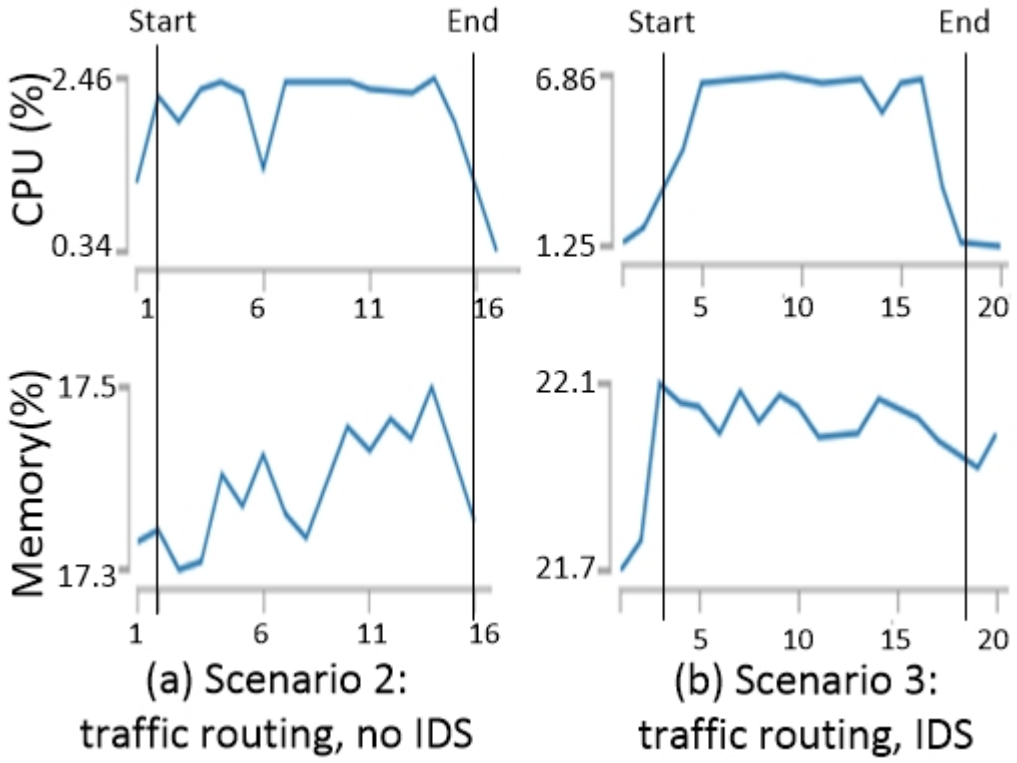

Figure 4.4 CPU/memory usage on NATVM. x-axis corresponds to a time period of attacks (16 minutes).

IDSVM and IDSDBVM : Fig. 4.5 demonstrate the percentages of CPU/memory usage on IDSVM/IDSDBVM in scenario 3 (SQLIIDaaS is enabled). For IDSVM, the highest CPU and memory usage are 17.7% (y-axis, top-left) and 55% (y-axis, top-right). Thus, the activation of IDSApp/DotNetMQ introduce an additional use of 12% CPU and 4% memory to receive and analyze HTTP/SQL traffic (5.08% CPU and 51.1% memory are used before the start of attacks). This overhead is acceptable to detect a big quantity of attacks during 16 minutes. For IDSDBVM, the additional averages of usage are about 5% CPU (y-axis, bottom-left) and 1% memory (y-axis, bottom-right) from the beginning of the period of attacks. This overhead is very low, because only two stored procedures are executed on IDSDBVM to validate the presence of templates in normal behavior and add the information of attacks.

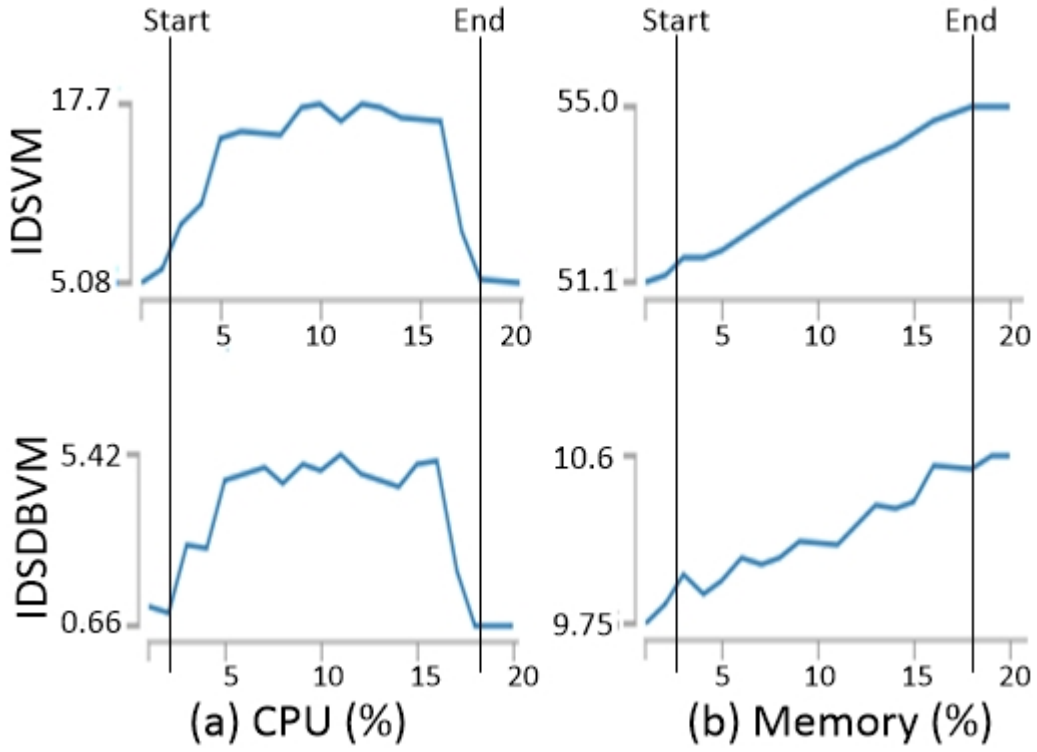

Figure 4.5 CPU/memory usage on IDSVM and IDSDBVM. x-axis corresponds to a time period of attacks (16 minutes).

4.7 Discussion

Our proposed solution framework is compatible with the most of Cloud's characteristics (On-Demand Self-Service, Resource Pooling, Rapid Elasticity and Measured Service) and then offered as a service from SaaS providers' point of view :

— **On-demand self-service :** SQLIIDaaS is offered as pre-built Amazon Machine Images (AMI) with all necessary tools and modules. A SaaS provider can subscribe (with its account number) to our service, via web interface, to create its own VMs (IDSVM, IDSDBVM) from these AMIs in a simpler and faster way. The configuration of NATVM, routing tables, and security groups can be automated by calling AWS API.

— **Resource pooling :** The computing resources of the framework (NATVM, IDSVM, and IDSDBVM) are pooled to serve multiple clients/applications simultaneously.

— **Rapid elasticity :** SQLIIDaaS is provided as a scalable service, since it is available as a set of VMs. For example, a SaaS provider can scale horizontally (e.g., create cluster

of IDSVM) or vertically (e.g., increase/decrease the size of VMs) the capacity of VMs according to demand (e.g., amount of traffic).

— **Measured service :** The amount of resources that SQLIIDaaS uses can be monitored, measured and controlled by using AWS tools (e.g., CloudWatch). The cost of SQLIIDaaS can be evaluated based on the size and time usage of NATVM, IDSVM and IDSDBVM.

Additionally, SQLIIDaaS has a high level of portability, openness and flexibility :

— **Portability :** First, SQLIIDaaS operates at the level of the HTTP/SQL protocols, independently of the programming language of application (considered as a black box). Second, it can be deployed in any another Cloud environments that supports VPC with minimal modifications. Third, A SaaS provider does not need to modify or analyze applications' code.

— **Openness/flexibility :** The HTTP/SQL analysis-based approaches can be implemented within in SQLIIDaaS in easy and flexible way : its modules can be enabled and disabled depending on SQLIA countermeasures to be integrated. For example, the CorrelationModule can be disabled, if an approaches does not need the correlation between HTTP and SQL. SQLIIDaaS enables SQL analysis-based approaches to determine the sources of attacks (Web sessions or IP addresses). Additionally, many SQLIA countermeasures can be combined together within SQLIIDaaS to improve the quality of detection.

Despite its advantages, SQLIIDaaS has some drawbacks. The integration of SQLIIDaaS increases HTTP responses because the traffic should be routed through the NATVM. A SaaS provider should do some configurations on the NATVM. The size of the NATVM should be at least medium to support three network interfaces (publicNet, WebAppNet, and AppDBNet). However, there is no additional cost of adding a network interface in AWS.

4.8 Conclusion

This paper proposes a SQLIA intrusion detection framework solution that can be deployed and used by SaaS providers in a service-based manner with high level of portability. A SaaS provider can easily subscribe with this framework to launch a set of VMs (from our framework's pre-build VMs) that can be associated on multiple SaaS applications without analyzing or modifying their source code. Thus, many Cloud properties can be provided such as on-demand self-service, resource pooling, rapid elasticity and measured service. The quality of detection is evaluated based on an existing SQLIA IDS. However, a SaaS provider can add security components or other IDS (on IDSVM) to improve the security of its applications

(deployed in public Clouds).

Research is ongoing to implement and compare several SQLIA IDS/algorithms (that analyse HTTP and/or SQL traffic) within the proposed framework. Interesting future work would be to make SQLIIDaaS fully as a service from tenant's point of view by ensuring on-demand self-service and pay-as-you-go characteristics.

CHAPITRE 5 ARTICLE 2 : ITADP : AN INTER-TENANT ATTACK DETECTION AND PREVENTION FRAMEWORK FOR MULTI-TENANT SAAS

Mohamed Yassin, Chamseddine Talhi and Hanifa Boucheneb
Revised and submitted to : Journal of Information Security and Applications

Abstract Software-as-a-service (SaaS) is a service-oriented Web application running on a Cloud environment. With the multi-tenancy, the SaaS provider can largely reduce the cost of resources and maintenance by sharing the application and database instances between its tenants (clients). This multi-tenancy affects the security of tenants, specifically, when several tenants use the same tables of a single database. Indeed, an important consequence of this full multi-tenancy is that a malicious tenant user can view or modify the rows of other tenants. Consequently, the detection and prevention of attacks among tenants is a key security requirement that should be addressed by the provider. In this sense, this paper proposes an inter-tenant attack detection and prevention framework, based on SQL syntactic analysis, for multi-tenant SaaS. This framework is integrated in Amazon Web Services (AWS) public Cloud and meets accuracy, portability, compatibility, and ease of integration requirements. The experimental results show that the framework works with small overhead on the virtual machines and minimal impact on the HTTP response time.

5.1 Introduction

Cloud Computing (CC) is a new paradigm that allows a provider to deliver infrastructure as a service (IaaS), platform as a service (PaaS), and software as service (SaaS) with on-demand, rapid elasticity and pay-per-use properties for its tenants [1, 2]. A SaaS refers to a service-oriented Web application running on a Cloud environment (IaaS or PaaS) and offered by a provider for several tenants [1,2,105,106]. The SaaS providers and tenants benefit from these applications. Indeed, the providers reduce the resources, operational, and maintenance costs by sharing the application and database servers among several tenants [2,40,67,68,105,106]. Each tenant has its own domain name (e.g., tenant.SaaSApp.com) without managing underlying Cloud resources. The end-users of tenants access these applications from various devices. In a typical multi-tenant SaaS, some tables of a single database are shared by several tenants. This full multi-tenancy guarantees a high cost-effectiveness since dedicating a database to each tenant is expensive for SaaS providers. However, the confidentiality, inte-

grity, availability, and accountability of tenants' data are still a relevant and specific security issues for multi-tenant SaaS (MTSaaS) [4, 21, 22, 34, 37, 39, 40, 42, 44, 45, 47, 48]. An important consequence of this multi-tenancy is that a malicious tenant user can view or modify the data of other tenants, for example, by executing SQL injection attacks via the Web pages of its tenant. Thus, a fundamental challenge for a SaaS provider is to protect the data of each tenant from other co-located tenants and to identify the malicious tenant users. This data access protection achieves tenant isolation, which is crucial from tenant's perspective.

Furthermore, from the tenant's point of view, the application and database must be used, as they are installed on its own infrastructure. In other words, a tenant is not responsible for attacks coming from co-located tenants. Consequently, the SaaS provider should prevent a given malicious tenant user from accessing the data of other tenants via the domain of its tenant [21, 42, 45, 48]. However, there is no automatic solution to detect or prevent these new attacks caused by multi-tenancy. According to Microsoft, "the developer needs to ensure that no tenant has unwanted access to other tenants' data" [67].

In order to improve the security of MTSaaS, encryption [49, 65], design patterns [67, 68], and tenant isolation [40, 64, 69] approaches have been proposed. The encryption [49, 65] ensures only the confidentiality of data by enabling each tenant to encrypt its data with its own key. The design patterns, [67, 68] which consist to return or modify only the data of current tenant (executor of HTTP request), can be applied by developers during the development (or migration) of the application. However, a malicious tenant user can bypass these design patterns at run-time even if they are implemented correctly. Accidentally, it can access the data of other tenants if the developers do not apply these patterns correctly. Only a few research papers have dealt with the tenants' data isolation in a multi-tenant relational database [40, 69]. In [40], two data placement algorithms are proposed in order to avoid having (as much as possible) the data of competing tenants with the same business type BT (e.g., Health-care, Financial) and size (e.g., small, medium, enterprise) in the same database. However, these algorithms have failed to find such a database, and, thus, several competing tenants may reside in the same database. In [69], a HTTP proxy needed to be installed in the infrastructure of each tenant in order to sign its HTTP requests with its own key. Therefore, this proposition does not detect the unauthorized access of the tenants' data since they can pass through the HTTP requests even if the SSL protocol is used.

In order to overcome the limitations of previous approaches, we propose a framework to detect and prevent automatically unauthorized data access (accidental or malicious) across tenants during the execution of SQL queries. This framework gives SaaS providers accountability concerning the tenants' data. This accountability is crucial, especially for full MTSaaS, where

multiple users, who could be malicious, of competing tenants simultaneously access the shared tables of a single database. For example, it enables the SaaS provider to identify the tenant that represents the highest risk and/or the malicious tenant users who execute attacks against other tenants.

Main contributions :

— A dynamic inter-tenant attack detection and prevention framework, based on SQL syntactic analysis, is proposed for full MTSaaS. This framework ensures tenants' data isolation in terms of security during the execution of SQL queries without modifying the back-end database.

— Two integrations (detection and prevention) of the previous framework are deployed and evaluated (quantitatively and qualitatively) in a real public Cloud (AWS) based on two multi-tenant SaaS.

The rest of paper is organized as follows. Section 5.2 describes the architecture of a typical MTSaaS and the motivation of this research. Section 5.3 presents a literature review on encryption, design patterns, tenant isolation and Cloud IDS. Section 5.4 describes the proposed inter-tenant attack detection and prevention (ITADP). Section 5.5 presents the implementation and evaluation of ITADP. Section 5.6 discusses the proposed ITADP. Finally, Section 5.7 outlines the conclusions and future research implications.

5.2 Background, context and motivation

A SaaS refers to a service-oriented Web application that a provider hosts in a Cloud environment to serve its tenants [1, 2, 107]. A SaaS is multi-tenant when several tenants share one or more application/database instances [40, 105, 106, 108]. Thus, each tenant leases a specific domain and pays according to product edition or size (e.g., standard, professional and enterprise) and/or user accounts [2, 40]. Consider, for example, the SaaS provider that hosts an online store application to serve hundreds, even thousands of tenants [108].

5.2.1 SaaS architecture

A SaaS is a Web application which is divided into presentation, business and data access layers [109]. The presentation layer implements the Web pages (e.g., HTML, HTML5, JSON, CSS) to receive the HTTP requests from users. The business logic layer (e.g., Java EE, ASP.NET) manages the business rules and communication between the presentation and data access layers. The data access layer focuses on the access to databases using Structured

Query Language (SQL). This multi-layer separation facilitates the assigning tasks to developers, the development and maintenance of application. For example, only the user interface (presentation layer) should be re-implemented to allow the access on existing application from mobile devices.

A SaaS is deployed on N physical (virtual) tiers (N-tier) [109]. Each tier consists of a separate IaaS (e.g., VMs, virtual networks, load balancer) to make the application more scalable, resilient and secure. For example, many security mechanisms can be added at each tier which can be scaled separately and works even if some VMs fail. A typical SaaS consists of Web tier (front end), middle tier (also named business or application tier) and data tier (back-end). The middle tier is optional but recommended. For example, in two-tier architecture (client-server), the Web tier contains the presentation and business/data logic and the data tier contains the databases. This complicates the load balancing and increases the charge on server.

Figure 5.1 illustrates a three-tier architecture of multi-tenant SaaS (MTSaaS), based on Simple Object Access Protocol (SOAP) or Representational State Transfer (REST) Web services, deployed in a public Cloud [2, 40, 105, 106, 109, 110]. The Web tier as well as middle tier consists of a load balancer (HTTPLB or INTLB) and several VMs (IaaS). In Web tier, a Web server (PaaS) is installed on each VM (IaaS) to host an instance of Web application (Web pages). The middle tier consists of one or many VMs that exposes Web services (SOAP or REST). The data tier consists of one or more databases managed by a database management system (DBMS) such as SQL Azure. The three-tier architecture is adopted by the most of SaaS providers to be able to expose their Web services for third parties.

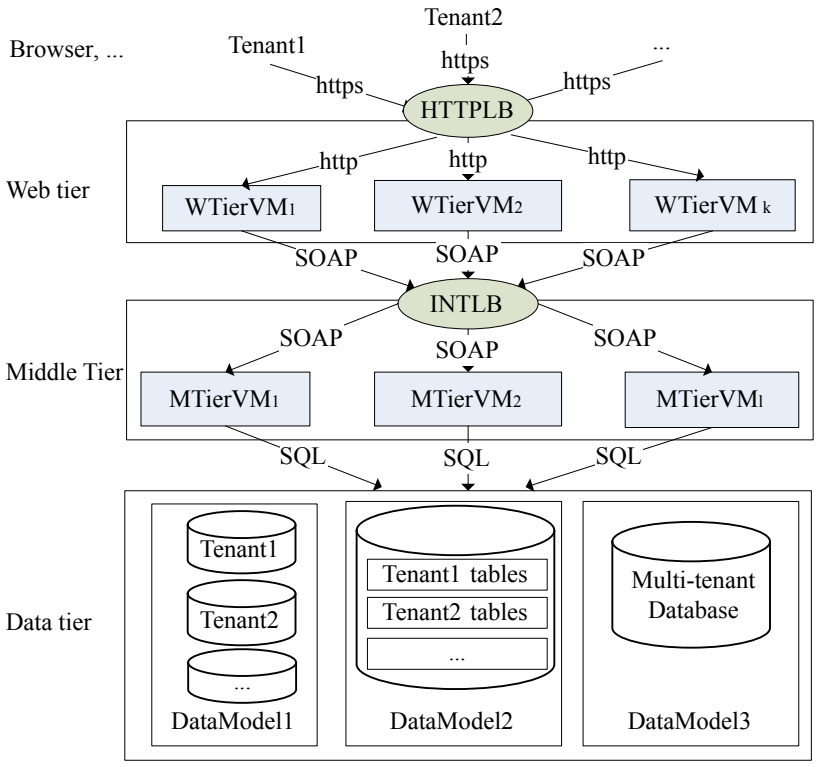

Figure 5.1 Three-tier architecture of service-oriented multi-tenant SaaS (MTSaaS)

Three data models (see Figure 5.1) have been proposed to store the data of several tenants of MTSaaS [40, 67, 105, 106, 110] :

— **Data model 1 :** A database is dedicated to each tenant. This model is very similar to the on-premises deployment, but the databases are installed on one or more VMs in the Cloud. In this model, the tenant isolation and security is high. Furthermore, it is difficult to attack the databases of many tenants through the domain of a tenant since the users of a tenant connect to its database through a different connection. However, the cost of the infrastructure and maintenance is also high because the SaaS providers should locate and manage several databases.

— **Data model 2 :** Several tenants share a single database with a set of tables (schema) for each tenant. For example, the tables Orders1, Orders2 and Orders3 of the database belong to tenant1, tenant2 and tenant3, respectively. A SaaS provider can reduce the cost with this model. However, the management of connection and data access is a complex task since each tenant has its own tables. Let us imagine the complexity of the management of privileges if we have 100 tenants with a set of tables for each of them. Additionally, a set of tables must be created for each new tenant.

— **Data model 3 :** Several tenants share some tables of a single database. In order to

distinguish between tenants, an attribute (e.g., TenantID) should be added to each shared table. This model ensures a high level of cost-effectiveness and a generic implementation of data access according to TenantID column. However, the data isolation among co-located and competing tenants in terms of security is not a trivial task because it should be achieved by the developers at rows level of the shared tables.

In this paper, we assume that a single database is dedicated for several tenants (Data model 3). This model reflects the multi-tenancy and ensures the cost-effectiveness, but requires particular attention regarding tenants' data security more than the other two models.

5.2.2 Motivation

In SaaS model, the multi-tenancy poses new security risks related to the data and privacy of the tenants that share the same application and database [4, 21, 22, 34, 37, 39, 40, 42, 44, 45, 47–49]. This multi-tenancy is identified as the most important security problem of SaaS [39]. There is a high potential of intrusion tenants' data and a malicious tenant user can bypass security checks in order to access sensitive data of other tenants [4, 34]. According to Cloud Security Alliance CSA [37], the multi-tenancy creates new targets for intrusion and presents critical and complicated security implications for provider since tenant can gain access to other tenants' data (deployed in a public Cloud).

Consequently, a SaaS provider must ensure that a tenant does not represent any risk to the data or privacy of other tenants [42]. In full multi-tenancy model, the responsibility of SaaS provider is to prevent inter-tenant attacks (SQL injection, command injection, insecure direct object references, and cross-site scripting) [45] and data leakage among tenants [49], since the data of multiple tenants is often stored in plain text [45, 66] in the same database. The protection of tenants' data must be ensured at application/database level [4, 34, 48]. Additionally, it should be applied not only on tenant's data at rest or in transit, but also in process since a malicious tenant can perform attacks against other tenants during the processing of its data [21, 47]. To detect and fix the vulnerabilities of MTSaaS, there is a need to new penetration testing tools to validate that a tenant cannot compromise the data of other tenants [4, 34, 37].

In this paper, we focus on the detection and prevention of attacks among the tenants of MTSaaS. The multi-tenancy is at the origin of these new attacks which can occur accidentally or maliciously [37,40,42,44,49,68,111]. Accidentally, the data of other tenants can be returned during the execution of the requests of an un-malicious tenant user due to programming errors, bugs or malfunction [37, 40, 49]. Maliciously, a tenant user can access the data of other tenants by executing SQL injection (SQLIA), Cross-site scripting (XSS), or Cross Site

Request Forgery (CSRF) attacks [40, 44, 68, 111]. To illustrate these attacks, assume that the following SQL query is written in the source code of Orders Web service :

```
strQuery="select * from Orders where TenantID=" + pTenantID
```

Normally, a tenant1 user (u1tenant1), who is authenticated with its user name and password, can retrieve all tenant1's orders as follows :

- The following GET HTTP request is sent to the application :
  GET http ://www.tenant1.SaaSApp.com/Orders.aspx ?pTenantID=1
- The application sends the following SOAP request to WSOrder Web service :
  GET /WSOrder.asmx HTTP/1.1 <sBody><GetOrders>
  <pTenantID>1</pTenantID ></ GetOrders ></sBody >
- The WSOrder executes the GetOrders function that sends the SQL query (`select * from Orders where TenantID=1`) to database server.

