UQAC
Université du Québec
à Chicoutimi

LES TESTS UNITAIRES COMME MÉTHODE DE PRÉVENTION CONTRE LES INJECTIONS SQL

PAR MARTIN RENAUD

MÉMOIRE PRÉSENTÉ À L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI EN VUE DE L'OBTENTION DU GRADE DE MAÎTRE ÈS SCIENCES (M. SC.) EN INFOMATIQUE

QUÉBEC, CANADA

© MARTIN RENAUD, 2024

RÉSUMÉ

Ce mémoire de maîtrise explore l'utilisation des tests unitaires afin de prévenir les vulnérabilités relatives aux injections SQL. Les injections SQL demeurent encore et toujours l'une des failles de sécurité les plus courantes et potentiellement dangereuses dans les applications. Cette étude examine comment les tests unitaires peuvent servir de rempart contre ces vulnérabilités en permettant d'identifier celles-ci et en les corrigeant de façon adéquate.

L'approche adoptée dans ce mémoire comprend une analyse approfondie des techniques de test actuelles et de leur intégration dans les processus de développement sécuritaire de logiciel. En mettant l'accent sur la méthodologie de développement et des scénarios réalistes, nous souhaitons proposer une pratique de test unitaire spécifique pour détecter les faiblesses et prévenir les injections SQL dès les premières phases du développement.

Les résultats de cette étude démontrent que l'adoption de tests unitaires appropriés peut renforcer la sécurité du code source en réduisant significativement les risques d'exploitation des vulnérabilités aux injections SQL. En démontrant l'efficacité de ces derniers, ce mémoire offre un aperçu des bénéfices de cette approche aux équipes de développement pour renforcer la résilience des applications contre les attaques par injections SQL.