A threat model (see Figure 5.2) shows the attacks that can target the tenants' data of MTSaaS. These attacks can be classified according to the source (tenant domain) and target (tenant rows) as follows :

- **Classical :** These attacks are performed by a user against the data of other users of the same tenant. These attacks can occur even if the application and database of each tenant are completely isolated in its own infrastructure (on-premise).
- **One-to-one :** These attacks are performed by a tenant's user from the domain of its tenant (tenant1) in order to access the rows of another tenant (tenant2). A concrete example is when a tenant1's user (u1tenant1) modifies the pTenantID parameter to 2 to generate the following SQL query (at run-time) [44, 68] :
  `Select * from Orders where TenantID=2`
  The previous query allows the u1tenant1 to read the orders of tenant2. The u1tenant1 can also modify the pTenantID if it is stored in the cookies or in Web session variables, since they are created by the server in the client side to allow the application/server to identify the sessions/tenants.
- **One-to-many :** These attacks are similar to the previous ones. The only difference is that an user accesses the rows of several tenants (tenant1 and tenant3) using the domain of its tenant (tenant2). A concrete example is when a malicious user injects `OR 1=1` in an input fields of Web page [44, 68, 111] in order to generate the following SQL query :
  `select * from Orders where TenantID=1 OR 1=1`
  This query allows the u1tenant1 to read the orders of other tenants without injecting special character (`e.g., -, '`) used in classical SQL injection attacks. Accidentally, if the programmer does not filter correctly on TenantID (e.g., `"Select * from`

`Orders"` instead of `"select * from Orders where TenantID=" + pTenantID)`, the u1tenant1 can retrieve the orders of other tenants.

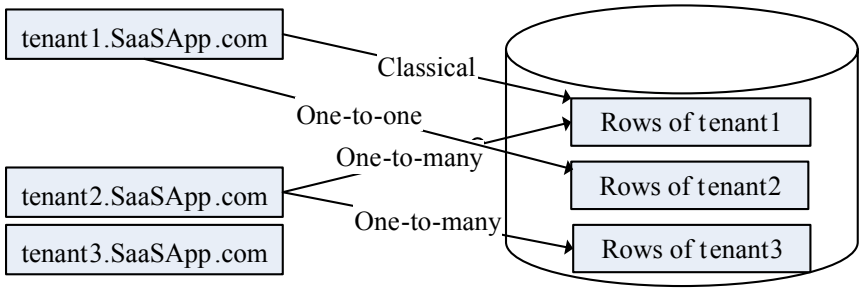

Figure 5.2 Threat model

5.3 Literature review

Many Cloud IDS have been proposed [75, 76, 107]. In [75], a security model is proposed to detect the attacks among the tenants of VMs (IaaS layer). In [76], Snort-based IDS for IaaS/PaaS layer is integrated in AWS. A SQL injection framework is offered for SaaS providers [107] ; however, it does not detect unauthorized access among the tenants of MTSaaS. All these previous IDS do not deal with the multi-tenancy issues of SaaS. However, many researches and enterprises have been proposed multi-tenancy encryption, design patterns and tenant isolation.

5.3.1 Multi-tenancy encryption

The data encryption with specific key for each tenant [49, 65] can ensure the confidentiality of tenants' data if the key of a tenant is not revealed to other tenants. However, it is costly in terms of performance (encrypting/decrypting of all data) and does not ensure the data integrity of tenants. For example, a malicious tenant can execute an unauthorized update/delete query against other tenants. 81.8% of Cloud providers encrypt the data in transit using TLS or SSL [66] which does not protect against inter-tenant attacks. However, only 9.4% of these providers encrypt tenants' data once it is stored in Cloud.

5.3.2 Multi-tenancy design patterns

In order to isolate tenants' data for MTSaaS, many multi-tenancy design patterns have been proposed at application and/or database levels [67, 68]. An application filter-based pattern is

to retrieve only the data of corresponding tenant (executor of HTTP request). This filtering is based on the tenant's database, tenant's tables (Orders1 for tenant1) and TenantID (1 for tenant1) depending on data model 1, 2 and 3, respectively. For example, a developer can write the following SQL Query to filter the data by TenantID at run-time (Data model 3) :

```
"Select * from Orders Where TenantID=" + pTenantID
```

A database permission-based creates an access account for each tenant with the necessary privileges. This pattern allows a tenant to only access its own database (for data model 1), set of tables or Schema (for data model 2), or rows of each table (for data model 3). For example, the administrator should give the users of tenant1 the permission to add, remove, modify, or delete only the rows of the table Orders1.

However, the effectiveness of these patterns depends on the competence of developers. Furthermore, the consequences are very serious if a developer forgets to filter on TenantID (e.g., the Where clause is used without TenantID=pTenantID) : a tenant can compromise the data of other tenants. Additionally, an attacker can bypass this filtering and return the orders of all tenants, for example, by injecting `1' OR 1=1-` in pTenantID parameter. Otherwise, a validation must be done, for example, to verify if a SQL query, executed from tenant1.SaaSApp.com, manipulates only the data of tenant1. This validation is costly, which mainly impacts the application performance, and is more complex in case of Insert, Update and Delete queries. Note that the management of privileges at database level is a complex task especially when hundred of tenants share the same tables.

5.3.3 Tenant isolation

In order to isolate tenants from each other, AWS [64] proposes to deploy the application of each tenant in a separated account, virtual private Cloud (VPC) or subnet. However, the SaaS provider should ensure the tenant isolation at application level for MTSaaS [64]. Azure [67] proposes that the developer applies the previous multi-tenancy design patterns. To overcome these limitations, few research papers have dealt with the data isolation of SaaS tenants. The authors of [69] propose to install an HTTP proxy on the internal network of each tenant in order to sign the HTTP using a secret key provided by SaaS provider. Thus, it does not meet the novel SaaS architecture where the application and database are installed on the provider's infrastructure. In order to reduce the risk of attacks among the competing tenants, the authors of [40] avoid, as much as possible, having the data of two tenants with the same BT in the same database. The proposed algorithms try to find a database, which does not have any tenant with the same BT and size of the new tenant. If these algorithms fail to find such as database even after the redistribution of existing tenants' data, the data of

new tenant will be randomly placed in a database. However, the distribution of tenants' data is costly in terms of performance and there is a high possibility to place the data of several tenants in the same database. A simple counterexample is when the data of tenant1, tenant2, and a new tenant (tenant3) are placed in the same database. In this case, the alternative is to dedicate a database for each tenant (Data model 1).

5.4 ITADP : Inter-tenant attack detection and prevention framework for multi-tenant SaaS

Our inter-tenant attack detection and prevention (ITADP) framework aims to identify and prevent a tenant user from accessing (maliciously or accidentally) the rows of other tenants of MTSaaS, regardless of the type of attack (e.g., SQL injection, modification of cookies/-parameters). Indeed, ITADP checks if a given SQL, executed from the domain of a tenant (e.g., tenant1.SaaSApp.com), manipulates the rows of other tenants (e.g., tenant2, tenant3) by abstracting the shared tables instead to access their full data. In order to enable a SaaS provider to perform additional protection, ITADP applies some common security mechanisms such as IP address blacklisting and HTTP checking.

The syntactic analysis of SQL queries is used to detect classic SQL injection attacks (syntactically incorrect, tautology, union, and piggybacked queries) according to the syntax by checking if an attacker injects special characters (`e.g.,', -, '`) [112–114]. These approaches check if the syntactic tree of a SQL query without user input (extracted from the source code) and the one generated with user input (at run time) are different ; if they are, there is a potential SQL injection attack. These approaches require analysis and/or modification of the source code and do not detect attacks between the tenants of MTSaaS because they are limited to syntactic analysis. For example, the SQL query (`select * from Order where TenantID >1`) returns the orders of all tenants ; however, it is syntactically correct, does not contain any special characters, and is not evaluated as true (tautology). In ITADP, we analyze the semantics of each SQL query by traversing its syntactic tree in order to verify that it can return or modify unauthorized tenants' data. The following subsections present the overview, requirements, modules, and integration of ITADP in a public Cloud.

5.4.1 Overview

ITADP works in detection and prevention modes and consists of Interceptor, Correlator, ITADetector, and HTTPChecker modules. Figure 5.3 shows the integration of ITADP modules into a three-tier MTSaaS (deployed in a Cloud) to detect inter-tenant attacks. The

Interceptor and Correlator are activated on the VM that exposes the data access layer (e.g., MTierVM in a three-tier architecture) only in detection mode. The Interceptor collects and sends the HTTP requests and SQL queries to the Correlator. The Correlator then identifies the source (HTTP, SOAP) of each SQL query. In prevention mode, the ITADetector is invoked in the source code (data access layer) to reject the SQL queries that considered as inter-tenant attacks. The ITADetector is deployed on a set of VMs (ITADVMs) to be able to analyze the large quantity of SQL queries received from all Correlators (detection mode) and VMs that expose the data access layer (prevention mode). An alert is stored in the ITADDB for each SQL query considered as an inter-tenant attack. The HTTPChecker is installed on each WTierVM to reject the HTTP requests coming from malicious tenant users. For two-tier MTSaaS, the HTTPChecker, Interceptor, and Correlator are installed on each WTierVM. The modules of ITADP are offered as a set of pre-built VMs (with all necessary tools). Thus, a SaaS provider can use these VMs to create and auto-scale its own VMs according to traffic.

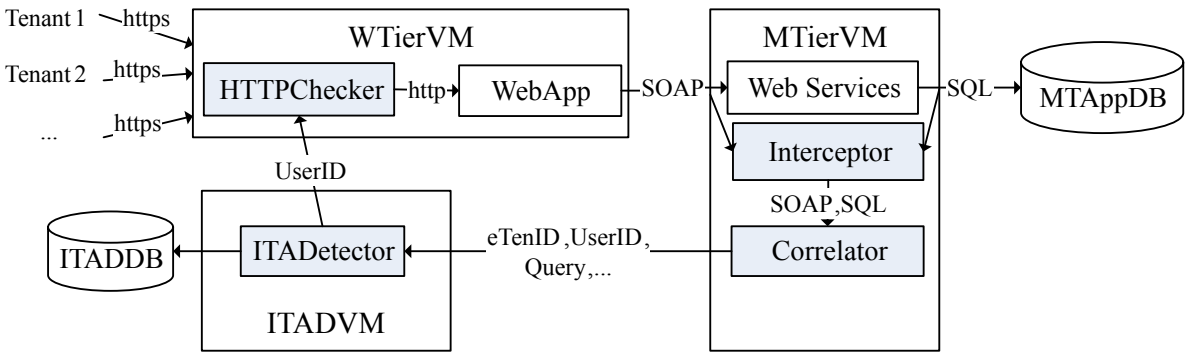

Figure 5.3 Overview of ITADP

5.4.2 Requirements

ITADP aims to meet the following requirements :

- **Accuracy :** The accuracy of ITADP is measured by the values : true positive, false negative, false positive, the detection rate and precision ;
- **Portability :** The ITADP should be portable to a MTSaaS, regardless of its programming language, technologies (e.g., Entity framework), and architecture (e.g., two-tier, three-tier) ;
- **Compatibility :** The ITADP should be compatible with HTTP, HTTPS (HyperText Transfer Protocol Secure) and each type of DBMS ;

— **Ease of integration :** The SaaS provider should integrate the ITADP with minimum configuration and modification of the MTSaaS. This integration should not impose constraints on the deployment of the MTSaaS ;
— **Efficiency :** The ITADP should work with a small overhead (e.g., CPU usage), especially, on MTSaaS VMs ;
— **Minimal impact :** The ITADP should have minimal impact on the HTTP response time when it is integrated to protect a MTSaaS.

5.4.3 Modules

Figure 5.4 resumes the principal functionality and the interaction of the modules of ITADP to process a SQL query. In detection mode, the Correlator sends (to the ITADetector) the SQL Query (Query) generated, for example by a function (F) of Web service (WS), after an action performed by the user (UserID) of (eTenID) from an IP address (IP), where eTenID is the identifier of the tenant that executes the SQL query. In prevention mode, the ITADetector receives the eTenID, UserID, IP, WS, F, and Query from the MTSaaS (data access logic) and returns the result of detection.

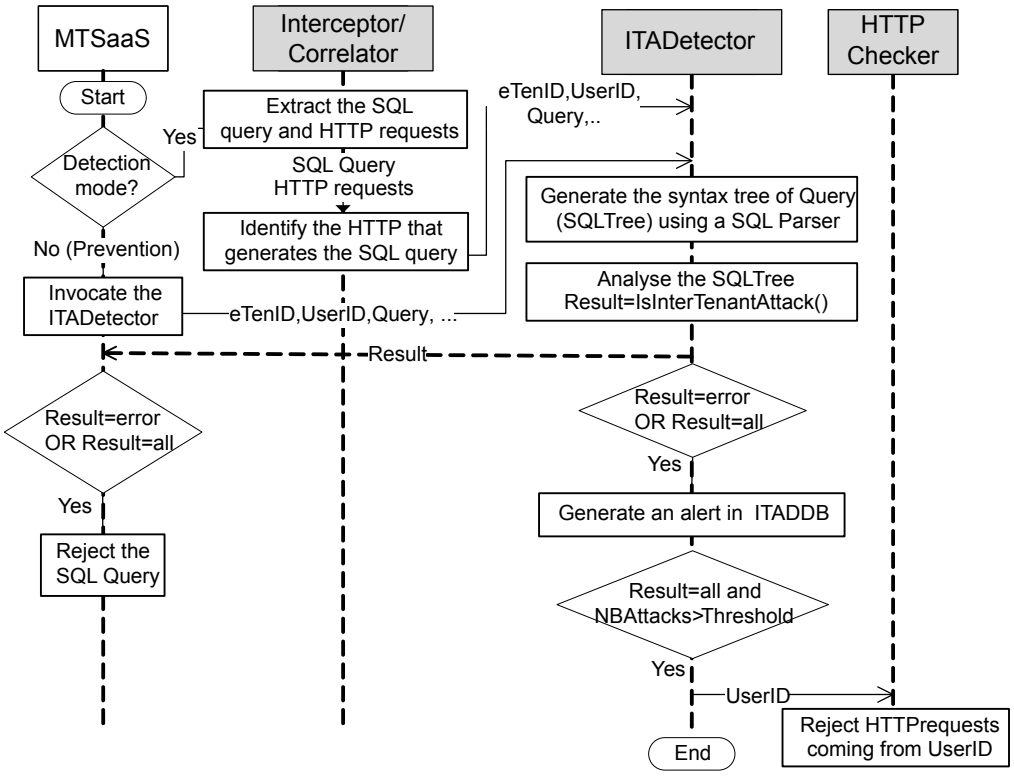

Figure 5.4 Main functionality and interaction of ITADP modules

Interceptor

The Interceptor collects the requests (HTTP, SOAP) and SQL queries. Thus, it should be placed and implemented according to the architecture (e.g., two-tier, three-tier) and the technologies (e.g., Entity Framework, Web service) of the MTSaaS. In a three-tier architecture, it is installed on each middle-tier VM (MTierVM) that contains the data access logic or Web services. To reduce the impact on the HTTP response time, the collection of traffic should be performed without slowing the HTTP requests and SQL queries (asynchronously) in detection mode.

Correlator

The Correlator identifies the request (e.g., HTTP, SOAP) that generates each SQL Query by executing the correlation algorithm of [107]. This algorithm realizes the correlation between HTTP and SQL traffic (at run-time) without modifying the source code of the application/-services. Thus, relevant information about the source of each SQL Query is extracted from the identified HTTP request, including the (eTenID), (UserID), (IP), (WS), and (F).

ITADetector

The ITADetector represents the core of ITADP which checks if a SQL Query allows a tenant user to access, in writing (Update, Insert, or Delete) or reading (Select), the rows of other tenants at run-time. The insert SQL queries can be easily validated in order to identify if they add a row with a TenantID different from the eTenID to a given table. For example, the following Insert query is considered as an inter-tenant attack if it is executed from tenant1.SaaSApp.com (eTenID=1) :

```
Insert into Orders(TenantID,OrdNum,OrdDate, .)values(2,45,01/09/2017,.).
```

However, the Select, Update, or Delete queries return, modify, or delete tenant rows depending on the Where clause. The validation of these queries is a complex task since the expression (Expr) of a Where clause is a combination of an unlimited number of different types of expressions (e.g., composed condition, atomic condition, sub queries). In order to have a structured representation of the Expr of a Where clause, a SQL Parser [25] is used to generate the syntactic tree (SQLTree) of the Query, including the binary syntactic tree of the Where clause (WhereTree). Table 5.1 presents a basic SQL grammar [25, 115, 116] that is adapted to a multi-tenant database when two types of attributes for each shared table can be distinguished : the ID of the tenant (TenantID) and all other attributes (Attr).

Table 5.1 Basic SQL grammar for multi-tenant database

Element	Symbol	Form/Description
Multi-tenant database	MTAppDB	{ta1, ta2,...}
Table	ta	(TenantID, attr1, attr2,...)
TenantID	TenantID	TenantID attribute
Attribute	Attr	All attributes except TenantID
Constant	Cst	Constant value
SQL Query	Query	SQLI|SQLU|SQLD|SQLS
SQL Insert	SQLI	Insert into ta (TenantID, Attr1,..) Values(pTenantID,v1,..) Insert into ta(TenantID,Attr1,Attr2,...) SQLS
SQL Update	SQLU	Update ta Set attr1=v1,... Where (Expr)
SQL Delete	SQLD	Delete ta Where (Expr)
SQL Select	SQLS	Select * From ta1, ta2, ... Where (Expr) LSQLS UNION|INTERSECT|EXCEPT RSQLS (Combined)
Expression	Expr	LExpr|Cond AND RExpr|Cond LExpr|Cond OR RExpr|Cond NOT(LExpr|Cond) Attr|TenantID IN SQLS
Atomic condition	Cond	TenantID OperC Cst Attr|Cst OperC Attr|Cst
Comparison operator	OperC	=| != |>|>=|<|<=|LIKE
Where syntax tree	WhereTree	binary syntax tree of the Where clause

Figure 5.5 shows a generic syntactic binary tree of the Where clause (WhereTree), formulated according to the previous SQL grammar and generated by the SQLParser. Each node (n) is a tuple (type, value, lnode, rnode), defined as follows. The type is root, internal, or leaf. The value is an expression (Expr) if the type is root or internal. Otherwise, it is a condition (Cond). The lnode and rnode are the left child and right child.

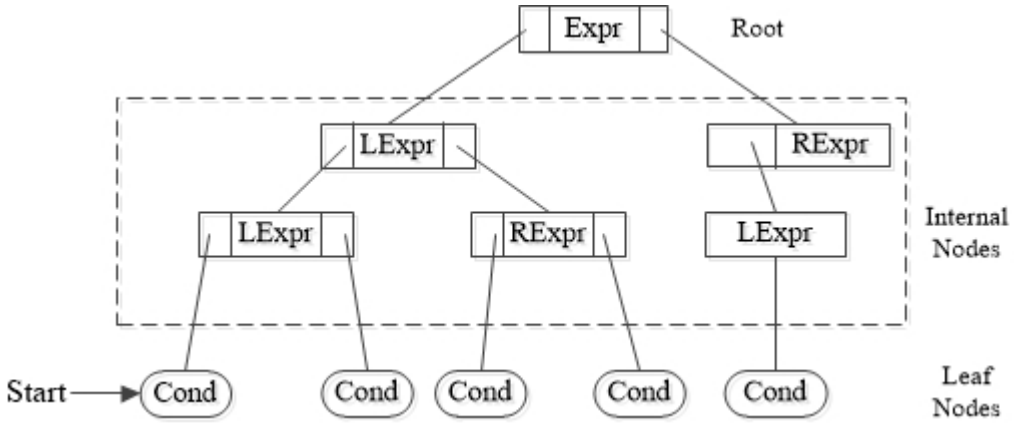

Figure 5.5 binary syntactic tree of Where clause (WhereTree)

The ITADetector analyzes the syntactic tree (SQLTree) of the Query (generated by SQLParser) according to a SQL grammar (see Table 5.1) and many proposed rules. It determines if the Query is composed of two Select queries (IsCombined=True) and executes the IsInterTenantAttack algorithm (see Algorithm 1), which works as follows. For the Update, Delete, uncombined Select queries, the VisitNode algorithm is applied on the root of the WhereTree (1-3). Otherwise, the VisitNode is applied on the root of the WhereTree of the left and right queries, depending on the Minus (See Table 5.2), Union (See Table 5.3), or Intersect (See Table 5.4) operation that combines the left and right queries. Then, the ITADetector works according to the value (Result) returned by the IsInterTenantAttack algorithm :

— Result $\in$ {*none*,*executor*} : The Query is considered normal.

— Result = *all* : The Query is considered as a malicious inter-tenant attack. If the number of attacks realized by the tenant user (UserID) is more than a threshold, the UserID is sent to the HTTPChecker. This threshold is defined by the provider for each tenant(e.g., according to its number of attacks).

— Result = *error* : The Query is considered as an accidental inter-tenant attack. In this case, there is no additional reaction against the tenant user because it is not responsible for this attack. However, the provider should correct the source code to prevent the vulnerability that caused the accidental inter-tenant attacks.

Algorithm 1 IsInterTenantAttack : returns none, all, executor, or error

Input : eTenID, Type, SQLTree, IsCombined
Output : *none* or *executor*, if the Query is normal
Output : *all*, if the Query is a malicious inter-tenant attack
Output : *error*, if the Query is an accidental inter-tenant attack
// S : Select, U :Update, D :Delete
1: **if** ((Type==‘S’ **AND** IsCombined == False) **OR** (Type ∈ {‘U’,‘D’})) **then**
2: **return** VisitNode($eTenID$,$WhereTree.Root$)
3: **end if**
// The syntax form of Query (Combined) is $LSQLS\ op\ RSQLS$
4: **if** ($Query$ is ($LSQLS\ op\ RSQLS$) **AND** op ∈ {‘MINUS’,‘INTERSECT’,‘UNION’}) **then**
5: **return** VisitNode($eTenID$,$LSQLS.WhereTree.Root$) *op* VisitNode($eTenID$,$RSQLS.WhereTree.Root$)
6: **end if**

Table 5.2 The MINUS operation

MINUS	none	all	executor	error
none	none	none	none	error
all	all	none	all	error
executor	executor	executor	none	error
error	error	error	error	error

Table 5.3 The UNION/OR operations

UNION(∪)/OR	none	all	executor	error
none	none	all	executor	error
all	all	all	all	error
executor	executor	all	executor	error
error	error	error	error	error

Table 5.4 The INTERSECT/AND operations

INTERSECT(∩)/AND	none	all	executor	error
none	none	none	none	error
all	none	all	executor	error
executor	none	executor	executor	error
error	error	error	error	error

VisitNode : The VisitNode algorithm (see Algorithm 2) recursively traverses the binary tree of the Where clause (WhereTree) according to InOrder depth-first traversal [117]. The traverse order is left child, node, and right child. The starting point is the lowest left level of the WhereTree, which should be determined a priori (by passing through the left child of

nodes from the root until the last leaf). This algorithm is applied (recursively) to each node (n) and returns *none* (The Query does not satisfy the rows of any tenant), *executor* (The Query only satisfies the rows of its executor (eTenID)), *all* (The Query can satisfy the rows of all tenants), or *error* (a potential accidental attack).

- Steps 1-3 (trivial case) : If the current node of the WhereTree is a leaf that contains an atomic condition Cond (Figure 5.5), the TenantsSatCond algorithm (Algorithm 3) is invoked. The VisitNode algorithm must stop without traversing the rest of the WhereTree if the TenantsSatCond algorithm returns (*error*).
- Steps 4-6 : If the value of the current node is an expression (Expr) with a sub query (SQLS), the VisitNode is applied on the SQLS (lnode).
- Steps 7-9 : If the syntax of the current node (n) is a left expression (LExpr) and right expression (RExpr) combined with an operation op (OR/AND), VisitNode(n) = VisitNode(lnode) *op* VisitNode(rnode), where lnode.value = LExpr and rnode.value = RExpr. The (OR) and (AND) operations are defined in Table 5.3 and Table 5.4, respectively.
- Steps 10-12 : If the current node (n) has one child and the n.value is NOT(LExpr), the VisitNode algorithm is applied (recursively) on its child (lnode) until it reaches a leaf with a atomic condition (Cond). The NOT operand is defined in Table 5.5.

Algorithm 2 VisitNode : traverses recursively the WhereTree (*Expr*) and returns *none*, *all*, *executor*, or *error*

Input : eTenID
Input : Node n=(type,value,lnode,rnode) : type∈{root,internal,leaf}, value is an expression (Expr) or a condition (Cond)
Output : *none*, *all*, *executor*, or *error*

1: **if** ($n.type == leaf$) **then** ▷ n is leaf
// Trivial case : leaf node with an atomic condition (Cond) as value
2: **return** TenantsSatCond($eTenID$,$n.value$)
3: **end if**
4: **if** ($n.value$ is ($Attr$ IN $SQLS$) **OR** $n.value$ is ($TenantID$ IN $SQLS$)) **then**
5: **return** VisitNode($eTenID$,$n.lnode$) ▷ n has one child lnode
6: **end if**
// n.value is the combination of left (LExpr) and right (RExpr) expressions
7: **if** ($n.value$ is ($LExpr$ op $RExpr$) **AND** op ∈ {OR,AND}) **then**
8: **return** VisitNode($eTenID$,$n.lnode$) op VisitNode($eTenID$,$n.rnode$)
9: **end if**
10: **if** ($n.value$ is ($NOT(LExpr)$)) **then** ▷ n has one child lnode
11: **return** NOT (VisitNode($eTenID$,$n.lnode$))
12: **end if**

Table 5.5 The NOT operand

Operand	NOT
none	all
all	none
executor	all

TenantsSatCond : The TenantsSatCond algorithm (see Algorithm 3) takes the eTenID and the atomic condition Cond (Cond ; leaf nodes of the WhereTree) as parameters and returns *none*, *all*, *executor*, or *error* according to the following rules :

— R1(1-7) : This algorithm adopts an optimization in step 5 ; if an atomic condition is different from TenantID = eTenID, the value (*error*) is returned to the VisitNode algorithm because a Query executed by eTenID should never contain a condition that satisfies other tenant identifiers (e.g., TenantID <eTenID).

— R2 (8-10) : The best case for a malicious tenant user (the worst case for the attacked tenants) is where the conditions such as (*Attr OperC Attr*), (*Attr OperC Cst*) satisfy the rows of other tenants at run time. In other words, the table (e.g., Order) rows of all TenantIDs that satisfy these conditions can be accessed by the malicious user (UserID) of eTenID.

— R3 (11-17) : A true condition (independent of TenantID) satisfies the rows of all tenants (e.g., TenantsSatCond (1=1) = *all*). However, a false condition does not satisfy any tenant's row (e.g., TenantsSatCond (1=2) = *none*).

Algorithm 3 TenantsSatCond : returns *none*, *all*, *executor*, or *error*

Input : eTenID
Input : An atomic condition (Cond)
Output : *none*, *all*, *executor*, or *error*

1: **if** (*Cond* is (*TenantID OperC Cst*)) **then** ▷ is : the same syntax form
2: **if** (*Cond* is (*TenantID* = *eTenID*)) **then**
3: **return** *executor*
4: **else**
5: **return** *error* ▷ The Query is an accident inter-tenant attack
6: **end if**
7: **end if**
8: **if** (*Cond* is (*Attr OperC Attr*) **OR** *Cond* is (*Attr OperC Cst*)) **then**
9: **return** *all*
10: **end if**
11: **if** (*Cond* is (*Cst OperC Cst*)) **then**
12: **if** ((*Cst OperC Cst*) *is true*) **then**
13: **return** *all*
14: **else**
15: **return** *none*
16: **end if**
17: **end if**

HTTPChecker

The HTTPChecker checks the HTTP requests to reject those coming from users within the list of malicious users (received from the ITADetector) for a specific period of time (defined

by the provider). A Web page is displayed to inform the user (who is considered as malicious) that their HTTP requests will be rejected during this period. Thus, the provider can confirm that this user controls its account by communicating with its tenant. Then, the provider can decide to delete or not the UserID from the list of malicious users. The HTTP Checker is implemented as a proxy and not in the source code of the application. To make the ITADP compatible with HTTP and HTTPS protocols, the HTTPChecker is deployed on each WTierVM when the HTTP requests are decrypted.

Illustrative examples

Let us consider the following SQL queries in the source code of Web service functions invoked by the application :

- `SQL1="Select * from Orders Where (TenantID=" + pTenantID + " and OrderNum>" + txtOrderNum.Text + ")"`
- `SQL2="Select * from Orders Where (TenantID=" + pTenantID + " and OrderDate>" + txtOrderDate.Text + ")"`
- `SQL3="Select * from Orders Where (TenantID<>" + pTenantID + ")"`

Assume that the user1t1, user1t2 and user1t3 execute during a small period (t) three HTTP requests from the domains of tenant1, tenant2 and tenant3 as follows. user1t1 enters 1 for pTenantID and 2 OR (1=1) for txtOrderNum. user1t2 enters 2 for pTenantID and 12/01/2017 for txtOrderDate. user1t3 enters 3 for pTenantID.

The application generates and sends the following SOAP requests to Web service :

- `SOAP1: Params={1,2) OR (1=1}`
- `SOAP2: Params={2,'12/01/2017'}`
- `SOAP3: Params={3}`

The Web service generates and sends the following SQL queries to database :

- `Query1: Select * from Orders Where (TenantID= 1 and OrderNum>2) OR (1=1):` returns the orders of tenant1 with order number greater than 2.
- `Query2: Select * from Orders Where (TenantID= 2 and OrderDate> '12/01/2017'):` returns the orders of tenant2 created after '12/01/2017'.
- `Query3: Select * from Orders Where (TenantID <>3):` returns the orders of tenant3.

The Interceptor captures the details of SOAP1, SOAP2, SOAP3 and Query1, Query2, Query3 and the Correlator associates the SQL queries to the corresponding SOAP requests as follows : (SOAP1, Query1), (SOAP2, Query2) and (SOAP3, Query3). The SQLParser generates the WhereTree of Query1, Query2 and Query3 (see Figure 5.6). Query1 and Query3 are considered as inter-tenant attacks : VisitNode(Query1.WhereTree.Root)=*all* and VisitNode(Query3.WhereTree.Root)=*all*. Query2 is considered as normal :

VisitNode(Query2.WhereTree.Root)=*executor*.

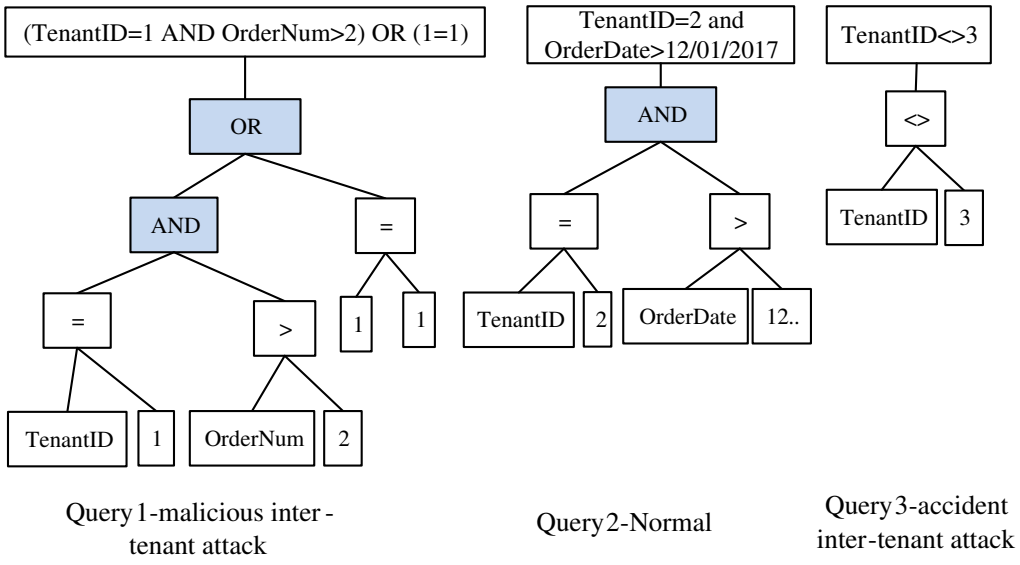

Figure 5.6 Where binary syntax trees (WhereTree) of Query1, Query2 and Query3

5.4.4 Integration in a public Cloud

Figure 5.7 shows the integration of ITADP in a public Cloud (AWS) to detect or prevent the inter-tenant attacks that targets a MTSaaS. The MTSaaS is deployed in AppSNet subnet of VPC and consists of an internet gateway, a HTTP load balancer (HTTPLB), several WTierVMs (WTierVM1, WTierVM2, ...), an internal load balancer (INTLB), several MTierVMs (MTierVM1, MTierVM2, ...) and MTAppDBVM. The internet gateway allows the VMs in the VPC to communicate with the Internet. An instance of the MTSaaS (e.g., Web pages, presentation layer) is installed on each WTierVM. The HTTPLB distributes the HTTP requests over the WTierVMs. The middle logic (e.g., data access, Web services) is installed on each MTierVM. The INTLB distributes the traffic (e.g., SOAP requests) over the MTierVMs. The multi-tenant database (with a shared table and schema) is installed on MTAppDBVM.

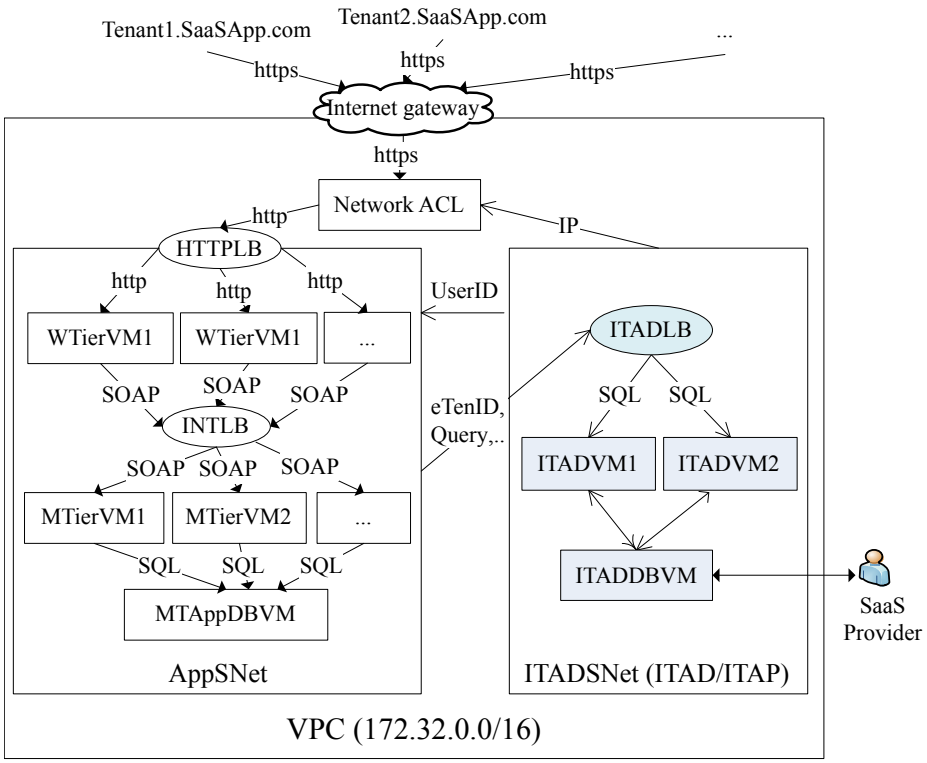

Figure 5.7 Integration of ITADP in a public Cloud (AWS)

ITAD : inter-tenant attack detection

The detection integration of ITADP (ITAD) consists of a private sub net (ITADSNet) in which ITADVM1, ITADVM2, ITADDBVM and an internal load balancer (ITADLB) are created. Additional ITADVMs can be created on-demand. In order to ensure a high availability, the ITADVM1 and ITADVM2 are created in different availability zones and the ITADLB then distributes the traffic (the calls of ITADetector) over ITADVM1 and ITADVM2. The Interceptor and Correlator are installed on each WTierVM (two-tier) or MTierVM (three-tier).

Once the ITADetector detects a number (more than a threshold) of malicious inter-tenant attacks realized by the user (UserID) of a tenant (eTenID), the traffic coming from the IP address (extracted from the HTTP request) of the user's machine is automatically blocked. Moreover, the ITADetector adds an inbound rule to the network access control list (ACL), associated with the VPC, to deny the traffic from this IP using AWS SDK for .NET. In this way, multi-level security is offered at the data (SQL queries), HTTP (by HTTPChecker), and VPC (network ACL) levels.

The data of tenants (e.g., TenantID, TenantName, DomainName), who are subscribed to MTSaaS, should exist in a table. Thus, the list of tenants is extracted from this table and stored in XML file. Then, the administrator of the MTSaaS configures (e.g., to modify the thresholds of tenants) this file to be used by the modules and algorithms of ITADP. We avoid the automatic adding of TenantID (by the ITADetector) during the detection process to prevent an attacker to inject a malicious TenantID.

ITAP : inter-tenant attack prevention

The inter-tenant attack prevention (ITAP) of ITADP is integrated into ITADSNet (see Section 5.4.4). In order to reject the SQL queries that represent inter-tenant attacks, the detection function of the ITADetector should be called synchronously in the source code (data access) before sending SQL queries to the database. Thus, there is no need to activate the Interceptor and Correlator because the necessary information (e.g., UserID, eTenID, Query) can be passed to the function as parameters.

5.5 Implementation and evaluation

We conducted our experiments in AWS public Cloud (Figure 5.7). Table 5.6 shows the specifications and costs of all VMs and load balancers in AWS.

Table 5.6 Specifications/costs of all VMs and load balancers in AWS

	VMs/Load Balancers	Size/type	Subnet	Modules	Hourly Cost
MTSaaS	WTierVM	t2.xlarge,windows, 4vCP,16GiB RAM	AppSNet	HTTPChecker -2-tier :Interceptor, Correlator	0.2266
	MTierVM	t2.xlarge,windows, 4vCP,16GiB RAM	AppSNet	-3-tier :Interceptor, Correlator	0.2266
	HTTPLB/ INTLB	Load balancer	AppSNet	N/A	0.025
	MTAppDBVM	t2.2xlarge,windows, 8vCPU,32GiB RAM	AppSNet	Multi-tenant database	0.4332
ITAD ITAP	ITADLB	Load Balancer	ITADSNet	N/A	0.025
	ITADVM	t2.large,windows, 2vCPU,8GiB RAM	ITADSNet	ITADetector	0.1208
	ITADDBVM	t2.small,windows, 1vCPU,2GiB RAM	ITADSNet	ITADDB	0.032

5.5.1 Implementation of ITADP

The modules of ITADP are implemented in C# (.NET). The implementation of the Interceptor depended on the technology used to communicate with the databases. The Correlator implemented the correlation algorithm of [107]. The Interceptor and Correlator are deployed on each WTierVM (in a two-tier architecture) and MTierVM (in a three-tier architecture). In ITAD, the ITADetector is implemented as a C# application (ITADApp) that received the data from the Correlators using client-server socket programming. In ITAP, it is implemented as a Web method (IsInterTenantAttack) of a Windows Communication Foundation (WCF) Web service (ITAPWS). The SQL trees were generated using the SQLParser of [25] for two reasons : (1) it generates the WhereTree as a binary tree, which allows the application of the proposed algorithms ; (2) it accepts any DBMS (e.g., MSSQL, ORACLE) as a parameter, which makes ITADP compatible with DBMS. The HTTPChecker is implemented as a reverse proxy Web server using Fiddler API [118] and is installed on each WTierVM. It verifies whether or not the (UserID) belonged the blacklist of malicious users in order to reject malicious HTTP requests.

5.5.2 Applications

The ITADP is evaluated based on the following applications :

— **MTSaaS1 :** HacmeBank [119] is a vulnerable online banking application. The application (ASP.NET) communicates with the database by invoking the Web services (middle-tier). We modified the application and services to support several tenants. A TenantID column is added to each table, and new rows are generated from the existing rows for all tenants. The application, Web services, and multi-tenant database are deployed on a WTierVM, four MTierVMs, and a MTAppDBVM (three-tier architecture). The Interceptor is implemented using WinPcap API [120] to extract the HTTP requests (port 80) and SQL queries (port 1443 for MSSQL).

— **MTSaaS2 :** EventCloud [121] is a multi-tenant SaaS, built using ASP.NET, model, view and controller (MVC) pattern, Angular, and Entity Framework. It enables all tenants that share the same tables of the single database to manage their users and events. We deployed this application on four WTierVMs and its database on MTAppDBVM. The validation of input users is disabled. Since Entity Framework is used to communicate with the database, an Interceptor class is configured and implemented in the application to capture the HTTP requests and the SQL queries.

5.5.3 Scenarios and experimental results

We evaluated the overhead, the detection quality, the impact on response time and the infrastructure cost of ITADP based on the previous SaaS applications (MTSaaS1 and MTSaaS2). For each application, a multi-thread traffic generator (called TrafficGenerator) is developed to generate a large quantity of HTTP requests (malicious and normal) for four tenants. SQLMap [103], an open source penetration testing tool, is used in a thread to generate HTTP requests with classic SQL injection attacks. Since SQLMap neither covers all inter-tenant attacks nor normal traffic, some threads generated the HTTP requests with the remaining attacks, such as one-to-one (see section 5.2.2) and attacks without special characters. Other threads generate normal HTTP requests (which did not access the data of other tenants). The TrafficGenerator of each application was deployed on a high-performance VM (outside of the VPC), AttackVM, to stress the application and VMs with a large quantity of mixed HTTP requests during a small time period. It was executed during the following scenarios :

- **Scenario 1 (without ITADP) :** The Interceptor and Correlator were deactivated, and the ITADP VMs (ITADVM1, ITADVM1, and ITADDBVM) were stopped ;
- **Scenario 2 (with ITAD) :** The ITADP VMs (ITADVM1, ITADVM1, and ITADDBVM) were started. The Interceptor and Correlator were activated on each MTierVM (for MTSaaS1) and WTierVM (for MTSaaS2). The ITADApp was activated on each ITADVM ;
- **Scenario 3 (with ITAP) :** The ITADVM1, ITADVM2, and ITADDBVM were started. The Interceptor and Correlator were deactivated. The Web method (IsInterTenantAttack) of ITAPWS was invoked in synchronous mode in the source code of the Web service (for MTSaaS1) and application (for MTSaaS2).

All previous scenarios were performed without the activation of HTTPChecker. Otherwise, the grater majority of HTTP requests would be rejected, and, thus, the quality detection, HTTP response, and overhead would not be evaluated.

Overhead (CPU)

To evaluate the overhead of ITADP, we used Amazon CloudWatch, a monitoring and management service, to measure the CPU usage on the application (MTSaaS1 and MTSaaS2) and ITADP VMs.

MTSaaS1 : Figure 5.8 and Figure 5.9 show the percentage of CPU usage on MTSaaS1 and ITADP VMs, respectively. Scenario 1, scenario 2, and scenario 3 took approximately 16, 16, and 11 minutes, respectively, to process 107,621 HTTP requests. The comparison between Figures 5.8.a and 5.8.b demonstrates that the ITAD requires about 6% as a maximum ad-

ditional CPU usage (on MTierVM2) to capture and correlate the SOAP/SQL traffic. The comparison between Figures 5.8.a and 5.8.c demonstrates that the integration of ITAP decreases the traffic period (for five minutes) and CPU usage on MTSaaS1 VMs from 20% (scenario 1) to 8% (scenario 3) on MTierVM2. Indeed, the rejection of the majority of HTTP requests (in scenario 3) allowed the MTSaaS1 application to process the HTTP requests faster than scenario 1 and scenario 2. Note that the execution of each inter-tenant attack consumes a lot of resources since the data of several tenants can be treated. The comparison between Figures 5.9.a and 5.9.b demonstrates that the maximum CPU percentage used by ITADP on ITADVM1 increased from 18% (with ITAD) to 27% (with ITAP). The CPU consumption of ITAP was greater than ITAD for two reasons. First, the ITAP was implemented as a Web service and exposed by Internet Information Services (IIS) that should be activated on ITADVMs. Second, it analyzed the same quantity of SQL queries but during a smaller time period than scenario 2.

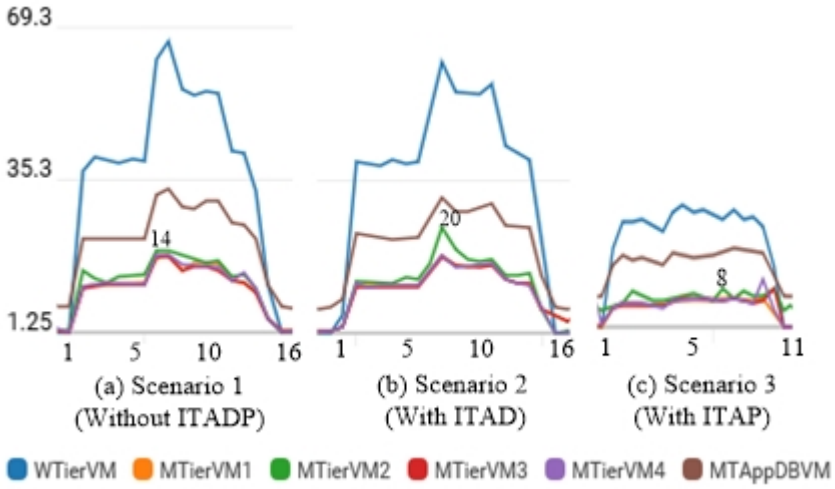

Figure 5.8 x-axis (traffic period in minutes) and y-axis (%CPU usage on MTSaaS1 VMs)

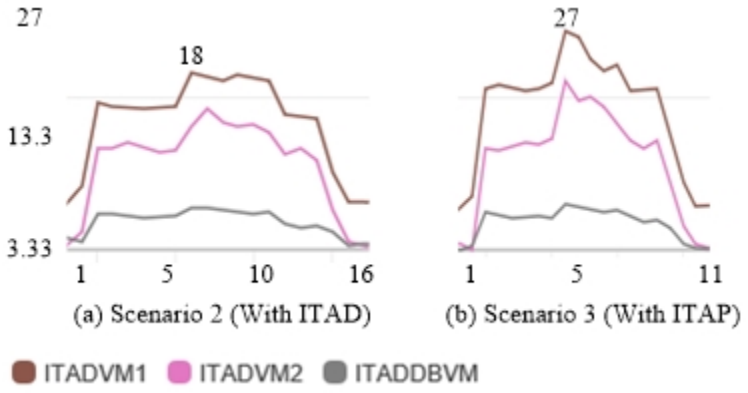

Figure 5.9 x-axis (traffic period in minutes) and y-axis (%CPU usage on ITADP VMs)

MTSaaS2 : Figure 5.10 shows the percentage of CPU usage on MTSaaS2 VMs without ITADP, with ITAD, and with ITAP. Scenario 1, scenario 2, and scenario 3 took approximately 15, 15, and 14 minutes, respectively, to process 105,468 mixed (normal and malicious) HTTP requests. The comparison between Figures 5.10.a and 5.10.b demonstrates that the ITAD slows slightly the processing of HTTP requests and requires insignificant additional CPU usage on WTierVMs to send the SQL queries to ITADApp. The comparison between Figures 5.10.a and 5.10.c demonstrates that the MTSaaS2 is capable of processing the same quantity of HTTP requests in scenario 3 at a faster rate than scenario 1 and scenario 2 due to the rejection of potential malicious HTTP requests (with inter-tenant attacks). The comparison between Figures 5.11.a and 5.11.b demonstrates that ITAP utilizes a maximum percentage of the CPU (20%) more than ITAD (14%).