TABLE DES MATIÈRES

RÉSUMÉ ii

LISTE DES TABLEAUX v

LISTE DES FIGURES vi

LISTE DES ABRÉVIATIONS ix

DÉDICACE x

REMERCIEMENTS xi

AVANT-PROPOS xii

CHAPITRE I – INTRODUCTION 1

1.1 CONTEXTE 1

1.2 MOTIVATION 6

1.3 OBJECTIFS 7

1.4 CONTEXTE DE LA RECHERCHE 8

1.5 CONTRIBUTION 10

1.6 ORGANISATION DU DOCUMENT 10

CHAPITRE II – REVUE DE LA LITTÉRATURE 12

2.1 RISQUE ET APERÇU DES ATTAQUES PAR INJECTION SQL 14

2.2 TECHNIQUES ET MEILLEURES PRATIQUES POUR CONTRER LES INJECTIONS SQL 20

2.3 LES TESTS UNITAIRES ET LEURS UTILITÉS DANS UN CONTEXTE DE PRÉVENTION CONTRE LES INJECTIONS SQL 25

CHAPITRE III – MÉTHODOLOGIE SCIENTIFIQUE 30

3.1 JUSTIFICATION DE LA DÉMARCHE ET SYNTHÈSE DE LA MÉTHODE 30

3.2 SYNTHÈSE DE LA MÉTHODE ET STRATÉGIE DE TEST 34

3.3 PLAN DE TEST 35

3.3.1 MÉTHODE DE CRÉATION DE SCÉNARIOS DE TESTS 36

3.3.2 LES SCÉNARIOS DE TESTS UNITAIRES 38

3.3.3 TESTS UNITAIRES - TRANSVERSAUX AU SYSTÈME 50
CHAPITRE IV – RÉSULTATS 53
4.1 INTRODUCTION 53
4.2 JEU DE DONNÉES 54
4.3 APPROFONDISSEMENT DE LA MÉTHODE 62
4.4 RÉSULTATS 63
4.5 CONCLUSION 92
CHAPITRE V – DISCUSSION 95
5.1 IMPLICATION DE L'APPROCHE POUR LES DÉVELOPPEURS 95
5.1.1 QUELLES CONNAISSANCES LES DÉVELOPPEURS ONT BESOIN POUR CRÉER LES TESTS ? 96
5.1.2 QUELLES CONNAISSANCES LES DÉVELOPPEURS ONT BESOIN POUR EXÉCUTER LES TESTS ? 97
5.2 LIMITATION DE LA RECHERCHE 98
5.2.1 MENACES À LA VALIDITÉ : INTERNE 99
5.2.2 MENACES À LA VALIDITÉ : EXTERNE 101
5.2.3 PERSPECTIVES D'AMÉLIORATION 102
5.3 CRITIQUE DE L'APPROCHE 103
5.4 CONCLUSION 105
CHAPITRE VI – CONCLUSION 107
BIBLIOGRAPHIE 111

LISTE DES TABLEAUX

TABLEAU 4.1 : EXEMPLE D'INTRANTS COMPOSÉS DE PARAMÈTRES. 58

TABLEAU 4.2 : MATRICE DES ATTENTES (A) ET DES RÉSULTATS (R). 59

LISTE DES FIGURES

FIGURE 1.1 – DÉPENSES DE 2020 ET 2021, POUR CHACUN DES GRANDS SECTEURS DE LA SÉCURITÉ, EN MILLIONS DE DOLLARS AMÉRICAINS. . . . 3

FIGURE 1.2 – CYCLE DEVOPS 4

FIGURE 1.3 – BRÈCHES SUBIES PAR LES ORGANISATIONS EN 2022. SOURCE : IBM. . . . 5

FIGURE 1.4 – CYCLE DEVOPS 9

FIGURE 2.1 – EXEMPLE D'UN PROCESSUS DE CRÉATION D'UNE REQUÊTE LÉGITIME VERSUS LA CRÉATION D'UNE REQUÊTE MALVEILLANTE CONTENANT UNE INJECTION SQL. . . . 17

FIGURE 2.2 – DIAGRAMME INDIQUANT LE NOMBRE DE VULNÉRABILITÉS LIÉES AUX INJECTIONS SQL DE 2013 À 2023. SOURCE : CVE DETAILS 20

FIGURE 2.3 – CROISSANCE D'UN PROJET AVEC ET SANS TESTS UNITAIRES. REPRODUIT AVEC LA PERMISSION DE MANNING. SOURCE : KHORIKOV. . . . 28

FIGURE 3.1 – SCHÉMA DES CONTRE-MESURES. . . . 33

FIGURE 4.1 – SCHÉMA DE LA BASE DE DONNÉES *CHINOOK* SOUS MYSQL ET MSSQL, INCLUANT LES TABLES *USERACCOUNT* ET *LOGS*. . . 56