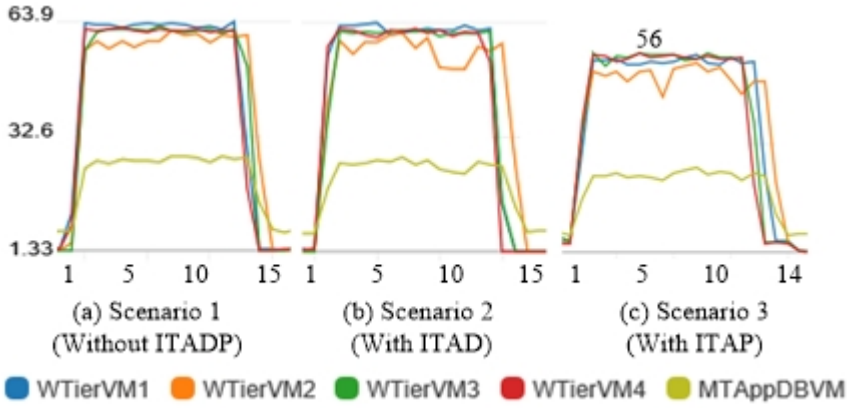

Figure 5.10 x-axis (traffic period in minutes) and y-axis (%CPU usage on MTSaaS2 VMs)

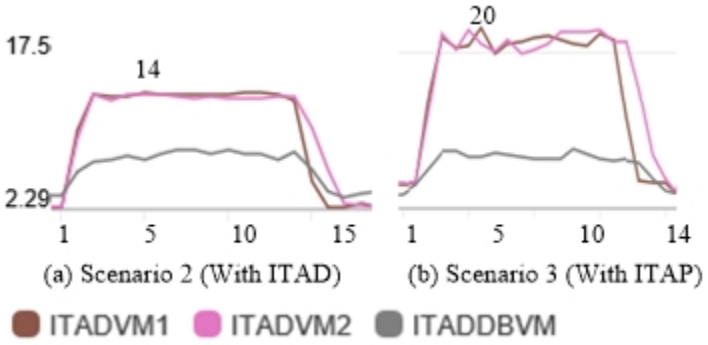

Figure 5.11 x-axis (traffic period in minutes) and y-axis (%CPU usage on ITADP VMs)

Detection quality

Generally, the detection quality of IDS can be measured according to the values of true positive (TP), false negative (FN), false positive (FP), the detection rate (DR), and precision (PR) [107]. In the context of ITAD, a malicious query is an inter-tenant attack that can allow (at run-time) a tenant user to access the data of other tenants. However, a normal query does not allow a tenant user to access the data of other tenants. Thus, we defined the TP, FN, FP, DR, and PR as follows. The TP is the number of malicious queries detected by ITADP. The FN is the number of malicious queries that are not detected by ITADP. The FP is the number of normal queries considered as malicious queries by ITADP. The detection rate (DR) and precision (PR) are the TP/(TP + FN) and TP/(TP + FP), respectively.

In order to measure the TP, FN and FP, we replayed scenario 2 by storing all SQL queries with the detection result (true or false) of ITADP. After that, we executed all SQL queries on the real shared tables. If the execution result of a given SQL satisfies the rows of tenants other than the executor, it was considered as a malicious query ; otherwise, it was considered normal query. Note that the Update, Delete, and Insert where converted into select queries to keep the original data. For MTSaaS1, the TP, FN, FP, DR, and PR values were 52988, 1581, 3471, 0.971, and 0.938 with ITAD. They were 53928, 1031, 1801, 0.981, and 0.967 with ITAP. For MTSaaS2, the TP, FN, FP, DR, and PR values were 10336, 486, 791, 0.954, and 0.929 with ITAD. They were 10346, 478, 748, 0.955, and 0.932 with ITAP. The detection quality of ITAP was a bit better than ITAD because there was no need to activate the Correlator.

HTTP response time

The LoadImpact tool [122] measures the performance of an application by sending a massive amount of normal HTTP requests. In order to evaluate the impact of ITAD and ITAP on the HTTP response time, the LoadImpact tool was used to simulate 50 virtual users in five minutes under the previous three scenarios. Table 5.7 shows the number of normal HTTP requests generated by LoadImpact for MTSaaS1 and MTSaaS2. These results reveal that adding ITAD and ITAP decreased the number of HTTP requests for MTSaaS1 to 615 and 1534, respectively. The adding of ITAD and ITAP decreased the number of HTTP requests for MTSaaS2 to 337 and 1653, respectively.

Infrastructure cost

The hourly infrastructure cost of ITADP is calculated according to table 5.6. In order to simplify the calculation, we assume that all VMs have the same volume (40 GB) and that the hourly storage cost of each VM is 0.0028 USD. As a result, ITADP costs about 0.307 USD for one or part of a hour.

Table 5.7 Quantitative evaluation of ITADP based on MTSaaS1 and MTSaaS2

Application	Criteria	Without ITADP	ITAD	ITAP
MTSaaS1	Detection rate (DR)	N/A	0.971	0.981
	Precision (PR)	N/A	0.938	0.967
	Normal HTTP requests using LoadImpact (5 minutes)	36,185 HTTP	35,570 HTTP	34,651 HTTP
	Mixed 107,621 HTTP requests	16 minutes	16 minutes	11 minutes
MTSaaS2	Detection rate (DR)	N/A	0.954	0.955
	Precision (PR)	N/A	0.929	0.932
	Normal HTTP requests using LoadImpact (5 minutes)	37,468 HTTP	37,131 HTTP	35,815 HTTP
	Mixed 105,468 HTTP requests	15 minutes	15 minutes	14 minutes

5.5.4 Comparative evaluation

In this section, we compare the impact of ITAD and ITAP on MTSaaS1 and MTSaaS2 according to the requirements of Section 5.4.2.

Qualitatively : ITAD and ITAP equally meet the compatibility requirement because they support the HTTP/HTTPS protocols and DBMS, whether it is for MTSaaS1 or MTSaaS2. ITAD meets the portability requirement more than ITAP, which requires a modification in the source code to prevent the inter-tenant queries. The ease of integration of ITAD

depends on its Interceptor module which is related to the communication technology with the database (Entity Framework, ADO.NET). As a result, the ITAD can be integrated into MTSaaS1 more easily than into MTSaaS2. The Interceptor works at the TCP protocol level without any modification of MTSaaS1. However, it is implemented and integrated as a class in MTSaaS2. The impact of ITAP is higher than ITAD on HTTP response time because the invocation of ITAPWS slows down all SQL queries to check them, even if they are normal.

Quantitatively : According to Table 5.7, ITADP is slightly more accurate for MTSaaS1 than for MTSaaS2 due to the complexity of the SQL queries (e.g., nested SQL queries) generated by the Entity Framework. Despite the generation of a similar number of HTTP requests during a similar period of time for MTSaaS1 and MTSaaS2 (see table 5.7), the ITADP utilizes the CPU on MTSaaS1 VMs more than on MTSaaS2 VMs. Thus, the ITADP can be considered to be more efficient for MTSaaS2 than for MTSaaS1.

5.6 Discussion

As shown in Section 5.5, the ITADP is integrated and evaluated according to two MTSaaS in a real IaaS Cloud (AWS). However, it is portable to other IaaS platforms (e.g., Azure) because it is offered as a set of VMs and uses common IaaS techniques (e.g., VPC, load balancers, network ACL). A SaaS provider can develop and deploy a MTSaaS using PaaS Cloud (e.g., force.com) without controlling the infrastructure of application (e.g., VM, VPC). In this case, the ITAP can be invoked in the source code of application since it is implemented as a Web service (ITAPWS).

ITADP works as an application vulnerability scanner that performs black box testing on a MTSaaS to identify potential inter-tenant attacks. Since the ITAD is provided as pre-built VMs (e.g., amazon machine image), a SaaS provider can create its own ITADVMs and ITADDBVM from these VMs to monitor a MTSaaS. At the end of a traffic generation period (scenario 2 in Section 5.5.3), the provider can consult the ITADDB to find the detected inter-tenant attacks (e.g., Select * from Orders where TenantID<>1) with the source of vulnerability (e.g., web service). Thus, the provider can modify the source code to prevent these inter-tenant attacks before being exploited by malicious tenants.

ITADP provides additional prevention at the HTTP and infrastructure (VPC) levels. First, the HTTPChecker quickly rejects the HTTP requests of malicious users when a number (defined for each tenant) of SQL queries are considered as malicious inter-tenant attacks. Second, the traffic originating from malicious IP addresses can be rejected since the corresponding inbound rules are added automatically.

5.7 Conclusion

Despite the advantage of multi-tenancy in SaaS, especially for reducing the cost of resources and maintenance, there is still an important need for SaaS providers and tenants to detect and/or prevent attacks among tenants. In order to detect these new attacks, caused by this multi-tenancy, we propose the ITADP to SaaS providers since the tenants do not control the source code or underlying resources of the application. We implemented a prototype of ITADP to demonstrate its practicality. We also proposed and compared the detection and prevention integrations of ITADP. Experiments demonstrated that ITADP provides a protection against inter-tenant attacks with a small overhead and minimal impact on the application. In our future work, we will work on the adaptation of ITAD to the NoSQL database and/or REST Web services.

CHAPITRE 6 ARTICLE 3 : MULTI-TENANT INTRUSION DETECTION FRAMEWORK AS A SERVICE FOR SAAS

Mohamed Yassin, Hakima Ould-Slimane, Chamseddine Talhi and Hanifa Boucheneb
Revised and submitted to : IEEE Transactions on Cloud Computing

Abstract Information technology (IT) service providers are nowadays moving toward cloud computing. Software-as-a-service (SaaS) refers to cloud-based service-oriented web applications. As a result of computation outsourcing, a customer (tenant) can subscribe to a self-service SaaS and use it on a pay-per-use basis. To reduce resource costs, a single instance of SaaS serves multiple tenants (multi-tenancy). However, outsourcing and multi-tenancy bring about new security issues. Indeed, tenants lose control over the source code, databases and infrastructure and cannot deploy their own intrusion detection system (IDS). In this context, the provider must not only integrate their preferred IDS into a public cloud, but also protect the tenants according to their individual security requirements. We put forth a multi-tenant intrusion detection framework as a service for SaaS (MTIDaaS) to allow the provider to undertake such integration. Our MTIDaaS has been integrated and tested in a real public cloud environment. It provides security-as-a-service (SecaaS) for both provider and tenant with high levels of portability, flexibility and cost-effectiveness. The experimental results demonstrate that our MTIDaaS offers easy integration of IDS with little virtualization overhead and insignificant impact on HTTP response time.

6.1 Introduction

Cloud computing (CC) is an Internet-based paradigm through which a provider can deliver various services (e.g., servers, virtual machines, applications) for multiple tenants (or customers) [1, 2]. A provider can automatically scale and share these services across its tenants to reduce resources and operational costs. A tenant can subscribe to the services without having to worry about their implementation or the underlying resources. Three main layers make up cloud architecture : infrastructure-as-a-service (IaaS), platform-as-a-service (PaaS) and software-as-a-service (SaaS) [1, 2]. IaaS refers to infrastructure resources such as virtual machine (VM) and virtual private cloud (VPC) offered via the Internet. PaaS is the middle layer and refers to the operating system (OS), database management system (DBMS), web server and environment (that makes it easier to develop and to deploy cloud-based applications). SaaS is the top layer and refers to service-oriented web application running in a cloud

environment [2, 106, 110].

Security-wise, the SaaS layer is more open to end-users (that can be malicious) than the PaaS or IaaS layers (often accessible only to administrators and developers). The SaaS layer is thus the most vulnerable with 39% of attacks (followed by the PaaS layer) [4]. SaaS have increasingly fallen victim to attacks such as SQL injection (SQLIA) and cross-site scripting (XSS) [22,123]. IaaS providers such as Amazon Web Services (AWS) focus on preventing network attacks such as IP spoofing and port scans, but they do not offer protection against application-layer attacks [4]. Consequently, SaaS providers are responsible for protecting the multiple instances of their applications/services and ensuring the confidentiality, integrity and availability of their tenants' data [4, 21, 22, 48, 111, 123, 124]. This paper aims to improve the security of SaaS by allowing providers to integrate and share across their tenants an intrusion detection system (IDS) as a service. An IDS can detect malicious or unauthorized activities by monitoring networks, VMs, systems or applications [125].

Multi-tenancy and outsourcing offer benefits to both provider and tenant, but many security challenges must be addressed to protect a SaaS. As a consequence of outsourcing, tenants have no control over the source code and the underlying resources. Therefore, they are limited in their ability to deploy their preferred IDS. A provider should be able to easily integrate several IDS as a service into its cloud infrastructure. Furthermore, a tenant should be able to customize a multi-tenant IDS because security requirements vary from one tenant to the other because of differing degrees of data sensibility, budget, etc. [21, 22, 36, 124]. In addition, the security requirements of one tenant might be in conflict with those of another [124]. For example, entering SQL keywords via input fields on web pages could be accepted by one tenant and considered malicious by another. Protecting all tenants with the same way could increase the false-positive rate. For instance, a tenant might consider a certain number of requests per second to be normal behavior whereas another would not.

On the one hand, a number of application-layer IDS have been put forth to detect SQLIAs and XSS attacks [25, 33, 113, 126–131]. These application-layer IDS have not been integrated into the cloud despite some having good detection quality. On the other hand, several IDS have been deployed in cloud environment, but they focus on IaaS [73–83, 85–88, 132, 133]. These cloud-based IDS analyze the headers of packets to detect network attacks such as denial-of-service (DoS) and distributed denial-of-service (DDoS) attacks. They however fail to detect application-layer attacks such as SQLIAs and XSS attacks because they do not deal with HTTP or SQL traffic. This paper is interested in these application-layer attacks. It proposes and evaluates a framework, called MTIDaaS, to detect these attacks in the context of public cloud. Its main contributions can be summarized as follows :

— Integrating application-layer IDS as SecaaS for both providers and tenants of SaaS in a public cloud.
— Allowing a SaaS provider to share the integrated IDS across several tenants.
— Allowing each tenant to subscribe, customize and use this multi-tenant IDS on-demand while paying according to traffic and security requirements.

The key challenges in building a multi-tenant intrusion detection framework as a service for SaaS (MTIDaaS) differ from those of existing IaaS-based IDS : (1) The SaaS provider has the complicated task of splitting the cost of the MTIDaaS fairly between its tenants with differing traffic and security requirements ; (2) Offering fine-grained security monitoring and intrusion detection, at the tenant level, increases in complexity when tenants share tables in a database given the low degree of tenant isolation in SaaS ; (3) The MTIDaaS should adapt in real time to the kinds of traffic/attacks supported by the integrated IDS and the configuration/requirements of each tenant. For example, it should analyze only the traffic of tenants for whom the detection service is currently activated ; (4) The implementation of the MTIDaaS should ensure a high level of portability, cost-effectiveness and flexibility to increase its usability. High portability allows the provider to integrate the MTIDaaS without changing the source code of the applications or services. High cost-effectiveness allows it to significantly reduce MTIDaaS costs. High flexibility allows a tenant to use the integrated IDS as its own ; (5) The SaaS provider should identify the tenants who consume more MTIDaaS resources than others to react and protect the MTIDaaS from these malicious or expensive tenants (e.g., send an alert, charge additional fees).

The paper is organized as follows : Section 6.2 introduces SaaS architecture and a real-world scenario that motivated our research. Section 6.3 presents a state of the art of research on security issues of web application and cloud-based IDS. Sections 6.4 and 6.5 are devoted to MTIDaaS and its evaluation. Section 6.6 discusses the advantages and limitations of MTIDaaS and its comparison with other SecaaS approaches. Section 6.7 concludes the paper.

6.2 Background and motivation

A software-as-a-service or SaaS can be defined as a service-oriented application developed and deployed by a third party or provider in a cloud environment [1, 2]. To reduce the cost of infrastructure and maintenance, a SaaS provider shares one or more application and database instance(s) across multiple tenants [2, 106, 110]. A tenant subscribes to and customizes the SaaS via its own domain name (e.g., www.tenant1.SaaSApp.com) while paying according to the purchased product edition (e.g., standard, professional or enterprise) and/or user accounts. As shown in Fig. 6.1, SaaS architecture consists of three layers : the web application,

the web services (e.g., Representational State Transfer or REST, Simple Object Access Protocol or SOAP) and the database [2,106,110]. The web (e.g., Internet Information Services or IIS, Apache) and database (e.g., Azure SQL, Amazon Relational Database Service or AWS RDS) servers act as the PaaS of a SaaS and are installed on VMs, which form the IaaS level. To ensure high availability, load balancers can be installed at the application and/or service levels to distribute traffic over several VMs.

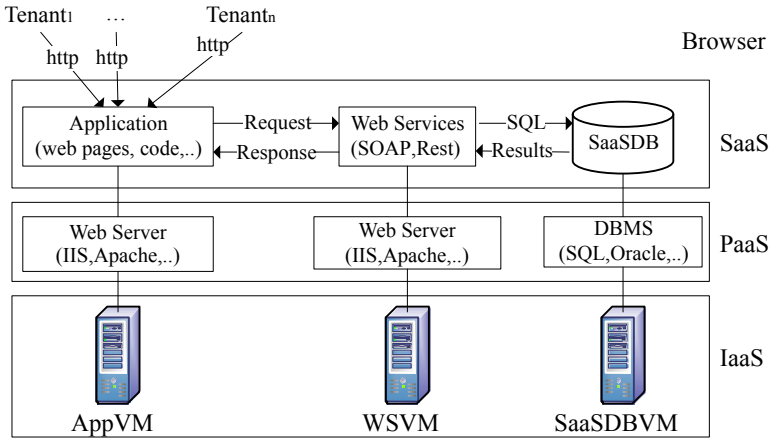

Figure 6.1 Architecture of SaaS

There are three models designed to manage the data of multiple SaaS tenants [106, 110] : (1) A database per tenant, (2) a schema per tenant and (3) shared tables. In the database-per-tenant model, the provider dedicates a separate database for each tenant. Thus, the tenant's data is isolated from others tenants' data. However, the cost of infrastructure and maintenance is high because the provider needs to manage and reserve resources for each tenant's database. In the schema-per-tenant model, the provider creates and manages a set of tables (schema) for each tenant within the same shared database. A SaaS provider can reduce the cost of resources using this model, but it reduces tenant isolation. Data management and development is also complex since a set of tables must be created for each new tenant. In the shared-tables model, the provider shares some tables within a single database across multiple tenants. Each shared table must have a specific field (e.g., TenantID) which contains a tenant's ID. This model is characterized by its cost-effectiveness and can be easily implemented by filtering by TenantID in the source code. However, the isolation level between tenants' data is low.

Security-wise, a SaaS can fall victim to application-layer attacks [123]. According to the Open Web Application Security Project (OWASP) [52], the top 10 web application security risks

in 2017 were : (A1) SQLIAs, i.e. when an attacker injects malicious SQL code or command as part of a SQL query of application or service to compromise the data ; (A2) an attacker accessing the passwords, keys, or session of other users when the authentication and session are incorrectly implemented ; (A3) an attacker stealing the sensitive data (credit card) of web service/application when it is not correctly protected (e.g., encryption) ; (A4) an attacker realizing XML DoS (XDoS) using the external entities if they are not validated by XML processors ; (A5) an attacker accessing unauthorized functionality and/or data if the restrictions/privileges are not properly enforced ; (A6) an attacker compromising the application, service and data by exploiting security misconfiguration of open cloud storage and HTTP/SOAP headers ; (A7) an attacker performing a XSS by executing a malicious script (HTML or JavaScript) in the browser of the victim to hijack user sessions or redirect the user to malicious sites ; (A8) an insecure deserialization leading to the execution of malicious code ; (A9) an attacker exploiting the known vulnerabilities of components used (e.g., libraries, frameworks) ; (A10) an attacker extracting or destroying data because of insufficient application or services monitoring. The preceding attacks are more consequential on SaaS than the traditional (on premise) model [4, 22, 123]. The data privacy of all tenants can be violated if an attacker compromises a shared database. The cost and limitation of resources transform traditional XDoS into an economic denial of sustainability (EDoS) that targets the financial resources and the trust between a provider and its tenants [134].

The integration of cloud-based IDS depends on tenant isolation and the control of tenants on the shared resources, which vary according to the cloud layer (e.g., IaaS or SaaS). Indeed, an IaaS tenant has a full control over its infrastructure resources (e.g., VMs, virtual networks). The degree of isolation between the IaaS tenants is high, since they share the physical servers. In contrast, SaaS tenants lose some control, since they accept that their provider will manage and store their data with that of other tenants. The SaaS layer has a lower degree of tenant isolation than the IaaS layer because the data of several tenants can be stored in the same tables within a single database. Consequently, the loss of control and the low degree of tenant isolation can cause privacy issues for a SaaS tenant even before the integration of a multi-tenant SaaS-based IDS.

Motivating scenario : Assume that a SaaS provider protects its web services against XDoS. This level of security can be considered as a baseline since the provider should ensure a high level of service availability : the unavailability of a service impacts all tenants. This provider can also provide and share additional IDS as a service against SQLIAs and XSS for its tenants. However, each tenant can use this shared IDS according to its own security requirements and budget.

6.3 State of art

An IDS monitors and analyzes the events or traffic on a system or network in order to detect and log intrusions (malicious or abnormal) [94, 125]. An intrusion can be defined as a set of actions which can compromise the integrity, confidentiality or availability of resources (e.g., service, VM, database). An IDS can be evaluated according to metrics such as detection rate, false-positive (normal behavior classified as intrusions) and false-negative (intrusions classified as normal behavior) rates. There are two main intrusion detection approaches which can be combined to improve the detection quality of an IDS : signature and anomaly. The signature-based approach is to build and maintain a database of signatures of known attacks : if an activity matches a signature, it is considered to be an attack. This approach produces low rates of false positives when the signature is sufficiently accurate. However, it can generate a high rate of false negatives because it detects only known attacks. The anomaly-based approach has two phases : learning and detection. In the learning phase, the normal behavior of the target entity (e.g., network, host, application, service) is defined by observing its normal activities (e.g., CPU, memory usages) during a set period of time. The detection phase triggers an alert when current activity deviates from the (previously generated) norm. This approach can detect unknown attacks. However, it can generate a high rate of false positives because it is hard to accurately generate normal behavior.

6.3.1 Web service/application countermeasures

Several countermeasures have been proposed to detect the attacks that target web applications/services. This section briefly introduces SQLIAs, XSS and XDoS, which are the most common attacks that target the SaaS layer [123], alongside some of their countermeasures.

A SQLIA occurs when an attacker injects SQL keywords (e.g., AND, OR) and special characters (e.g., –, ') through the input fields of web pages [25, 126, 127]. If the validation is found lacking, the attacker can thus generate (at runtime) SQL queries that are different from those intended by developers. There are numerous SQLIA countermeasures based on the static and/or dynamic analysis of HTTP requests and/or SQL queries [25, 113, 126–130]. The authors of [126] combined the analysis of HTTP and SQL traffic to reduce detection errors and false positives. In [25], a library is proposed to validate a SQL query against attack patterns after generating its syntax tree. In [127], an automaton is generated for each SQL query by analyzing the source code of a PHP web application : a SQL query is considered as a SQLIA (at runtime) if it does not conform to the generated automaton. SQLCheck [128] verifies the conformity of SQL queries according to models of legal queries. SQLProb [113] uses

a learning-based approach that validates user input within the generated query to learn the normal query structure. CANDID [129] compares syntax trees for each dynamic SQL query with and without user input : any discrepancy indicates an attack. In [130], a method that combines the lexical and syntax analysis of SQL queries has been proposed to detect SQLIA in CC, but there is not enough information (e.g., cloud platform, evaluation environment) to evaluate its relevance to CC.

A XSS exploits improper validation and/or encoding input that is embedded in the response data of web applications [131]. This vulnerability allows an attacker to inject a malicious script (e.g., JavaScript, VBScript) which seems to have originated from a trusted site using the victim's browser. The execution of this script leads to unauthorized access to cookies or sessions. In [131], a XSS signature-based detection captures all network packets and validates relevant packets according to predefined Snort security rules. Microsoft has been proposed an Anti XSS library that developers can invoke in their application/service source code [33].

A DoS leads to resources being unavailable for legitimate users [27,134,135]. A XML DoS (XDoS) occurs when an attacker sends a SOAP message with nested/cyclic tags (coercive parsing) or extremely large amounts of data (oversize payload) [134]. The web server can break down when the XMLParser processes the message without proper validation. In CC, several detection/prevention techniques such as [135] and [27] deal with XDoS. In [135], a XML vulnerability detection system checks each SOAP request against different DoS patterns. If a malicious SOAP request is sent, the sender's IP address is blacklisted. The authors of [27] suggest observing attack symptoms in SOAP-XML messages (e.g., high number of nested tags) and high usage of CPU to confirm the presence of a XDoS. Doing so prevents XDoS from XML messages with a dynamic number of nested tags and random frequency.

6.3.2 Cloud-based IDS

Existing cloud-based IDS can be divided into four categories according to where they are deployed and/or the type of traffic they monitor : Virtual Machine Manager (VMM), network, host/VM and collaborative [123, 125, 136].

An IDS can be integrated at the VMM level. To protect tenants' VMs against attacks such as privilege escalation and the exploitation of security tools, a security architecture that combines policy access control and IDS has been implemented using Xen [73]. A VM-Introspection-based IDS [74] has been proposed to detect attacks at the process and system-call levels (e.g., disabling security tools). It is placed at Dom0 of a VMM and extracts normal and attack traces using n-grams and term frequency-inverse document frequency approaches. However, a VMM-based IDS does not analyze HTTP or SQL traffic, which are at the appli-

cation level. In addition, a SaaS provider cannot access the VMM to integrate an application IDS.

A network IDS (NIDS) is deployed at network entry points and inspects the headers of packets to detect/prevent DoS, port scanning and spoofing attacks. In [75], security architecture is proposed for IaaS providers and their tenants : the provider can protect its own cloud infrastructure from the malicious tenants and a tenant can protect its VMs against insider attacks (e.g., DoS, malicious software, rootkits) coming from other tenants or administrators. Signature Snort-based NIDS [76] have been proposed for IaaS models to monitor and log suspicious network activities that target the VMs of tenants. The authors of [77] have proposed customized network security which routes the packets through multiple network firewalls using route tables. To prevent external and internal network attacks, these packets are inspected according to customized security rules. A Snort-based NIDS [78] integrated at the network layer makes it possible to generate new rules. However, these cloud NIDS are unable to analyze the HTTP requests and SQL queries sent as encrypted data in packets.

A host IDS (HIDS) monitors and analyzes the inbound/outbound traffic of tenants' VMs. The HIDS can inspect encrypted traffic on the VM, but it can fall victim to a DoS. A hybrid (signature-based and anomaly-based) NIDS [79] has been deployed on each VM to detect network attacks (e.g., DoS, DDoS) using Snort and many classifiers (e.g., associative rule and decision tree). An anomaly-based IDS [80] has been proposed to detect malicious programs on cloud VMs using Immediate System Calls : the normal system calls are generated during the training phase and considered as references to detect malicious programs during the detection phase. Known distributed system/network intrusions are detected by deploying the signatures of Snort on each cloud VM [81]. In [82], a HIDS is installed between the router and the cloud host to analyze the TCP/IP of packets on each VM. This analysis is performed according to generic cloud signatures (cloud data sets) prepared manually and based on opened ports (TCP/IP) of the cloud. A hybrid IDS [83] creates a profile (e.g., CPU, memory) for each VM, monitors all incoming packets to compare the data rate with the threshold and periodically checks the vulnerability. However, this IDS has neither been implemented nor evaluated. A HIDS [132] has been proposed to detect stealthy DoS, that can keep malicious behaviors undetectable through traditional mechanisms. To ensure service availability, the auto-scalability of the cloud is used to allocate additional resources. It is deployed on each VM on the MOSAIC private cloud. The preceding cloud HIDS monitor VMs to detect malicious system calls [80], network intrusions (TCP/IP protocols) [79, 81, 82] or abnormal usage of resources [83, 132]. Thus, they do not deal with HTTP/SQL traffic, as required to detect application-layer attacks.

A collaborative IDS (CIDS) is composed of several IDS installed on different cloud layers, virtual networks or VMs. In [85], a NIDS (Snort) is deployed on a virtual switch to monitor the incoming network packets and a HIDS (open-source tool) is deployed on each host VM. A central coordinator collects the alerts from the NIDS and HIDS. In [86], a high-level collaborative IDS divides the infrastructure (IaaS) into network, host and global logical layers. The routers (with NIDS) and physical hosts (with HIDS) of each IaaS provider are grouped into collaborative clusters. A NIDS collaborative framework [87] has been proposed for CC. Each NIDS is installed in a cluster and combines the signatures of Snort with an anomaly-based decision tree classifier to improve detection accuracy. Then, a correlation unit receives the alerts from all NIDS to prevent coordinated DDoS. A cooperative IDS [88] has been proposed and evaluated in the OpenStack framework to optimize the detection delay of DDoS. A manager agent is used to correlate the alerts received from different cloud layers. A hybrid IDS [133] combines a signature-based detection with an anomaly-based detection to defend against known and unknown attacks. It is evaluated in Virtualbox based on network, host and web server IDS. However, the preceding cloud CIDS have single points of failure, are not evaluated in public clouds, and do not deal with HTTP requests and SQL queries.

On the one hand, the preceding application-layer IDS/countermeasures (Section 6.3.1) may have good detection quality (e.g., high detection rates, low false-positive and false-negative rates) since they have been improved over time. However, none of these countermeasures are integrated into a public cloud. On the other hand, the cloud-based IDS listed above (Section 6.3.2) do not detect the attacks that target a SaaS. According to the Cloud Security Alliance (CSA) [37], SecaaS solutions should satisfy the main characteristics of CC (on-demand self-service, pay-per-use, scalability, resource pooling and high availability). A few cloud-based IDS [75, 76] meet certain SecaaS properties, but they focus on the IaaS layer and are offered to IaaS providers and tenants of VMs. Hence, we intend to fill the gap by integrating one or many IDS as a service for both SaaS providers and tenants. We also observed that the concept of multi-tenant IDS has not been sufficiently explored even for the IaaS layer.

6.4 MTIDaaS : Multi-tenant intrusion detection framework as a service for SaaS

Our main contribution is the integration as a service at the SaaS layer of IDS regardless of their detection approaches (signature-based or anomaly-based). A host or VM based IDS is appropriate to SaaS because it can monitor encrypted traffic when installed on each web services' VM (WSVM). The MTIDaaS can be considered as a VM-based IDS since two of its modules are installed on each WSVM to extract the HTTP requests and SQL queries.

However, the analysis of HTTP/SQL to detect many kinds of attacks (e.g., SQLIA, XSS) can increase CPU use on a WSVM to the point that it can break down. Hence, the MTIDaaS analyzes the HTTP/SQL on separate VMs to reduce the charge on WSVMs. The MTIDaaS is deployed on multiple VMs to ensure high availability and auto-scaling.

MTIDaaS involves the following four entities : (1) The IaaS provider (e.g., AWS) that offers the IaaS for both SaaS and MTIDaaS providers ; (2) The security third party that offers the MTIDaaS for SaaS providers as pre-built VM images (in a public cloud) ; (3) The SaaS provider that integrates the MTIDaaS into its infrastructure so it can be used by its tenants ; (4) The SaaS tenant that can subscribe, unsubscribe, customize, start and stop the MTIDaaS's detection service via web pages. The SaaS and MTIDaaS providers are tenants of the same IaaS provider.

A SecaaS solution should satisfy on-demand self-service, scalability, resource pooling, pay-per-use and high availability [1, 37]. For the provider, a scalability is achieved in the architecture and integration of the MTIDaaS in a public cloud. A resource pooling is achieved by sharing the resources across multiple subscribed tenants of the MTIDaaS (multi-tenancy). A pay-per-use is achieved by evaluating the total infrastructure cost of the MTIDaaS. The cloud integration of MTIDaaS aims to ensure high availability (fault tolerance). The duplication of VMs and auto-scaling can make the MTIDaaS available in case of VMs' failures. With the MTIDaaS, each subscribed tenant can select types of attacks and specify security requirements at any time. Additionally, it can use the MTIDaaS on-demand self-service and pay-per-use, which are achieved by grouping the tenants' traffic in real-time.

6.4.1 Architecture

The main modules of the MTIDaaS are : RequestAgent, SQLAgent, IDSModule and CostGenerator (Fig. 6.2). RequestAgent and SQLAgent analyze the SOAP/REST requests and SQL queries respectively. Then, they send the TenantID (TenID), Request (Req) and SQL to IDSModule (e.g., as parameters of a web service's request). The correlation between SOAP/-REST requests and SQL queries is performed during the analysis of SQL queries. The IDSModule is the integrated application IDS that stores the alerts in MTAlertDB and information related to tenant traffic in the IDS database (IDSDB). CostGenerator periodically consults the IDSDB to generate the infrastructure/resource costs for the provider and each subscribed tenant. In order to ensure high portability, we consider the application, service and database as black boxes which make our design vendor-agnostic. Hence, we intercept, analyze and correlate the SOAP/REST requests (that target the services) and the SQL queries (generated by the services) on each WSVM without modifying the application/service source code.

This decouples the framework from the development environment/programming language and DBMS/database structure. Cost-effectiveness is achieved by sharing the same instance of multi-tenant IDS across multiple tenants instead of dedicating IDS VMs for each tenant.

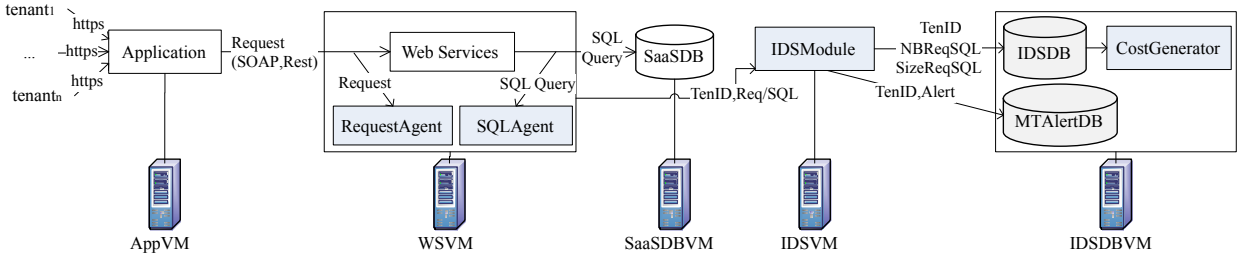

Figure 6.2 Architecture of MTIDaaS

In the context of MTIDaaS, the IDS, tenant, request and SQL query are defined as follows :

— An IDS is a tuple (Request, SQL, Type, Phase, Attacks), where Request and SQL are Boolean values to indicate whether the IDS analyzes or not the SOAP/Rest or SQL traffic. The type is signature-based or anomaly-based. The phase is training or detection if the type is anomaly. Attacks is the list of attacks detected by the IDS.

— A tenant is a tuple (TenID, Name, Domain, IsSubscribed, IDSStarted, Attacks), where TenID, Name and Domain are the identifier, name and web domain (e.g., www.tenant1.SaaSApp.com) of the tenant. IsSubscribed and IDSStarted are boolean values to indicate whether the tenant subscribes to the MTIDaaS's detection service and starts it. Attacks is the list of attacks specified by the tenant.

— A request (Req), generated by the SaaS to call a web service, is a tuple (TenID, UserID, TimeStamp, Params), where TenID and UserID are the identifiers of tenant and its user that executed the request. TimeStamp is the time of occurrence of request. Params=$\{param_1, param_2, \ldots\}$ is the list of parameters used in the web service's request.

— A SQL query (SQL), generated by web service (WS) following a SOAP/Rest call, is a tuple (TenID, TimeStamp, Type, SQLStat, Literals), where TenID is the identifier of tenant which executes the SQL. TimeStamp is the time of occurrence of SQL. Type is Insert, Update, Delete or Select. SQLStat is the SQL statement. Literals $=\{lieral_1, literal_2, \ldots\}$ is the list of literals, where a literal is any part of a SQL statement that is neither an identifier (keyword, table name, field name,...) nor an operator of the SQL language.

RequestAgent

RequestAgent is installed on each web services' VM (WSVM), where the requests are decrypted, to listen to the TCP/HTTP port instead of being implemented in the service source code. Thus, it can intercept and analyze the packets of HTTP requests (Algorithm 4). The TenID is extracted from the header of request (Step 1) and the tenant (object) of (TenID) is returned (Step 2). If the tenant is subscribed to and starts the detection, the parameters (Params) are extracted by parsing the request (Steps 3-4). If the IDS analyzes the HTTP traffic, the (Req) object is generated and sent to IDSModule ; If it analyzes the HTTP and SQL traffic, a threshold is calculated as the average length of the request's parameters (Steps 6-11). Coefficient c is adjusted empirically. After a number of experiments using different values for c, it was found that a value of $c = 0.6$ provides the best quality of correlation. (Req) is generated and added into SetReqs (the list of Request candidates). The threshold and SetReqs are used during the correlation between HTTP requests and SQL queries.

Algorithm 4 RequestAnalysis : generates and sends a Request

Input : $RequestMessage$ ▷ Extracted from data of packet
Input : $TimeStamp$ ▷ packet occurrence time
Input : IDS ▷ Integrated IDS
Input : $SetReqs$ ▷ Set of Requests used for Request/SQL correlation
Output : Req

1: $TenID =\leftarrow ExtractTenID(RequestMessage)$
2: $Tenant \leftarrow GetTenant(TenID)$
3: **if** ($Tenant.IsSubscribed$ **and** $Tenant.IDSStarted$) **then**
4: $Params \leftarrow ParseRequest(RequestMessage)$
5: **if** ($IDS.SQL == false$) **then** ▷ IDS deals only with HTTP
6: $Req \leftarrow (TenID, Params, TimeStamp)$
7: $SendToIDSModule(TenID, Req)$
8: **else** ▷ IDS deals with HTTP(SOAP/Rest) and SQL traffic
9: $Threshold \leftarrow \frac{c}{L} \times \sum_{i=1}^{L} Param_i.length$
10: $Req \leftarrow (TenID, Params, Threshold, TimeStamp)$
11: $SetReqs.Add(Req)$ ▷ Add Req into SetReqs
12: **end if**
13: **end if**

SQLAgent

SQLAgent is installed on each WSVM and listens to the TCP/SQL port to intercept the SQL queries' packets and then extract the SQLStat and Timestamp of each SQL query, if

the integrated IDS deals with SQL traffic (Algorithm 5). The correlation between HTTP and SQL traffic is required to identify the request (Req) that generates each SQL query for the IDS (e.g., [126]) that validates the both requests and SQL queries, for example, to reduce false positives. This correlation is used by the IDS (e.g., [25,113,127–129]) that inspects only the SQL queries to identify the source of the attacks (e.g., UserID, TenantID, web pages, web session, IP address). Thus, provider administrators can take many actions (e.g., blocking malicious web sessions, IPs). Algorithm 5 works as follows :

Steps 2-3 : The TenID is extracted from each SQL query according to the used data model. (1) For database-per-tenant, it is extracted from the connection String. (2) For schema-per-tenant, it is extracted from the tables' name of Query. For example, the query (`Select * from Orders1`) is generated for tenant1. (3) For shared-tables, it is identified by analyzing the query. For example, the query (`Select * from Orders1 Where (TenantID=1)`) is generated for tenant1. The TenID, type (Insert, Update, Delete or Select) and Literals are identified by analyzing the SQL statement (SQLStat).

Steps 4-17 : if the tenant is subscribed to and starts the detection, the Req which generates the SQL is identified by calculating the mapping scores between the SQL and certain candidates requests (SetReqs). A candidate Req must have the same TenID of SQL and occur before the SQL for a period of $2 \times t$ milliseconds, where t is the required time (average) to generate the SQL queries for each request (Req). It is multiplied by 2 to cover all possible (candidate) requests. The initial value of t is 1 second and decreases according to the performance of WSVM. This optimization reduces significantly the number of calculated mapping scores (see next paragraph). As a result, the Req which has the minimum mapping score with SQL is considered as the generator of SQL. Then, the (TenID,Req,SQL) tuple is generated and sent to IDSModule.

Algorithm 5 SQLAnalysis : generates and sends a SQL

Input : $SQLStat$ ▷ Extracted from data of packet
Input : $TimeStamp$ ▷ packet occurrence time
Input : IDS ▷ Integrated IDS
Input : t ▷ the time (average) required for the generation of SQL Query
Input : $SetReqs = \{Req_1, Req_2, .., Req_K\}$
Output : SQL

if $(IDS.SQL == true)$ **then**
  $(TenID, Type, Literals) \leftarrow AnalyseSQL(SQLStat)$
  $Tenant \leftarrow GetTenant(TenID)$
  if $(Tenant.IsSubscribed$ **and** $Tenant.IDSStarted)$ **then**
    for $i = 1$ to K **do**
      if $(SQL.TenID = Req_i.TenID$ **and**
$SQL.Timestamp > Req_i.Timestamp$ **and** $SQL.Timestamp$ - $Req_i.Timestamp <= 2 \times t)$
then
        $Score \leftarrow MappScore(Req_i.Params, SQL.Literals)$
        if $(Score <= Req_i.Threshold)$ **then**
          $MinScore \leftarrow Score$
          $Req \leftarrow Req_i$
        end if
      end if
    end for
    $SQL \leftarrow (TenID, Type, SQLStat, TimeStamp)$
    $SendToIDSModule(TenID, Req, SQL)$
  end if
end if

MappScore : The mapping score quantifies the similarity between the parameters of HTTP (SOAP/Rest) request (Req) and literals of SQL query (SQL). Consider Req.Params=$\{Param_1, \ldots, Param_L\}$ and SQL.Literals=$\{Literal_1, \ldots, Literal_M\}$. The similarity between each ($Param_i$ and $Literal_j$) strings is evaluated using string distance computation algorithm [137], which calculates the number of modifications to transform a string to another one. Further, a parameter is used in the web service source code for generating a literal of SQL query. Thus, the minimal string distance between each parameter $Param_i$ of Req and all the literals of SQL is computed in order to identify in which literal this parameter is used. The mapping score between a HTTP and a SQL is evaluated as the average, over all the parameters of the HTTP, of the minimum distances :

$$MappScore(Req, SQL) = \frac{1}{L} \sum_{i=1}^{L} \min_{j=1}^{M} Dist(Param_i, Litteral_j)$$

In order to illustrate the calculation of mapping scores and the selection of Req that generated

each SQL, we define the correlation matrix between the SQLs and Reqs of a tenant occurred during $2 \times t$ on a WSVM as follows :

$$\mathbf{S} = \begin{array}{c} \\ Req_1 \\ . \\ Req_i \\ . \\ Req_K \end{array} \begin{array}{c} \begin{array}{ccccccc} SQL_1 & SQL_2 & . & SQL_j & . & SQL_N \end{array} \\ \begin{bmatrix} s_{11} & s_{12} & . & s_{1j} & . & s_{1N} \\ . & . & . & . & . & . \\ s_{i1} & s_{i2} & . & s_{ij} & . & s_{iN} \\ . & . & . & . & . & . \\ s_{K1} & s_{K2} & . & s_{Kj} & . & s_{KN} \end{bmatrix} \end{array}$$

Where s_{ij} is the mapping score between the Req_i and SQL_j, $MappScore(Req_i, SQL_j)$, defined in Equation 1. Each SQL is associated to the Req with the minimal score.

Illustrative example : Consider the following SQL queries in the code source of LogIn and Order web services :

- LogIn.asmx : `SQL="Select * from Users Where TenantID =" + pTenantID + " and User='" + pUser.text + "' and Password='" + pPass.text + "' "` It allows the authentication of a tenant user.
- Order.asmx : `SQL="Select * from Orders Where TenantID ="+ pTenantID + " and Num >"+ pNum.text + " and Supp like '"+substring(pSupp.text,1,4)+"%' and Date>'" + pDate.text + "' "`
  It gets the orders of a tenant which are ordered from some suppliers after a specifc date.

Let us consider the following scenario (at run-time) :

- User1 of tenant1 enters (`Admin' OR 1=1 -`) and (`123`) in user and password to authenticate.
- User1 of tenant2 enters (`Jean`) and (`Jacques`) in user and password to authenticate.
- User2 of tenant2 enters (2), (`Costco`) and (`12/01/2017`) in number, supplier and date to get orders.

The following Reqs/SQLs are generated during $(2 \times t)$:

- Req1 (LogIn) : `(1, Admin' OR 1=1 -, 0x202CB..)`,
  where `0x202CB..` is the `HASHBYTES(MD5,123)`
- Req2 (LogIn) : `(2, Jean,0xF2942..)`,
  where `0xF2942..` is the `HASHBYTES(MD5,Jacques)`
- Req3 (Order) : `(1, 2,Costco, 12/01/2017)`
- SQL1 :`Select * from Users Where TenantID=1 and User='Admin' OR 1=1 -' and Password='0x202CB..'`

— SQL2 : `Select * from Users Where TenantID=2 and User='Jean' and Password='0xF2942..'`

— SQL3 : `Select * from Orders Where TenantID=1 and Num>2 and Supp='Cost%' and Date>'12/01/2017'`

The following correlation matrix resumes the calculation of mapping scores between each (Req, SQL).

$$\mathbf{S} = \begin{array}{c} \\ Req_1 \\ Req_2 \\ Req_3 \end{array} \begin{array}{c} \begin{array}{ccc} SQL_1 & SQL_2 & SQL_3 \end{array} \\ \begin{bmatrix} \mathbf{0} & 9.5 & 9.5 \\ 5 & \mathbf{0} & 5 \\ 5.6 & 7.33 & \mathbf{0.7} \end{bmatrix} \end{array}$$

For example, $MappScore(Req_2, SQL_2)$=0. $EditDistance(param_1, literal_1)$= $EditDistance('jean','jean')$=0 means that the $param_1$ is used without modification in the first literal of SQL query.

IDSModule

This module represents the IDS that analyzes the HTTP and/or SQL traffic. Generally, it can be an application IDS, detection algorithm or a combination of multiple detection entities. It should be modified to receive the TenID with each (Req, SQL) pair to distinguish between tenants. A profile (e.g., attack types, selected rules) is created for each subscribed tenant. Thus, each tenant can customize the IDS by modifying its profile. To meet the pay-per-use property, IDSModule regularly sends the number and size of Req/SQL to the IDSDB to generate the tenants' bills. Technically, this IDS is a pre-built VM with the detection entity. IDSModule is deployed on separate VMs to make it easier to integrate the IDS with high scalability and availability. Algorithm 6 summarizes the general functionality of IDSModule.

Algorithm 6 General functionality of IDSModule

Input : $TenID$
Input : SQL
Input : Req
Input : IDS

```
1: Tenant ← GetTenant(TenID)
2: for each AttackID ∈ Tenant.Attacks do
3:     if (IDS.SQL == true) then
4:         if IDS.IsAttack(AttackID, Req, SQL) then
5:             Alert=GenerateAlert(AttackID,Req,SQL)
6:             MTAlertDB.Add(TenID,Alert)
7:         end if
8:     else
9:         if IDS.IsAttack(AttackID, Req) then
10:            Alert=GenerateAlert(AttackID,Req)
11:            MTAlertDB.Add(TenID,Alert)
12:        end if
13:    end if
14: end for
```

CostGenerator

This module is responsible for the generation of provider and tenant costs (sections 6.4.2 and 6.4.3) every hour. The provider cost model computes the infrastructure cost of MTIDaaS (payed by the provider). The tenant cost model estimates the bill of MTIDaaS for each tenant. To reduce the number of VMs, this simple module is deployed on the same VM of IDSDB and MTAlertDB databases (IDSDBVM).