FIGURE 4.2 – RÈGLES D'ACCÈS ATTRIBUÉES AUX COMPTES DE SERVICES. . 57

FIGURE 4.3 – TEST UNITAIRE À PARTIR DE LA COUCHE MÉTIER. . . . 61

FIGURE 4.4 – ARCHITECTURE DE LA SIMULATION DU LOGICIEL.. . . . 61

FIGURE 4.5 – SOMMAIRE DES RÉSULTATS DES TESTS SÉCURISÉS ET NON SÉCURISÉS POUR LA SUITE DE SCÉNARIOS 1 À 4 64

FIGURE 4.6 – RÉSULTATS DU TEST 1.1. . . . 65

FIGURE 4.7 – RÉSULTATS DU TEST 1.2. . . . 66

FIGURE 4.8 – RÉSULTATS DU TEST 1.3 67
FIGURE 4.9 – RÉSULTATS DU TEST 2.1 68
FIGURE 4.10 – EXCEPTION - INTRANT 6 69
FIGURE 4.11 – EXCEPTION - INTRANT 7 69
FIGURE 4.12 – RÉSULTATS DU TEST 3.1 70
FIGURE 4.13 – RÉSULTATS DU TEST 3.2 71
FIGURE 4.14 – EXCEPTION - INTRANT 6 72
FIGURE 4.15 – RÉSULTATS DU TEST 3.3 72
FIGURE 4.16 – RÉSULTATS DU TEST 4.1 74
FIGURE 4.17 – EXCEPTION - INTRANT 5 75
FIGURE 4.18 – EXCEPTION - INTRANT 6 75
FIGURE 4.19 – RÉSULTATS DU TEST 4.1 NON SÉCURISÉ SOUS MSSQL 75
FIGURE 4.20 – RÉSULTATS DU TEST 4.2 76
FIGURE 4.21 – EXCEPTION - INTRANT 5 77
FIGURE 4.22 – RÉSULTATS DU TEST 4.2 NON SÉCURISÉ SOUS MSSQL 77
FIGURE 4.23 – RÉSULTATS DU TEST 4.3 78
FIGURE 4.24 – EXCEPTION - INTRANT 5 79
FIGURE 4.25 – RÉSULTATS DU TEST 4.3 NON SÉCURISÉ SOUS MSSQL 79
FIGURE 4.26 – RÉSULTATS DU TEST 4.4 80
FIGURE 4.27 – EXCEPTION - INTRANT 4 81
FIGURE 4.28 – RÉSULTATS DU TEST 4.4 NON SÉCURISÉ SOUS MSSQL 81
FIGURE 4.29 – EXCEPTION - INTRANT 4 81
FIGURE 4.30 – APERÇU DES RÉSULTATS DES TESTS 5 À 7 82

FIGURE 4.31 – RÉSULTATS DES TESTS SOUS MYSQL ET MSSQL 83
FIGURE 4.32 – EXCEPTION SOUS MYSQL 83
FIGURE 4.33 – EXCEPTION SOUS MSSQL. 84
FIGURE 4.34 – RÉSULTATS DES TESTS SOUS MYSQL ET MSSQL 85
FIGURE 4.35 – EXCEPTIONS DES TESTS SOUS MYSQL ET MSSQL, POUR LES SCÉNARIOS 6 86
FIGURE 4.36 – RÉSULTATS DES TESTS DES SCÉNARIOS 7 87
FIGURE 4.37 – EXCEPTIONS DES TESTS SOUS MYSQL ET MSSQL, POUR LES SCÉNARIOS 7 88
FIGURE 4.38 – ATTAQUE PAR SQLMAP SUR LES APIS. 89
FIGURE 4.39 – RÉSULTAT NON CONCLUANT, PROVENANT DE L'ATTAQUE SQLMAP 90
FIGURE 4.40 – RÉSULTAT CONCLUANT, PROVENANT DE L'ATTAQUE SQLMAP 91
FIGURE 4.41 – EXCEPTIONS LEVÉES PAR NOS CONTRÔLES, LORS DE LA SIMULATION DE L'ATTAQUE. 92

LISTE DES ABRÉVIATIONS

API Application Programming Interface
CaaS Cybercrime as a Service
DAST Dynamic Application Security Testing
DoS Denied of Service
HTML HyperText Markup Language
JDBC Java Database Connectivity
MSSQL Microsoft SQL Server
MySQL Oracle MySQL
OWASP Open Web Application Security Project
RCE Remote Code Execution
SAST Static Application Security Testing
SGBD Système de gestion de base de données
SQL Structured Query Language
SQLi SQL Injection
XML Extensible Markup Language
XSS Cross Site Scripting

DÉDICACE

Je dédie ce mémoire à ma conjointe, Louise et mes enfants Benjamin, Antoine et Coraline. Ils ont su m'encourager durant mes études. Sans eux et leur soutien, aucune étude universitaire n'aurait été possible.

Vous êtes mes amours. Merci pour tout.

REMERCIEMENTS

Réaliser un projet de recherche à temps partiel ne semble pas chose simple. On doit continuellement naviguer entre notre vie professionnelle et notre vie académique. D'un côté, oeuvrer pour solutionner des problèmes opérationnels et technologiques, et d'un autre côté focaliser sur un sujet de recherche. J'ai donc dû trouver cet équilibre.