6.4.2 Provider cost model

An IaaS provider offers on-demand infrastructure services billed by the hour [138]. To compute the infrastructure cost of the MTIDaaS for one hour (or part of an hour), we propose a SaaS provider cost model based on the one provided by an IaaS provider such as AWS. This cost, denoted as CostMTIDaaS, is the sum of several costs (Equation 6.1).

$$CostMTIDaaS = CostAgents + CostDetect + CostLB + CostDB \tag{6.1}$$

— CostAgents is the cost of RequestAgent and SQLAgent deployed as a unique software entity on each WSVM. It is expressed by Equation 6.2 :

$$CostAgents = \sum_{i=1}^{I}(CPUPerc_i \times CostWSVM_i) \tag{6.2}$$

Where I is the total number of WSVMs, $CPUPerc_i$ is the percentage of CPU (average) used by RequestAgent and SQLAgent on $WSVM_i$ and $CostWSVM_i$ is the hourly cost of $WSVM_i$ which depends on the number of vCPU cores.

— CostDetect is the total cost of all IDSVMs on which the IDSModule is installed. It is expressed by Equation 6.2 :

$$CostDetect = \sum_{j=1}^{J}(CostIDSVM_j + CostIDSVMStor_j) \tag{6.3}$$

Where J is the total number of IDSVMs, $CostIDSVM_j$ is the hourly cost of $IDSVM_j$ and $CostIDSVMStor_j$ is the storage cost of $IDSVM_j$.

— CostLB is the cost of IDSLB load balancer which distributes the (TenID,Req/SQL) over all IDSVMs. It is expressed by Equation 6.4 :

$$CostLB = CostLBUsage + CostLBData \tag{6.4}$$

Where CostLBUsage is the fixed cost of IDSLB (for one hour) and CostLBData is charged according to the total size of distributed data during one hour.

— CostDB is the hourly cost of IDSDBVM which contains the alerts (generated by all IDSModules), IDS usage of all tenants and CostGenerator module. It is expressed by Equation 6.5 :

$$CostDB = CostIDSDBVM + CostIDSDBVMStor \tag{6.5}$$

Where $CostIDSDBVM$ and $CostIDSDBVMStor$ are the hourly usage and storage costs of $IDSDBVM$, respectively.

6.4.3 Tenant cost model

Dividing costs between tenants is an open research question and there is no set tenant cost model for SaaS [139,140]. We resolve this issue for the MTIDaaS by monitoring the traffic of each tenant. This monitoring can also identify the tenants that use the MTIDaaS the most. Each tenant's bill is calculated for each hour according to Equation 6.6.

$$\begin{aligned} CostMTIDaaS(TenID) = CostAgents(TenID) + CostDetect(TenID) + \\ CostLB(TenID) + CostDB(TenID) \end{aligned} \tag{6.6}$$

— CostAgents(TenID) : is the cost of RequestAgent and SQLAgent for (TenID) expressed by Equation 6.7 :

$$CostAgents(TenID) = CostAgents \times \frac{NBReqSQL(TenID)}{NBReqSQL} \tag{6.7}$$

Where CostAgents is computed by Equation 6.2, NBReqSQL(TenID) is the total number of (TenID,Req/SQL) and NBReqSQL is the total number of all (TenID,Req/SQL).

— CostDetect(TenID) : is the most important to be divided between tenants since the CostDetect (Equation 6.2) is the highest cost of MTIDaaS (the cost of all IDSVMs). Not only the number/size of Req and SQL but the security requirements of each tenant should also be considered. However, the evaluation of used resources to process each (TenID, Req/SQL) for each attack type is costly. Hence, we decide to make a trade-off between the cost-effectiveness and the accuracy by associating a detection cost to each attack type. This attack cost depends on the quantity of signatures (signature-based IDS) or complexity of anomaly detection (anomaly-based IDS) and is evaluated, for example, during a training phase or every hour by Equation 6.8 :

$$CostAttack(AttackID) = CostDetect \times \frac{ExecTime(AttackID)}{ExecTimeAllAttacks} \tag{6.8}$$

Where AttackID is the identifier of attack, ExecTime(AttackID) is the average execution time on all IDSVMs to detect AttackID and ExecTimeAllAttacks is the total of the average execution time on all IDSVMs to detect all attacks. The CostDetect of (TenID) is expressed by Equation 6.9 :

$$CostDetect(TenID) = NBReqSQL(TenID) \times \sum_{AttackID=1}^{N} Subscribe(TenID, AttackID) \times \frac{CostAttack(AttackID)}{NBReqSQL(AttackID)} \tag{6.9}$$

Where NBReqSQL(TenID) is the total of (Req/SQL) for TenID. Subscribe(TenID,$Attack_n$) is equal to 1 if the (TenID) is subscribed to ($Attack_n$) ; otherwise it is equal to 0. CostAttack($Attack_n$) is calculated by Equation 6.8 and ReqSQL($Attack_n$) is the total of (TenID,Req/SQL) validated against $Attack_n$ for all tenants.

— CostLB(TenID) is expressed by Equation 6.10 :

$$CostLB(TenID) = CostLB \times \frac{SizeReqSQL(TenID)}{SizeReqSQL} \tag{6.10}$$

Where CostLB is the total cost of IDSLB (Equation 6.4), SizeReqSQL(TenID) is the total size of (TenID,Req/SQL) for TenID and SizeReqSQL is the total size of (TenID,Req/SQL) for all tenants.

— CostDB(TenID) is expressed by Equation 6.11 :

$$CostDB(TenID) = CostDB \times \frac{NBAlert(TenID)}{NBAlert} \tag{6.11}$$

Where CostDB is the hourly cost of IDSDBVM (Equation 6.5), NBAlert (TenID) is the total number of TenID's alerts and NBAlert is the total alerts of all tenants.

6.5 Implementation and evaluation

This section presents the implementation and evaluation of the MTIDaaS.

6.5.1 Implementation

Fig. 6.3 shows the SaaS and the MTIDaaS being deployed in a VPC of the AWS public cloud. The SaaS is deployed in a public subnet (SaaSSnet) and made of an internet gateway, a load balancer (WSLB), an application VM (AppVM), WSVMs and a database's VM (SaaSDBVM). The Internet gateway allows the VPC to communicate with the Internet. An instance of the SaaS application is installed on AppVM. The web services are installed on WSVM1, WSVM2, WSVM3 and WSVM4, and the WSLB distributes the requests across them.

The MTIDaaS is developed as pre-built VMs and deployed as follows. Multiple IDSVMs can be created in two private subnets (IDSSnet1 and IDSSnet2) configured in two availability zones (unique physical locations). The creation of IDSVMs in multiple availability zones protects the MTIDaaS from data center failures and thus improves its availability. An IDSDBVM contains the MTAlertDB and IDSDB databases. RequestAgent and SQLAgent are implemented as a single C# application (WSApp) and installed on each WSVM. WSApp listens to traffic on the TCP/HTTP (e.g., 8080 for SOAP) and TCP/database (e.g., 1443 for MS SQL) ports and then extracts (TenID, Req/SQL) tuples using the WinPcap API [120]. IDSModule is implemented as a Web service (IDSWS) in C#. The IDSWS is installed on each IDSVM and invoked in WSApp. A load balancer (IDSLB) distributes the web service requests, which call the IDSWS with TenID and Req/SQL as parameters, over the IDSVMs. CostGenerator is developed as a stored procedure in the IDSDB to calculate the provider and tenant costs every hour. Table 6.1 describes the VMs of SaaS and MTIDaaS in AWS [138].

For instance, IDSModule offers SQLIAIDS [19] and AntiXSS [33] as two intrusion detection approaches. SQLIAIDS is an anomaly-based IDS that analyzes both HTTP requests and SQL queries to detect SQLIAs. We adapted this IDS to SaaS to analyze SOAP/REST instead of HTTP requests. In the learning phase, the templates of normal SQL queries are generated

in IDSDB (installed on IDSDBVM) for each web service. In the detection phase, the SQL queries are compared with generated templates to detect SQLIAs and store the corresponding alerts in MTAlertDB (installed on IDSDBVM). The AntiXSS library uses white listing to provide protection against XSS attacks. It defines a valid or allowable set of characters to detect untrusted Java scripts and XML content.

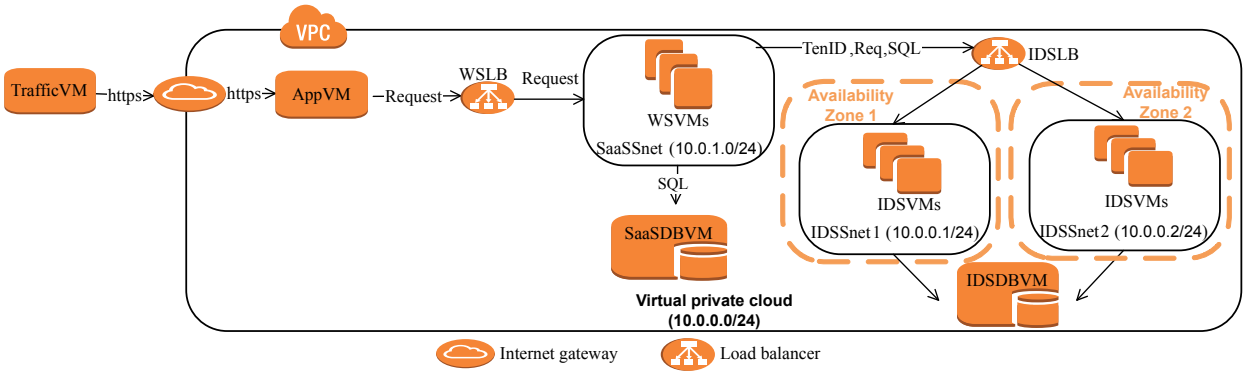

Figure 6.3 Deployment of a SaaS and MTIDaaS in AWS

Table 6.1 Description of Different VMs

VM	Type	vCPU	RAM (GiB)	Installation	Price (Hour)
AppVM	t2.2xlarge	8	32	Application	0.4332
WSVM	t2.large	2	8	Web Services RequestAgent SQLAgent	0.1208
SaaSDBVM	t2.medium	2	4	SaaSDB	0.0644
IDSVM	t2.medium	1	2	IDSModule	0.0644
IDSDBVM	t2.micro	1	1	IDSDB MTAlertDB CostGenerator	0.0162

6.5.2 Evaluation

The scalability, overhead, impact on response time, detection quality, and costs (for the provider and tenants) of MTIDaaS are evaluated for a vulnerable SaaS application by generating normal and malicious POST HTTP requests.

Application : Hacmebank [119] is used as a vulnerable (to SQLIAs) service-oriented (SOAP) web application. It is a real-world banking online application implemented using .NET and Microsoft SQL Server. We have extended this application by adding TenantID columns on

each database's table (shared-tables data model). The TenantID is added to the parameters to support several tenant domains. The application, web services and database are deployed on respectively AppVM, four WSVMs and SaaSDBVM.

Scenarios : An IDS is evaluated according to the traffic generated from the signatures of attacks or using vulnerability exploitation tools [94]. We used vulnerability exploitation tools to evaluate not only the detection quality, but also the overhead of MTIDaaS on all VMs (in real time). To this end, we generate normal and malicious (with SQLIA and XSS attacks) HTTP requests. The SQLIAs are generated using SQLMap [103] as an open-source penetration testing tool. SQLMap generates SQLIAs by combining its mechanisms with other exploitation tools (e.g., Metasploit). The XSS are generated using XSSless [141], an automated open-source XSS payload generator. The malicious Java scripts are injected into HTTP parameters.

SQLMap and XSSless commands are configured with the domain name of each tenant. We implement (in C#) a multi-threaded application (TrafficApp) to target the web domain of a tenant with a big quantity of POST HTTP requests. The TrafficApp is installed on a (c4.2xlarge) AWS VM (TrafficVM). Two threads (ThrSQLIA and ThrXSS) generate the HTTP requests of a tenant with SQLIAs and XSS by executing its SQLMap and XSSless commands. A thread (ThrXDoS) generates parameters with XML bomb attacks [142]. Another thread (ThrNormal) generates a random quantity of normal HTTP requests (without malicious java scripts, special characters/SQL keywords or XML bomb attacks). TrafficApp is executed 25 times to generate 1,435,226 mixed (normal and malicious) HTTP POST requests for 25 tenants under the two following scenarios :

Scenario 1 : The WSApp (RequestAgent and SQLAgent) is deactivated and the MTIDaaS VMs (IDSVM1 and IDSDBVM) are stopped. In the threads, a random quantity of requests is configured and memorized for each tenant.

Scenario 2 : The WSApp is activated on each WSVM. The MTIDaaS VMs (IDSVM1 and IDSDBVM) are started. The security requirements of tenants (subscription to attacks) are configured and the same HTTP requests as in scenario 1 are generated. To prevent XDoS (XML bombs), we used a Microsoft .NET mechanism [142] to parse and validate the SOAP requests.

Scalability

To ensure automatic and dynamic scalability, we configure an auto-scaling group of IDSVMs as a target of the IDSLB. The minimum and desired number of IDSVM is one. The number

of minutes between two auto-scaling activities is five minutes. If the CPU usage on existing IDSVMs is greater than 55% for three successive minutes, a new IDSVM is launched from an Amazon Machine Image (AMI) pre-configured with the IDSWS. If it is smaller than 25% for five successive minutes, an IDSVM is terminated. This horizontal scalability reduces the cost of the MTIDaaS and ensures its availability even during the auto-scaling process.

Overhead

The overhead of the MTIDaaS is evaluated according to the CPU and memory usages on the SaaS and MTIDaaS VMs monitored using CloudWatch (a AWS monitoring and management service). Figures 6.4.a and 6.4.b show the percentage of CPU usage on SaaS VMs under scenarios 1 and 2 respectively. The activation of RequestAgent and SQLAgent introduces an additional overhead (averaged over all WSVMs) of 5.1% CPU and insignificant memory (0.2%). This overhead is low considering the large quantity of HTTP requests.

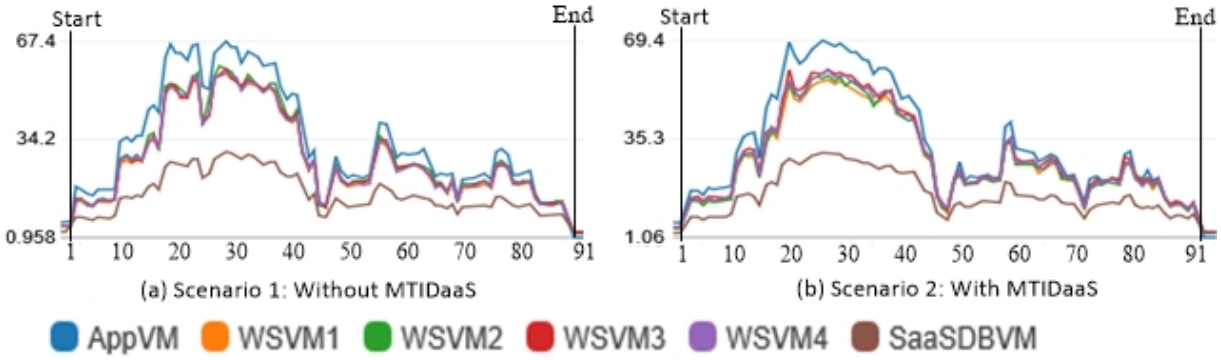

Figure 6.4 CPU usage on SaaS VMs during the 2 scenarios : x-axis (the traffic period) and y-axis (CPU percentage)

Fig. 6.5 shows the percentage of CPU and memory usages on the MTIDaaS VMs to analyze 1,435,226 HTTP requests and detect 379,565 SQLIAs and 410,917 XSS. Fig. 6.5.a demonstrates that the IDSVM2 is running between the 19th and 57rd minutes of the traffic period due to the auto-scaling group, and the maximum CPU usage is 57.8% and 37.2% on IDSVM1 and IDSVM2, respectively.

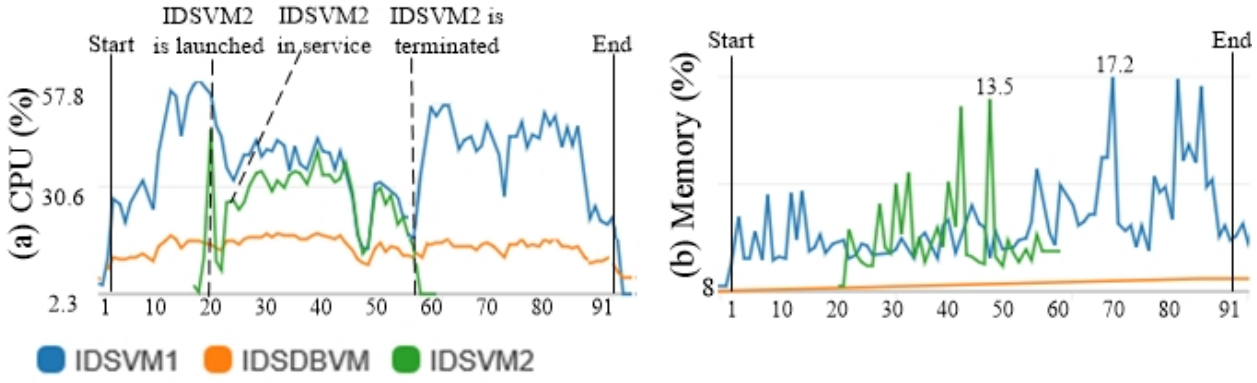

Figure 6.5 CPU and memory usages on MTIDaaS VMs in scenario 2 : x-axis (the traffic period) and y-axis (CPU/memory)

To compare SQLIAIDS and AntiXSS, scenario 2 is re-executed twice : (1) Only the SQLIAIDS is activated and (2) Only the AntiXSS is activated. According to Table 6.2, maximum CPU and memory usages for SQLIAIDS are greater than those for AntiXSS on IDSVM1. This gap of overhead is tied to the detection approaches of SQLIAIDS and AntiXSS. Furthermore, unlike AntiXSS that validates only the HTTP requests against the signatures of a library, SQLIAIDS is an anomaly-based IDS that analyzes the HTTP requests and SQL queries by communicating with the database of normal SQL models. The difference of overhead between SQLIAIDS (51.2% of CPU), AntiXSS (39.7% of CPU) and their combination (57.8% of CPU) is small because IDSModule is implemented as a single function of a web service that must be called in the three cases. The combination of SQLIAIDS and AntiXSS (See Fig. 6.5) nonetheless requires the creation of IDSVM2 that is running only for 38 minutes.

Table 6.2 Comparison of SQLIAIDS and AntiXSS

IDS	CPU IDSVM1	Memory IDSVM1	TP	FN	FP	DR	PR
SQLIAIDS	51.2%	16.9%	379,565	9,034	11,960	0.977	0.969
AntiXSS	39.7%	11.8%	410,917	20,963	1,084	0.951	0.997

Response time

To evaluate the impact of the MTIDaaS on HTTP response time, we used the loadImpact [122] tool since it can perform a deep performance evaluation of web applications (e.g., response time) by simulating multiples web sessions and virtual users (VUs). We executed the same test scenario (30 VUs, 25 minutes) with and without the MTIDaaS. The application

processes 81,576 HTTP requests with the MTIDaaS, and 81,985 without. The average HTTP response time is 236 ms with the MTIDaaS and 235 ms without. Hence, the integration of the MTIDaaS decreases by 409 the number of HTTP requests processed and increases by 1 ms the average response time.

Detection quality

The detection quality of an IDS depends on true positives (TP), false positives (FP), false negatives (FN), the detection rate (DR) and the precision rate (PR) [94]. In the context of the MTIDaaS, TP is the number of malicious HTTP requests considered as attacks by the integrated IDS. FP is the number of normal HTTP requests considered as attacks by the integrated IDS. FN is the number of malicious HTTP requests considered as normal by the integrated IDS. DR and PR equal `TP/(TP + FN)` and `TP/(TP+FP)` respectively. Table 6.2 shows the TP, FN, FP, DR and PR values for SQLIAIDS and AntiXSS. The detection rate of SQLIAIDS is better than AntiXSS, which must update its signatures periodically. However, the precision of AntiXSS is better than SQLIAIDS, which is an anomaly-based IDS. In general, these detection and precision rates are acceptable considering that SQLMap and XSSless generate advanced SQLIAs using evasion techniques and a large list of XSS malicious scripts, respectively. For example, the detection rates of Amnesia [127] and SQLProb [113] are respectively 1 and between 0.98 and 1.

Costs/Pay-per-use

The infrastructure cost of the MTIDaaS is computed according to Equation 6.1 and Table 6.1. In our experiment, this cost is equal to 0.34 USD/2 hours, where the average CPU usage of the MTIDaaS on the WSVMs is about 5.1%, the hourly cost of the IDSLB is 0.025 USD, and the hourly storage cost of each VM is 0.0028 USD (for 80 GBi). Table 6.4 presents the subscriptions, HTTP requests, SQLIA, XSS and bills (calculated according to Equation 6.6) of tenants. Note that the first six tenants are configured with different and comparable requirements and amounts of traffic. The average execution times to detect SQLIAs and XSS are respectively 2.56 ms and 0.78 ms. As shown in Table 6.4, each tenant's bill depends on the amount of traffic it generates and its subscription. According to Table 6.4, although the number of requests and attacks for tenant2 and tenant3 are similar, tenant2 pays more than tenant3 because the detection of SQLIA costs more than XSS. Tenant1's bill is about 4 times larger than that of tenant4. These results also demonstrate the cost-effectiveness of the MTIDaaS due to its scalability and multi-tenancy : (1) The cost of the MTIDaaS is 0.4072 USD without auto-scaling (two IDSVMs must run for two hours) and 0.34 USD with

auto-scaling (IDSVM1 is running for two hours, and IDSVM2 for one) ; (2) Each tenant should pay about 0.34 USD/2 hours (at least) to lease a separate MTIDaaS VMs. Note that the combination and installation of many IDS (i.e., SQLIAIDS, AntiXSS) on the same IDSVM (instead of being installed on separate IDSVMs) reduces the global cost of the MTIDSaaS.

Table 6.3 Subscription, Number of HTTP Requests, Number of Attacks and Bills of Tenants

Tenant	Subscription	HTTP requests	SQLIA	XSS	Cost Agents	Cost Detect	Cost LB	Cost DB	Bill (USD/2 hours)
1	SQLIA,XSS	206,550	63,450	68,223	0.0073	0.0336	0.0072	0.0055	0.0374
2	XSS	144,600	N/A	47,349	0.0051	0.0054	0.005	0.0038	0.0167
3	SQLIA	140,400	42,980	N/A	0.0049	0.0175	0.0049	0.0037	0.0226
4	SQLIA,XSS	51,392	16,041	16,615	0.0018	0.0084	0.0018	0.0014	0.0094
5	XSS	50,203	N/A	15,898	0.0018	0.0018	0.0017	0.0013	0.0057
6	SQLIA	45,320	14,024	N/A	0.0016	0.0057	0.0016	0.0012	0.0074
7-25	SQLIA,XSS	796,761	243,070	262,832	0.028	0.1291	0.0278	0.0211	0.1437
Total		1,435,226	379,565	410,917	0.0505	0.2015	0.05	0.038	0.34

6.6 Discussion

In this section, we first discuss the advantages and limitations of MTIDaaS. Then, we compare MTIDaaS with other SecaaS approaches.

6.6.1 Advantages and limitations of the MTIDaaS

The MTIDaaS provides SecaaS for SaaS providers. (1) It is on-demand self-service since it is offered as pre-built VM images from which a SaaS provider can create its own VMs. (2) It is a scalable service : RequestAgent and SQLAgent are deployed on WSVMs which systematically scale up or down ; IDSModule is deployed on several IDSVMs which can scale according to the amount of traffic. (3) Its resources are shared across multiple tenants to ensure a high cost-effectiveness (resource pooling). (4) Its cost is evaluated using a proposed mathematical model (pay-per-use). (5) It can work even during an IDSVM breakdown (high availability).