Je souhaite remercier mes directeurs Raphaël Khoury et Sylvain Hallé. Ils ont su me guider tout au long de mon processus de recherche. J'aimerais remercier également Christophe Père qui a su me soutenir dans les bons et moins bons moments. Il a su me garder, d'une certaine façon, motivé.

Également, j'aimerais remercier mon collègue et ami, Patrick Leclerc de Beneva, qui a su me poser les bonnes questions et mettre à défi l'aspect appliqué de mon projet de recherche.

Finalement, j'aimerais également remercier mon ancien collègue et ami, Martin Trudel, pour son soutien dans ma démarche de futur scientifique.

AVANT-PROPOS

Lorsque je remettrai ce mémoire, j'aurai cumulé plus de vingt-et-une années en tant que professionnel dans le domaine des technologies de l'information. J'aurai passé un peu plus de la moitié de ma carrière dans le secteur du développement de logiciel. J'aurai touché à plus d'une dizaine de langage de programmation et tout autant dans les divers technologies qui soutiennent et opérationnalisent les logiciels. Mon intérêt dans la sécurité s'est manifesté lors d'un cours d'introduction, reçu vers la fin de mon baccalauréat à l'Université du Québec à Chicoutimi (UQAC). Ce dernier était donné par Sylvain Hallé, professeur à l'UQAC et co-directeur de ma recherche. Par ailleurs, il s'agissait d'un cours optionnel. Le premier et le seul cours sur la sécurité informatique que j'ai eu durant cette formation académique. Dès lors, j'ai commencé à m'intéresser au monde de la sécurité informatique, plus précisément à la cybersécurité. Lors de ma maîtrise, j'ai pu assister à d'autres cours dédiés à la sécurité, dont un cours en sécurité informatique donné par Raphaël Khoury, également professeur à l'UQAC et co-directeur de ma recherche.

Mes premières expériences professionnelles en sécurité furent comme testeur d'intrusion. Mes tâches se résumaient à personnifier un pirate informatique et tenter de m'introduire dans les systèmes informatiques en exploitant leurs failles. Par la suite, je suis devenu analyste en sécurité, puis architecte de sécurité et finalement conseiller en architecture.

Aujourd'hui, j'occupe le poste de conseiller en architecture de sécurité informatique chez Beneva et je me spécialise en sécurité du développement logiciel et en protection du périmètre applicatif.

CHAPITRE I
INTRODUCTION

1.1 CONTEXTE

En 2021, 18% des entreprises canadiennes ont été victimes de cyberattaques (Canada, 2022). En 2022, le Canada s'est positionné au 3e rang des coûts (5,64 millions de dollars américains) pour une brèche de données, derrière les États-Unis et le Moyen-Orient (IBM, 2023). Si au début des années 2000, seule une poignée d'initiés pouvaient mettre à mal des organisations, il est désormais concevable de prétendre que depuis les dix dernières années, cette pratique malveillante s'est démocratisée à un niveau où la société fait face à un fléau qui prend de l'ampleur chaque année [1]. Les techniques et la connaissance en piratage informatique se démocratisent à un point tel que la ligne devient de plus en plus mince entre les professionnels du domaine et les pirates. Seules les intentions semblent déterminer de quel côté de la ligne les initiés se positionnent. Il n'est plus rare de voir apparaître de nouveaux services malveillants en ligne, tels que des plateformes offrant des solutions clés en main de type *Cybercrime as a Service* (CaaS) (Manky, 2013), *Ransomware as a Service* (RaaS) (Alwashali *et al.*, 2021) ou *Drain as a Service* (DaaS) (Gatlan, 2024). Ce qui oblige par conséquent, les gouvernements, les institutions et les entreprises à se protéger contre les menaces du cyberespace.