The MTIDaaS ensures a high level of portability. (1) Although the MTIDaaS is evaluated in AWS, it can be deployed on other clouds. For example, it can be offered as Azure VMs. (2) The correlation between requests and SQL queries is performed without modifying the application, web services or database. (3) MTIDaaS works according to database-per-tenant, schema-per-tenant and shared-tables data models (See Algorithm 5). (4) It works correctly with both HTTP and HTTPS (Hyper Text Transfer Protocol Secure) because the requests and queries are extracted on web services' VMs when the data is in clear. Additionally, the

experimental results prove that the MTIDaaS can be easily integrated in a public cloud with minimum impacts on the SaaS VMs (e.g., CPU and memory usages) and HTTP response time.

A SaaS provider can offer the MTIDaaS as SecaaS (on-demand self-service and pay-per-use) for its tenants in a flexible way. Indeed, a tenant can subscribe, unsubscribe, start and stop the detection of specific attacks on-demand without the provider having to intervene. The experimental results prove that the cost payed by each tenant is related to the quantity of HTTP requests and the types of attacks (pay-per-use).

In addition to saving on detection costs, a SaaS provider can apply some common prevention mechanisms (e.g., blacklisting) following the detection of attacks that target each tenant. Consider the following practical examples : (1) Inbound security rules with malicious IP addresses (extracted from malicious HTTP requests) can be added automatically to the network access control list (ACL) of the VPC using AWS SDK ; (2) An HTTP proxy can be installed on AppVM to reject the requests coming from malicious users without analyzing the SQL queries. These malicious IP addresses and users are identified thanks to the correlation between HTTP requests and SQL queries. As a result, all tenants that share MTIDaaS can benefit from these prevention mechanisms.

Although the MTIDaaS focuses on the integration of IDS as a service into the SaaS layer, it can be adapted to detect the attacks that target other cloud layers. Furthermore, a NIDS (e.g., the signatures of Snort) can be integrated as a part of IDSModule because the MTIDaaS can provide the information (e.g., IP source, IP destination and occurrence time of packets) required for the detection of network intrusions (e.g., DDoS).

The developers and administrators of SaaS can apply the proposed best practices to eliminate some application security risks of section 6.2 [52]. For example, they can force the use of strong passwords to prevent attackers from compromising password and keys (A2), and they can encrypt the data and enforce transport layer security (TLS) for all web pages to prevent the sensitive data exposure (A3). In our evaluation, three IDS techniques are implemented to prevent XDoS (as baseline security), and detect SQLIAs and XSS (as additional security services), which are the most common attacks that target the SaaS layer [123]. However, other application attacks such as external entity (XXE) attack are also exploited by malicious users through the HTTP requests and/or SQL queries provided by the MTIDaaS in real time. Thus, the corresponding IDS can be implemented as a part of the MTIDaaS to detect these attacks.

6.6.2 Comparison of MTIDaaS with other approaches

Table 6.4 compares the MTIDaaS with other SecaaS approaches. In [75], a hybrid (signature and anomaly) SecaaS architecture is implemented as a part of Xen VMM to protect the infrastructure of the provider and the VMs of the tenants against malicious tenants (IaaS layer). The scalability and pay-per-use are not considered in this architecture ; however, its availability is high since there is no single point of failure. In IDSaaS [76], a Snort-based NIDS and an application are deployed in public and private subnets of AWS, respectively. The requests and responses are routed through a network address translation (NAT) VM and IDS VMs, which leads to large increases of the response time. The scalability of IDSaaS is low because only one instance of NIDS can be deployed. Its pay-per-use is not demonstrated because the cost payed by a customer is not evaluated. Its availability is low because the NAT and NIDS VMs are single points of failure. To identify the vulnerabilities of cloud applications, a security diagnosis as a service framework (SDaaS) [143] analyzes the application source code without dealing with IDS. The scalability and availability of SDaaS are practically demonstrated by configuring an auto-scale group of EC2 instances to maintain a threshold of response time. Compared to previous SecaaS approaches, unlike SDaaS which is limited to a single programming language (Java), the portability of the MTIDaaS is high because it does not access the source code of applications or services. The high scalability of MTIDaaS is practically demonstrated by our experiment results (Section 6.5.2). Its pay-per-use is demonstrated for both SaaS provider and tenant thanks to our proposed cost models. The MTIDaaS has a high availability because it contains no single point of failure.

Table 6.4 Comparison of MTIDaaS with Other Approaches

Criteria \ Approach	[75]	IDSaaS [76]	SDaaS [143]	MTIDaaS
Cloud layer	IaaS	IaaS	SaaS	SaaS
Deployment	VMM	VMs of public subnet	VMs	VMs of private subnet
Detection approach	Hybrid	Signature	N/A	Hybrid
Portability	Not considered	Public or private subnets	Not considered	Programming language, database
Security-as-a-service	For IaaS provider	For customers of IaaS	For SaaS provider	For SaaS provider and tenant
Scalability	Not considered	Low	High	High
Pay-per-use	Not considered	Not demonstrated	Not considered	Demonstrated for SaaS provider and tenant
Availability	High	Low	High	High

6.7 Conclusion

In SaaS, the classic web application and service attacks have more critical consequences because any security breach negatively affects all tenants and the reputation of cloud service providers. They can discourage tenants from subscribing to SaaS services. With the MTIDaaS, a SaaS provider can convince its tenants that their data is protected according to their individual security requirements. The MTIDaaS provides a portable and flexible security-as-a-service IDS for both SaaS provider and tenant. It is shared by all subscribed tenants to minimize the cost of resources. Research is ongoing to improve the accuracy of the tenant cost model by taking into consideration not only the quantity, but also the quality of the HTTP requests of each tenant. Interesting future work is allowing tenants to share the alerts and collaborate to improve the quality of detection.

CHAPITRE 7 DISCUSSION GÉNÉRALE

Dans ce chapitre, nous présentons la synthèse de nos travaux de recherche. Ensuite, nous résumons l'évaluation de nos cadres infonuagiques proposés.

7.1 Synthèse des travaux

Nos trois cadres traitent les questions de recherche posées dans le chapitre d'introduction.

— Le premier cadre, appelé SQLIIDaaS (for SQL Injection Intrusion Detection framework as a Service for SaaS providers), permet aux fournisseurs SaaS d'intégrer des contremesures des SQLIA en tant que service de sécurité. SQLIIDaaS s'adapte à la contremesure intégrée (flexible) et n'exige aucune modification ou analyse du code source (portable). Cette portabilité est assurée en considérant l'application SaaS comme une boîte noire et en corrélant les entrées (requêtes HTTP) et sorties (requêtes SQL) de cette application.

— Le deuxième cadre, appelé ITADP (for Inter-Tenant Attack Detection and Prevention framework for multi-tenant SaaS), permet aux fournisseurs SaaS de détecter et de prévenir les attaques entre les locataires qui partagent certaines tables d'une base de données relationnelle. Afin d'offrir une protection additionnelle aux niveaux HTTP et réseaux virtuelles, les mécanismes de "black listing" et de vérification HTTP sont appliqués.

— Le troisième cadre, appelé MTIDaaS (for Multi-Tenant Intrusion Detection framework as a Service for SaaS), offre une solution de détection d'intrusion en tant que service aux fournisseurs (libre-service, élasticité des ressources, mutualisation des ressources, paiement à l'usage et haute disponibilité) et locataires (libre-service et paiement à l'usage) d'un SaaS. Grâce à MTIDaaS, le fournisseur peut intégrer des contremesures détectant différentes attaques applicatives tout en répondant aux exigences de sécurité de chaque locataire. Les ressources (VMs, répartiteurs de charge, etc.) de MTIDaaS sont partagées entre les différents locataires. Le fournisseur et chaque locataire payent le service MTIDaaS à l'usage grâce à nos deux modèles de coût. Ce cadre est protégé comme suit : (i) il est déployé dans un réseau privé et ne possède pas donc une adresse IP publique qui peut être utilisée par des attaquants pour cibler les machines virtuelles (IDSVMs). De plus, il est uniquement accessible par les administrateurs des locataires ; (ii) il fournit le nombre de requêtes et le temps moyen de l'analyse des requêtes (à chaque minute) pour chacun des locataires sur toutes les IDSVMs. Cela

protège le cadre contre un locataire malintentionné qui génère une quantité massive de requêtes dans le but d'inonder le cadre ou d'invoquer des ressources additionnelles. Le fournisseur peut donc saisir des règles de protection afin de réagir contre ces locataires considérés comme coûteux ou malintentionnés. Par exemple, si un locataire génère un nombre de requêtes plus grand qu'un seuil (prédéfini) pendant un nombre de minutes (prédéfini), le fournisseur peut, par exemple, dédier des machines virtuelles (IDSVMs) à ce locataire ou arrêter l'analyse de ses requêtes.

7.2 Évaluation et résultats obtenus

Nous avons opté pour une méthode expérimentale d'évaluation en déployant les trois cadres proposés et certaines applications SaaS dans un VPC de l'infonuagique publique AWS. Nous avons utilisé des outils d'exploration des vulnérabilités et d'évaluation de la performance des applications Web pour générer des requêtes HTTP malveillantes et normales. Cela a permis d'analyser le comportement des applications et VMs en temps réel avant et après l'intégration de chaque cadre proposé.

7.2.1 Évaluation quantitative

La qualité de détection des cadres proposés est mesurée en fonction des principales métriques d'évaluation des IDS telles que les vrais positifs (TP), les faux positifs (FP), les faux négatifs (FN), les taux de détection (DR) et les taux de précision (PR). Pour chacun de ces trois cadres, nous avons effectué deux scénarios de test (avec ou sans chacun de ceux-ci) afin d'évaluer son impact sur le temps de réponse (aux requêtes HTTP) et son utilisation de processeur et/ou mémoire. De plus, nous avons fait une quantification des frais (en USD) d'ITADP et de MTIDaaS.

Pour SQLIIDaaS, l'environnement d'évaluation est composé de deux applications Web (chaque application est déployée sur une VM), d'une NATVM vers laquelle les requêtes sont acheminées et d'une VM (IDSVM) qui contient la contremesure intégrée. Les TP, FN, FP, DR et PR obtenus sont respectivement 44456, 1207, 0, 0.9735, et 1. L'intégration de SQLIIDaaS a fait diminuer de 31485 à 26738 le nombre des requêtes HTTP (générées pendant 5 minutes). Cette intégration a introduit une utilisation additionnelle de 5.61% de processeur et 0.4% de mémoire sur la NATVM.

ITADP est évalué en fonction de deux SaaS multi-locataires (MTSaaS1 et MTSaaS2) où les tables de données sont partagées par plusieurs locataires. ITADP est déployé sur deux VMs (ITADVM1 et ITADVM2). Les TP, FN, FP, DR, PR obtenus sont respectivement

entre 10336, 486, 791, 0.954, 0.929 et 53928, 1031, 1801, 0.981, 0.967, selon le mode de fonctionnement (détection ou prévention) et l'application testée (MTSaaS1 ou MTSaaS2). Cette qualité de détection est efficace surtout avec la génération d'une grande quantité des requêtes HTTP normales et malveillantes. Cependant le taux de faux positifs est supérieur à celui de faux négatifs. Cela est justifié par le fait que ITADP considère une requête SQL (par exemple, exécutée par le tenant d'identifiant eTenID) comme une attaque entre locataire dès qu'il trouve une condition satisfaisant les identifiants d'autres locataires (e.g., TenantID <eTenID) sans poursuivre l'analyse du reste de l'arbre de syntaxe de la clause Where. Bien que cette optimisation accélère considérablement le temps d'analyse des requêtes SQL, elle augmente le nombre de faux positifs. L'intégration d'ITADP en mode prévention a diminué de 37468 à 35815 le nombre des HTTP traitées par MTSaaS2 (pendant 5 minutes).

MTIDaaS est évalué en fonction d'un SaaS multi-locataire en intégrant deux IDS (SQLIAIDS et AntiXSS) pour détecter les XSS et SQLIAs. Les TP, FN, FP, DR et PR obtenus sont respectivement 379565, 9034, 11960, 0.977 et 0.969 (pour SQLIAIDS), et 410917, 20963, 1084, 0.951 et 0.997 (pour AntiXSS). L'intégration de MTIDaaS a diminué de 81985 à 81576 le nombre des requêtes HTTP (générées pendant 25 minutes). Cette intégration a introduit une utilisation additionnelle moyenne de 5.1% de processeur et 0.2% de mémoire sur les VMs contenant les services Web. Les qualités de détection de SQLIIDaaS et MTIDaaS sont similaires et liées aux IDS intégrés plus que l'architecture ou mécanismes d'intégration proposés. L'impact de MTIDaaS sur le temps de réponse (aux requêtes HTTP) est moins grave que celui de SQLIIDaaS. Cependant, les trois cadres ont ajouté une utilisation similaire et raisonnable de processeur et mémoire sur les VMs contenant les applications et services Web.

7.2.2 Évaluation qualitative

Une évaluation qualitative de SQLIIDaaS et MTIDaaS est réalisée par rapport à la portabilité, la flexibilité, l'élasticité des ressources, la mutualisation des ressources, le paiement à l'usage et la haute disponibilité.

Portabilité :

SQLIIDaaS et MTIDaaS ont le même degré de portabilité par rapport aux langages de programmation, DBMS et environnements infonuagiques. Cependant, ITADP est moins portable que SQLIIDaaS et MTIDaaS. Dans son mode de prévention, un service doit être invoqué dans le code source pour pouvoir rejeter les requêtes SQL considérées comme des attaques entre locataires.

Flexibilité :

SQLIIDaaS offre une flexibilité au fournisseur qui sera capable d'intégrer des techniques de détection des injections SQL. Cependant, MTIDaaS offre une flexibilité au fournisseur et au locataire. Le fournisseur peut combiner plusieurs contremesures au sein d'un même IDS. Le locataire peut sélectionner certains types d'attaques selon son budget et l'importance de ses données.

Élasticité des ressources :

Une élasticité des ressources est partiellement assurée dans SQLIIDaaS à cause de NATVM qui peut être mis à l'échelle uniquement en changeant sa capacité de processeur et mémoire (scale-up and scale down en anglais). Cette élasticité nécessite une interruption de service et complique l'approvisionnement automatique et dynamique des NATVMs selon l'évolution du trafic. Une élasticité automatique et dynamique des ressources de MTIDaaS est concrètement prouvée avec l'augmentation et la réduction du nombre des VMs en fonction du pourcentage d'utilisation du processeur. Cela permet de réduire le coût de la détection.

Mutualisation des ressources :

SQLIIDaaS mutualise ses ressources entre des applications Web indépendantes. Cependant, MTIDaaS permet au fournisseur de maximiser son profit en mutualisant ses ressources entre les différents locataires du même SaaS. Par exemple, un locataire, qui utilise le plus MTIDaaS, a réduit de 0.34 à 0.0374 USD/2 heures le coût en partageant l'infrastructure du MTIDaaS avec les autres locataires.

Paiement à l'usage :

SQLIIDaaS est compatible avec la propriété de paiement à l'usage puisqu'il est développé sous forme d'images des VMs. Néanmoins, dans MTIDaaS, un paiement à l'usage est montré en quantifiant le coût payé par le fournisseur et celui payé par chaque locataire dépendamment de sa quantité de trafic et les techniques de détection choisies.

Haute disponibilité :

SQLIIDaaS achemine les requêtes HTTP et SQL via NATVM ; par conséquent, elle représente un point de défaillance unique. Dans MTIDaaS, cette faiblesse est surmontée en associant les requêtes SQL aux HTTP correspondantes sur les VMs hébergeant les services Web.

Selon notre revue de littérature, très peu d'articles scientifiques proposent des solutions de sécurité en tant que service. Ces solutions n'offrent pas d'IDS pour les fournisseurs et les locataires des SaaS. Le troisième article détaille une comparaison entre MTIDaaS et ces solutions en fonction de portabilité, élasticité des ressources, mutualisation des ressources, paiement à l'usage et haute disponibilité. Suite à cette comparaison, nous constatons que MTIDaaS assure une élasticité des ressources et haute disponibilité comme [75,143]. Contrairement aux autres solutions, nos résultats d'expérimentation ont montré la portabilité, la mutualisation des ressources et le paiement à l'usage de MTSaaS.

CHAPITRE 8 CONCLUSION

Ce chapitre récapitule les grandes lignes de notre projet de recherche, à savoir les avantages, les limites ainsi que les améliorations futures des cadres proposés dans cette thèse.

8.1 Avantages des cadres proposés

La majorité des travaux de recherche portant sur la sécurité dans le contexte d'infonuagique, se focalisent sur la protection d'IaaS. Ainsi, cette thèse propose des solutions aux problèmes de sécurité de la couche SaaS. Les cadres proposés dans cette thèse permettent de détecter et de prévenir les attaques entre les locataires qui représentent le problème de sécurité le plus critique au niveau des SaaS multi-locataire. La résolution de ce problème a un impact positif sur l'adoption des services infonuagiques puisqu'elle encourage les locataires à s'abonner aux SaaS. Ces cadres permettent aussi au fournisseur SaaS la capacité d'intégrer et de mutualiser un IDS qui combine plusieurs techniques de détection, en tant que service entre les différents locataires. De plus, chaque locataire est capable de personnaliser et de configurer cet IDS selon ses propres exigences de sécurité. Contrairement aux travaux de recherches existants, nos cadres fournissent des services de sécurité (libre-service, élasticité rapide, mutualisation des ressources, paiement à l'usage et haute disponibilité) pour les fournisseurs et les locataires avec un niveau élevé de portabilité et de flexibilité. À notre avis, cette thèse ouvre la voie à la mutualisation d'IDS entre les différents locataires d'un SaaS comme une nouvelle orientation de recherche.

Plusieurs défis ont été soulevés et surmontés lors de la conception des trois cadres :

— Dans SQLIIDaaS, la corrélation entre les requêtes HTTP et SQL fournit les informations nécessaires à l'intégration des IDS applicatifs sans modifier le code source d'application ou services Web.

— Dans ITADP, chaque requête SQL est analysée en temps quasi-réel grâce à nos algorithmes optimisés. Cela permet de réduire l'impact sur le temps de réponse surtout que toutes les requêtes SQL doivent être analysées. L'analyse de la clause Where d'une requête SQL, composée d'un nombre inconnu de conditions et d'expressions combinées, est une tâche complexe. Afin de résoudre cette complexité, nous avons (i) formalisé une grammaire SQL adaptée à une base de données dans laquelle plusieurs locataires partagent certaines tables ; (ii) généré l'arbre syntaxique binaire de chaque requête SQL ; et (iii) et analysé cet arbre selon certaines règles afin de vérifier si cette requête SQL exécutée par un locataire n'accède pas en lecture ou en écriture aux données des

autres locataires.

— Dans MTIDaaS, le coût est adéquatement et équitablement divisé entre les locataires selon les ressources consommées pour répondre aux exigences de chacun d'eux. MTIDaaS est multi-locataire et s'adapte automatiquement en fonction des types d'attaques détectées par l'IDS intégré et les exigences de sécurité de chaque locataire en temps réel. La surveillance continue du trafic de chaque locataire permet au fournisseur d'identifier les locataires potentiellement malveillants ou qui consument plus les ressources de MTIDaaS afin de prendre des décisions comme l'envoi des alertes d'avertissement et la facturation à l'intention des locataires malintentionnés.

8.2 Limitations et améliorations futures

Nos travaux de recherche ont certaines limites qui peuvent être résumées et améliorées comme suit :

— Ils se limitent à la détection des attaques réalisées par les usagers des locataires. Par exemple, ils ne détectent pas les attaques ou accès illégitimes réalisés par les administrateurs du fournisseur SaaS contre les locataires ;

— Ils ne prennent pas en considération un cas particulier où les locataires d'un SaaS peuvent échanger certaines informations. Nous pensons qu'un modèle de contrôle d'accès à base des attributs (ABAC for Attribute Based Access Control) pourrait être une solution efficace et flexible pour ces SaaS. Cette solution permettrait aux locataires (dépendants) de partager des informations en saisissant certaines règles de sécurité ;

— Le modèle du coût de locataire se base sur sa quantité du trafic. Afin d'optimiser la précision de ce modèle, la moyenne du temps d'analyse des requêtes de chaque locataire peut être évaluée périodiquement ;

— Les cadres infonuagiques proposés sont évalués en fonction des services Web SOAP et bases de données relationnelles. Il serait utile de tester ces cadres avec des services Web REST et bases de données NoSQL (for non relational SQL).

RÉFÉRENCES

[1] A. W. Sokol et M. D. Hogan, "Nist cloud computing standards roadmap," *Special Publication (NIST SP)-500-291r2.2013*, 2013.

[2] L. Wu, S. K. Garg, S. Versteeg et R. Buyya, "Sla-based resource provisioning for hosted software-as-a-service applications in cloud computing environments," *IEEE Transactions on services computing*, vol. 7, n°. 3, p. 465–485, 2014.

[3] D. Informatique, "Infonuagique : un marché en hausse de 23 https ://www.directioninformatique.com/infonuagique-un-marche-en-hausse-de-23/53999, 2018.

[4] S. Subashini et V. Kavitha, "A survey on security issues in service delivery models of cloud computing," *Journal of Network and Computer Applications*, vol. 34, n°. 1, p. 1–11, 2011.

[5] P. Mell et T. Grance, "The nist definition of cloud computing," *National Institute of Standards and Technology.Special Publication 800-145*, p. 1–7, 2011.

[6] systancia, "Le cloud computing," http ://www.systancia.com/fr/modeles-du-cloud-computing, 2014, consulté le 13 août 2014.

[7] H. Judith, B. Robin, K. Marcia et H. Fern, "Comparing public, private, and hybrid cloud computing options," http ://www.dummies.com/DummiesArticle/Comparing-Public-Private-and-Hybrid-Cloud-Computing-Options.id-147134.html, 2014, consulté le 3 août 2014.

[8] wikipedia, "Infrastructure as a service," http ://fr.wikipedia.org/wiki/Infrastructure_as_a_service, 2014, consulté le 22 août 2014.

[9] cloud360, "Get your head in the clouds," http ://www.clouds360.com/index.php, 2014, consulté le 13 juillet 2014.

[10] I. RightScale, "Rightscale releases 2015 state of the cloud report," http ://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2015-state-cloud-survey, 2015, consulté le 24 juillet 2015.

[11] W.-T. Tsai, X. Sun et J. Balasooriya, "Service-oriented cloud computing architecture," dans *Information Technology : New Generations (ITNG), 2010 Seventh International Conference on*. IEEE, 2010, p. 684–689.

[12] R. Krebs, C. Momm et S. Kounev, "Architectural concerns in multi-tenant saas applications." dans *CLOSER*, 2012, p. 426–431.

[13] J. Rengarajan, “Architecting a multi-tenant saas solution – why is it different ?” http ://blog.techcello.com/2013/04/architecting-a-multi-tenant-saas-solution-why-is-it-different/, 2014, consulté le 26 novembre 2014.

[14] A. Contributors, “Multitenant applications,” http ://-docs.autofac.org/en/latest/advanced/multitenant.html#what-is-multitenancy, 2014, consulté le 12 septembre 2014.

[15] H. He, “Applications deployment on the saas platform,” dans *Pervasive Computing and Applications (ICPCA), 2010 5th International Conference on*. IEEE, 2010, p. 232–237.

[16] S. Paliwal, “Cloud application services (saas)–multi-tenant data architecture,” *Infosys technologies limited*, 2012.

[17] i. salesforce.com, “The force.com multitenant architecture,” https ://developer.salesforce.com/page/Multi_Tenant_Architecture, 2014, consulté le 26 novembre 2014.

[18] H. Debar, “An introduction to intrusion-detection systems,” *IBM Research, Zurich Research Laboratory,Saumerstrasse 4, CH 8803 Ruschlikon*, p. 1–18, 2005.

[19] M. Yassin, “Protection automatique des applications Web contre l’attaque par injection SQL,” Mémoire de maîtrise, Université du Québec à Montréal, 2014.

[20] T. Grandison et E. Terzi, “Intrusion detection technology,” *IBM Almaden Research Center*, nº. September, p. pages 1–7, 2007.

[21] S. Iqbal, M. L. M. Kiah, N. B. Anuar, B. Daghighi, A. W. A. Wahab et S. Khan, “Service delivery models of cloud computing : security issues and open challenges,” *Security and Communication Networks*, 2016.

[22] P. K. Chouhan, F. Yao et S. Sezer, “Software as a service : Understanding security issues,” dans *Science and Information Conference (SAI), 2015*. IEEE, 2015, p. 162–170.

[23] F. Valeur, D. Mutz et G. Vigna, “A learning-based approach to the detection of sql attacks,” dans *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2005, p. 123–140.

[24] G. Vigna, F. Valeur, D. Balzarotti, W. Robertson, C. Kruegel et E. Kirda, “Reducing errors in the anomaly-based detection of web-based attacks through the combined analysis of web requests and sql queries,” *Journal of Computer Security*, vol. 17, nº. 3, p. 305–329, 2009.

[25] “General sql parser,” 2017, URL : http://www.sqlparser.com/sql-parser-dotnet.php [accessed : 2017-06-01].

[26] R. M. Sarhadi et V. Ghafori, “New approach to mitigate xml-dos and http-dos attacks for cloud computing,” *International Journal of Computer Applications*, vol. 72, n°. 16, p. 27–31, June 2013, published by Foundation of Computer Science, New York, USA.

[27] K. Gupta, R. R. Singh et M. Dixit, “Cross site scripting (xss) attack detection using intrustion detection system,” dans *Intelligent Computing and Control Systems (ICICCS), 2017 International Conference on.* IEEE, 2017, p. 199–203.

[28] W. G. J. Halfond et A. Orso, “Preventing SQL injection attacks using AMNESIA,” *Proceeding of the 28th international conference on Software engineering - ICSE ’06*, p. 1–4, 2006.

[29] Z. Su. et G. Wassermann, “SQLCheck :The Essence of Command Injection Attacks in Web Applications,” *Proceedings of the 5th international workshop on Software engineering and middleware - SEM 205*, n°. January, 2006.

[30] A. Liu, Y. Yuan, D. Wijesekera et A. Stavrou, “Sqlprob : a proxy-based architecture towards preventing sql injection attacks,” dans *Proceedings of the 2009 ACM symposium on Applied Computing.* ACM, 2009, p. 2054–2061.

[31] S. Bandhakavi, P. Bisht, P. Madhusudan et V. Venkatakrishnan, “Candid : Preventing sql injection attacks using Dynamic candidate evaluations,” *In Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, n°. October, p. pages 12–24, 2007.

[32] N. Gruschka et N. Luttenberger, “Protecting web services from dos attacks by soap message validation,” dans *IFIP International Information Security Conference.* Springer, 2006, p. 171–182.

[33] “Antixss 4.3.0,” 2018, URL : https://www.nuget.org/packages/AntiXSS/ [accessed : 2018-05-01].

[34] R. Rai, G. Sahoo et S. Mehfuz, “Securing software as a service model of cloud computing : Issues and solutions,” *arXiv preprint arXiv :1309.2426*, 2013.

[35] Z. Xiao et Y. Xiao, “Security and privacy in cloud computing,” *Communications Surveys & Tutorials, IEEE*, vol. 15, n°. 2, p. 843–859, 2013.

[36] E. Khalil, S. Enniari et M. Zbakh, “Cloud computing architectures based multi-tenant ids,” dans *Security Days (JNS3), 2013 National.* IEEE, 2013, p. 1–5.

[37] *Cloud Security Alliance, Security guidance for critical areas of focus in cloud computing V4.0*, 2017.

[38] *Cloud Security Alliance. Defined Caregories of service 2011*, 2011.

[39] H. AlJahdali, A. Albatli, P. Garraghan, P. Townend, L. Lau et J. Xu, "Multi-tenancy in cloud computing," dans *Service Oriented System Engineering (SOSE), 2014 IEEE 8th International Symposium on*. IEEE, 2014, p. 344–351.

[40] E. Saleh, J. Sianipar, I. Takouna et C. Meinel, "Secplace : A security-aware placement model for multi-tenant saas environments," dans *2014 IEEE 14th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UTC-ATC-ScalCom)*. IEEE, 2014, p. 596–602.

[41] P. Rane, "Securing saas applications : a cloud security perspective for application providers," *Information Security Management Handbook (2010)*, vol. 5, 2010.

[42] I. Juniper Networks, "- securing multi-tenancy and cloud computing," Rapport technique, 2012.

[43] "Robbie higgins. securing a multi-tenant environment." 2015, URL : http://searchcloudsecurity.techtarget.com/tip/Securing-a-multi-tenant-environment [accessed : 2017-05-15].

[44] C. D'souza, "Sql injection in saas cloud layer," Rapport technique, 2013.

[45] K. Hashizume, D. G. Rosado, E. Fernández-Medina et E. B. Fernandez, "An analysis of security issues for cloud computing," *Journal of Internet Services and Applications*, vol. 4, n°. 1, p. 5, 2013.

[46] S. Jafarpour et A. Yousefi, "Security risks in cloud computing : A review. international journal of current engineering and technology (2016)," 2016.

[47] B. T. Rao *et al.*, "A study on data storage security issues in cloud computing," *Procedia Computer Science*, vol. 92, p. 128–135, 2016.

[48] P. K. Tiwari et S. Joshi, "A review of data security and privacy issues over saas," dans *Computational Intelligence and Computing Research (ICCIC), 2014 IEEE International Conference on*. IEEE, 2014, p. 1–6.

[49] C. Priebe, D. Muthukumaran, D. O'Keeffe, D. Eyers, B. Shand, R. Kapitza et P. Pietzuch, "Cloudsafetynet : Detecting data leakage between cloud tenants," dans *Proceedings of the 6th edition of the ACM Workshop on Cloud Computing Security*. ACM, 2014, p. 117–128.

[50] H. Tianfield, "Security issues in cloud computing," dans *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on*. IEEE, 2012, p. 1082–1089.

[51] V. J. Winkler, *Securing the Cloud : Cloud computer Security techniques and tactics*. Elsevier, 2011.

[52] "Owasp top 10 application security risks - 2017," 2018, URL : https://www.owasp.org/index.php/Top_10-2017_Top_10 [accessed : 2018-04-03].

[53] T. Karnwal, T. Sivakumar et G. Aghila, "A comber approach to protect cloud computing against xml ddos and http ddos attack," dans *Electrical, Electronics and Computer Science (SCEECS), 2012 IEEE Students' Conference on.* IEEE, 2012, p. 1–5.

[54] Miscrosoft, "Understanding ws-security," https ://msdn.microsoft.com/en-us/library/ms977327.aspx, 2015, consulté le 2 mars 2015.

[55] OASIS, "Oasis web services security (wss)," https ://www.oasis-open.org/committees/tc_home.php ?wg_abbrev=wss, 2015, consulté le 6 avril 2015.

[56] W3C, "Security," http ://www.w3.org/standards/xml/security, 2015, consulté le 23 août 2015.

[57] S. VivinSandar et S. Shenai, "Economic denial of sustainability (edos) in cloud services using http and xml based ddos attacks," *International Journal of Computer Applications*, vol. 41, n°. 20, p. 11–16, 2012.

[58] S. Suriadi, A. Clark et D. Schmidt, "Validating denial of service vulnerabilities in web services," dans *Network and System Security (NSS), 2010 4th International Conference on.* IEEE, 2010, p. 175–182.

[59] D. H. Parekh et R. Sridaran, "An analysis of security challenges in cloud computing," *IJACSA) International Journal of Advanced Computer Science and Applications*, vol. 4, n°. 1, 2013.

[60] S. Curtis, "Barclays : 97 percent of data breaches still due to sql injection," http ://www.techworld.com/news/security/barclays-97-percent-of-data-breaches-still-due-sql-injection-3331283/, 2015, consulté le 19 janvier 2015.

[61] J. Williams et D. Wichers, "Owasp : Top 10 - 2013. the ten most critical web application security risks," 2013.

[62] S.-t. Sun, T. H. Wei, S. Liu et S. Lau, "Classification of sql injection attacks," Rapport technique, 2006.

[63] W. G. J. Halfond, J. Viegas et A. Orso, "A classification of sql injection attacks and countermeasures," *Georgia Institute of Technology*, p. 1–11, 2006.

[64] "SaaS Solutions on AWS. Isolation architectures. Amazon Web Services." Rapport technique, 2017.

[65] "Securing multi-tenant appliations," 2017, URL : https://msdn.microsoft.com/en-us/library/hh534483.aspx [accessed : 2017-07-10].

[66] "Cloud security that accelerates business," 2017, URL : https://www.skyhighnetworks.com/cloud-security-blog/only-9-4-of-cloud-providers-are-encrypting-data-at-rest/ [accessed : 2017-03-21].

[67] "Design patterns for multitenant saas applications and azure sql database," 2017, URL : https://docs.microsoft.com/en-us/azure/sql-database/sql-database-design-patterns-multi-tenancy-saas-applications [accessed : 2017-01-04].

[68] S. S. Gulati et S. Gupta, "A framework for enhancing security and performance in multi-tenant applications," *International Journal of Information Technology*, vol. 5, n°. 2, p. 233–237, 2012.

[69] E. Saleh, I. Takouna et C. Meinel, "Signedquery : Protecting users data in multi-tenant saas environments," dans *Advances in Computing, Communications and Informatics (ICACCI), 2013 International Conference on.* IEEE, 2013, p. 213–218.

[70] M. Ficco et M. Rak, "Intrusion tolerance of stealth dos attacks to web services," dans *Information Security and Privacy Research.* Springer, 2012, p. 579–584.

[71] I. Prevoty, "Product overview," https ://www.prevoty.com/product/, 2015, consulté le 12 août 2015.

[72] Imperva, "Imperva securesphere web application firewall," http ://www.imperva.com/Products/WebApplicationFirewall, 2015, consulté le 22 août 2015.

[73] V. Varadharajan et U. Tupakula, "On the design and implementation of an integrated security architecture for cloud with improved resilience," *IEEE Transactions on Cloud Computing*, vol. 5, n°. 3, p. 375–389, 2016.

[74] P. Mishra, V. Varadharajan, E. Pilli et U. Tupakula, "Vmguard : A vmi-based security architecture for intrusion detection in cloud environment," *IEEE Transactions on Cloud Computing*, 2018.

[75] V. Varadharajan et U. Tupakula, "Security as a service model for cloud environment," *IEEE Transactions on network and Service management*, vol. 11, n°. 1, p. 60–75, 2014.

[76] T. Alharkan et P. Martin, "Idsaas : Intrusion detection system as a service in public clouds," dans *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012).* IEEE Computer Society, 2012, p. 686–687.

[77] J. He, K. Ota, M. Dong, L. T. Yang, M. Fan, G. Wang et S. S. Yau, "Customized network security for cloud service," *IEEE Transactions on Services Computing*, 2017.

[78] C. N. Modi, D. R. Patel, A. Patel et M. Rajarajan, "Integrating signature apriori based network intrusion detection system (nids) in cloud computing," *Procedia Technology*, vol. 6, p. 905–912, 2012.

[79] C. N. Modi et D. Patel, "A novel hybrid-network intrusion detection system (h-nids) in cloud computing," dans *Computational Intelligence in Cyber Security (CICS), 2013 IEEE Symposium on.* IEEE, 2013, p. 23–30.

[80] S. Gupta et P. Kumar, "An immediate system call sequence based approach for detecting malicious program executions in cloud environment," *Wireless Personal Communications*, vol. 81, n°. 1, p. 405–425, 2015.

[81] A. Khaldi, K. Karoui et H. B. Ghezala, "Framework to detect and repair distributed intrusions based on mobile agent in hybrid cloud," dans *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2014.

[82] M. Moorthy et M. Rajeswari, "Virtual host based intrusion detection system for cloud," *International Journal of Engineering & Technology*, p. 0975–4024, 2013.

[83] C. Ambikavathi et S. Srivatsa, "Improving virtual machine security through intelligent intrusion detection system," *Indian Journal of Computer Science and Engineering (IJCSE)*, vol. 6, p. 39, 2015.

[84] C.-C. Lo, C.-C. Huang et J. Ku, "A cooperative intrusion detection system framework for cloud computing networks," dans *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on.* IEEE, 2010, p. 280–284.

[85] U. Nagar, P. Nanda, X. He et Z. T. Tan, "A framework for data security in cloud using collaborative intrusion detection scheme," dans *Proceedings of the 10th International Conference on Security of Information and Networks.* ACM, 2017, p. 188–193.

[86] S. T. Zargar, H. Takabi et J. B. Joshi, "Dcdidp : A distributed, collaborative, and data-driven intrusion detection and prevention framework for cloud computing environments," dans *Collaborative computing : Networking, applications and worksharing (collaboratecom), 2011 7th international conference on.* IEEE, 2011, p. 332–341.

[87] D. Singh, D. Patel, B. Borisaniya et C. Modi, "Collaborative ids framework for cloud," *International Journal of Network Security*, vol. 18, n°. 4, p. 699–709, 2016.

[88] S. Ghribi, A. M. Makhlouf et F. Zarai, "C-dids : A cooperative and distributed intrusion detection system in cloud environment," dans *2018 14th International Wireless Communications & Mobile Computing Conference (IWCMC).* IEEE, 2018, p. 267–272.

[89] L. Wu, S. K. Garg, S. Versteeg et R. Buyya, “Sla-based resource provisioning for hosted software-as-a-service applications in cloud computing environments,” *Services Computing, IEEE Transactions on*, vol. 7, n°. 3, p. 465–485, 2014.

[90] ZDNet, “Chiffres clés : le marché du cloud computing,” http ://www.zdnet.fr/actualites/chiffres-cles-le-marche-du-cloud-computing-39790256.htm ?p=3, 2015, consulté le 3 juillet 2015.

[91] C. Cerrudo, “Manipulating Microsoft SQL Server Using SQL Injection,” Application Security, INC., Rapport technique, 2010.

[92] salesforce, “Secure coding sql injection,” https ://developer.salesforce.com/page/Secure_Coding_SQL_Injection, 2015, consulté le 9 mais 2015.

[93] K. Scarfone et P. Mell, “Guide to intrusion detection and prevention systems (idps),recommendations of the national institute of standards and technology gaithersburg, md 20899-8930, usa,” *NIST Computer Science Special Publication 800-94*, n°. February, p. pages 1–127, 2007.

[94] T. Probst, E. Alata, M. Kaâniche et V. Nicomette, “Automated evaluation of network intrusion detection systems in iaas clouds,” dans *Dependable Computing Conference (EDCC), 2015 Eleventh European*. IEEE, 2015, p. 49–60.

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

[96] G. T. Buehrer, B. W. Weide et P. 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*, n°. September, p. pages 106–113, 2005.

[97] D. Scott et R. Sharp, “Abstracting application-level web security,” *In Proceedings of the 11th International Conference on the World Wide Web*, n°. January, p. pages 396–407, 2002.

[98] V. Varadharajan et U. Tupakula, “Security as a service model for cloud environment,” *Network and Service Management, IEEE Transactions on*, vol. 11, n°. 1, p. 60–75, 2014.

[99] OWASP, “Hacmebank vulnerable web application,” https ://www.owasp.org/index.php/HacmeBank, 2015.

[100] V. Levenshtein et F. Damerau, “Distance de damerau-levenshtein,” http ://fr.wikipedia.org/wiki/Distance_de_Levenshtein, 2014.

[101] H. ibrahim Kalkan, “Dotnetmq : A complete message queue system for .net,” http ://www.codeproject.com/Articles/193611/DotNetMQ-A-Complete-Message-Queue-System-for-NET, 2016.

[102] F. R. Gianluca Varenni, Loris Degioanni et J. Bruno, “The industry-standard windows packet capture library,” http ://www.winpcap.org/, 2015.

[103] B. D. A. G. et M. Stampar, “Automatic sql injection and database takeover tool,” https ://github.com/sqlmapproject/sqlmap, 2018.

[104] L. Impact, “Performence testing fordevops,” https ://loadimpact.com/, 2016.

[105] S. Paliwal, “Cloud application services (saas)–multi-tenant data architecture,” *Infosys technologies limited, URL : http ://www. cmg. org/wp-content/uploads/2012/11/m_94_4. pdf,[accessed on : 10 Sep 2014]*, 2012.

[106] “The force.com multitenant architecture,” 2017, URL : https://developer.salesforce.com/page/Multi_Tenant_Architecture [accessed : 2017-01-02].

[107] M. Yassin, H. Ould-Slimane, C. Talhi et H. Boucheneb, “Sqliidaas : A sql injection intrusion detection framework as a service for saas providers,” dans *Cyber Security and Cloud Computing (CSCloud), 2017 IEEE 4th International Conference on.* IEEE, 2017, p. 163–170.

[108] “Autofac contributors. multitenant applications.” 2014, URL : http://docs.autofac.org/en/latest/advanced/multitenant.html#what-is-multitenancy [accessed : 2017-01-07].

[109] “What is an n-tier architecture ?” 2018, URL : https://docs.microsoft.com/en-us/learn/modules/n-tier-architecture/2-what-is-n-tier-architecture [accessed : 2018-05-14].

[110] R. Krebs, C. Momm et S. Kounev, “Architectural concerns in multi-tenant saas applications.” *Closer*, vol. 12, p. 426–431, 2012.

[111] *Cloud Security Alliance, SecaaS Implementation Guidance,Category 3, Web Security*, 2012.

[112] K. Wang et Y. Hou, “Detection method of sql injection attack in cloud computing environment,” dans *Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC), 2016 IEEE.* IEEE, 2016, p. 487–493.

[113] A. Liu, Y. Yuan, D. Wijesekera et A. Stavrou, “Sqlprob : a proxy-based architecture towards preventing sql injection attacks,” dans *Proceedings of the 2009 ACM symposium on Applied Computing.* ACM, 2009, p. 2054–2061.

[114] G. Buehrer, B. W. Weide et P. A. Sivilotti, “Using parse tree validation to prevent sql injection attacks,” dans *Proceedings of the 5th international workshop on Software engineering and middleware.* ACM, 2005, p. 106–113.

[115] "Logical operators (transact-sql)," 2017, URL : https://docs.microsoft.com/en-us/sql/t-sql/language-elements/logical-operators-transact-sql [accessed : 2017-07-18].

[116] "Comparison operators (transact-sql)," 2017, URL : https://docs.microsoft.com/en-us/sql/t-sql/language-elements/comparison-operators-transact-sql [accessed : 2017-04-14].

[117] "Binary trees," 2017, URL : https://www.cs.cmu.edu/~adamchik/15-121/lectures/Trees/trees.html [accessed : 2017-06-14].

[118] "Telerik fiddler, the free web debugging proxy for any browser, system or platform," 2017, URL : http://www.telerik.com/fiddler [accessed : 2017-07-13].

[119] "Hacmebank vulnerable web application," 2017, URL : https://www.owasp.org/index.php/HacmeBank [accessed : 2017-05-03].

[120] "F. r. gianluca varenni, loris degioanni and j. bruno. the industry standard windows packet capture library," 2017, URL : http://www.winpcap.org [accessed : 2017-05-10].

[121] "Event cloud," 2019, URL : https://www.codeproject.com/Articles/1043326/%2FArticles%2F1043326%2FA-Multi-Tenant-SaaS-Application-With-ASP-NET-MVC-A [accessed : 2019-01-15].

[122] "Performence testing for devops," 2017, URL : https://loadimpact.com/ [accessed : 2017-06-04].

[123] S. Iqbal, M. L. M. Kiah, B. Dhaghighi, M. Hussain, S. Khan, M. K. Khan et K.-K. R. Choo, "On cloud security attacks : A taxonomy and intrusion detection and prevention as a service," *Journal of Network and Computer Applications*, vol. 74, p. 98–120, 2016.

[124] H. Tianfield, "Security issues in cloud computing," dans *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on.* IEEE, 2012, p. 1082–1089.

[125] P. Mishra, E. S. Pilli, V. Varadharajan et U. Tupakula, "Intrusion detection techniques in cloud environment : A survey," *Journal of Network and Computer Applications*, vol. 77, p. 18–47, 2017.

[126] G. Vigna, F. Valeur, D. Balzarotti, W. Robertson, C. Kruegel et E. Kirda, "Reducing errors in the anomaly-based detection of web-based attacks through the combined analysis of web requests and sql queries," *Journal of Computer Security*, vol. 17, n°. 3, p. 305–329, 2009.

[127] W. G. J. Halfond et A. Orso, "Preventing SQL injection attacks using AMNESIA," *Proceeding of the 28th international conference on Software engineering - ICSE '06*, p. 1–4, 2006.

[128] Z. Su. et G. Wassermann, “SQLCheck :The Essence of Command Injection Attacks in Web Applications,” *Proceedings of the 5th international workshop on Software engineering and middleware - SEM 205*, n°. January, 2006.

[129] S. Bandhakavi, P. Bisht, P. Madhusudan et V. Venkatakrishnan, “Candid : Preventing sql injection attacks using Dynamic candidate evaluations,” *In Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, n°. October, p. pages 12–24, 2007.

[130] K. Wang et Y. Hou, “Detection method of sql injection attack in cloud computing environment,” dans *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*. IEEE, 2016, p. 487–493.

[131] K. Gupta, R. R. Singh et M. Dixit, “Cross site scripting (xss) attack detection using intrustion detection system,” dans *Intelligent Computing and Control Systems (ICICCS), 2017 International Conference on*. IEEE, 2017, p. 199–203.

[132] M. Ficco et M. Rak, “Stealthy denial of service strategy in cloud computing,” *IEEE transactions on cloud computing*, vol. 3, n°. 1, p. 80–94, 2014.

[133] M. Jelidi, A. Ghourabi et K. Gasmi, “A hybrid intrusion detection system for cloud computing environments,” dans *2019 International Conference on Computer and Information Sciences (ICCIS)*. IEEE, 2019, p. 1–6.

[134] A. Shawahna, M. Abu-Amara, A. Mahmoud et Y. E. Osais, “Edos-ads : An enhanced mitigation technique against economic denial of sustainability (edos) attacks,” *IEEE Transactions on Cloud Computing*, 2018.

[135] R. M. Sarhadi et V. Ghafori, “New approach to mitigate xml-dos and http-dos attacks for cloud computing,” *International Journal of Computer Applications*, vol. 72, n°. 16, 2013.

[136] Z. Chiba, N. Abghour, K. Moussaid, A. El Omri et M. Rida, “A survey of intrusion detection systems for cloud computing environment,” dans *Engineering & MIS (ICEMIS), International Conference on*. IEEE, 2016, p. 1–13.

[137] V. Levenshtein et F. Damerau, “Distance de damerau-levenshtein,” http ://fr.wikipedia.org/wiki/Distance_de_Levenshtein, 2018.

[138] “Amazon ec2 pricing,” 2018, URL : https://aws.amazon.com/ec2/pricing/on-demand/ [accessed : 2018-06-10].

[139] T. Kaur, D. Kaur et A. Aggarwal, “Cost model for software as a service,” dans *Confluence The Next Generation Information Technology Summit (Confluence), 2014 5th International Conference-*. IEEE, 2014, p. 736–741.

[140] A. Schwanengel et U. Hohenstein, “Challenges with tenant-specific cost determination in multi-tenant applications,” 2013.

[141] “An automated xss payload generator written in python.” 2018, URL : https://github.com/mandatoryprogrammer/xssless [accessed : 2018-06-11].

[142] “Security briefs - xml denial of service attacks and defenses,” 2017, URL : https://msdn.microsoft.com/en-us/magazine/ee335713.aspx [accessed : 2018-6-14].

[143] M. Elsayed et M. Zulkernine, “Offering security diagnosis as a service for cloud saas applications,” *Journal of information security and applications*, vol. 44, p. 32–48, 2019.