En 2021, les entreprises canadiennes ont déclaré avoir investi plus de dix milliards de dollars en cybersécurité (Canada, 2022). Si la sécurité dans les secteurs des télécommunications et des infrastructures technologiques a été en mesure de répondre aux besoins de protection, il

1. World's Biggest Data Breaches and Hacks. Consulté en janvier 2023 à l'adresse https://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/

en est tout autrement dans le secteur du développement logiciel. La sécurité applicative reste encore un des secteurs les moins privilégiés du domaine de la sécurité. La société *Gartner* a publié un rapport sur les prévisions des dépenses mondiales en matière de sécurité et en gestion des risques (Gartner, 2021). Selon ce rapport, les prévisions des dépenses au niveau du segment relié à la sécurité applicative étaient au sixième rang en 2021, comparativement au huitième rang en 2020. Bien qu'il y ait eu une amélioration durant les dernières années, le manque de maturité des logiciels au niveau de la sécurité reste perceptible (Figure 1.1). Ce n'est certes pas par manque de référentiels et d'encadrement. Il existe plusieurs normes et standards, des cadres de gestion et de cadres de travail relatifs aux pratiques de l'industrie. Mais encore ici, faut-il les appliquer. Les solutions et produits commerciaux en sécurité applicative font de plus en plus surface sur le marché de la sécurité. Des solutions d'analyse et de détection combinant intelligence artificielle et service infonuagique font les grands titres des sites promotionnels et des événements de cybersécurité. Avec l'entrée en vigueur de la loi 25 en septembre 2021, au Québec, les organisations qui oeuvrent dans la province n'ont d'autre choix que de se conformer à cette nouvelle loi et de s'assurer de respecter les règles pour protéger non seulement les données, mais également les systèmes qui les traitent et qui les entreposent.

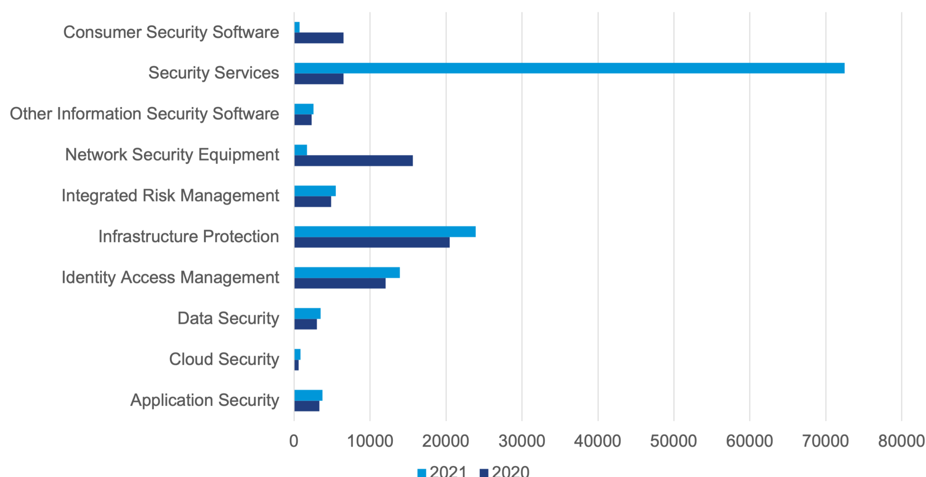

FIGURE 1.1 : Dépenses de 2020 et 2021, pour chacun des grands secteurs de la sécurité, en millions de dollars américains.

En effectuant quelques recherches sur internet, nous observons que les produits les plus courants sont ceux qui se concentrent sur la détection des vulnérabilités au niveau du code source ou du comportement du logiciel. Des solutions qui s'articulent principalement sur la phase de construction (*Build*) ou durant la phase de test. Rien ne semble exister au niveau de la phase de programmation (*Coding*) dans la boucle *DevOps* [2] (Figure 1.2).

Les tests sont une partie importante du processus de création et de maintenance en génie logiciel, puisqu'il s'agit du stade d'évaluation dont l'objectif est de repérer les anomalies, mais également de vérifier que le logiciel répond aux exigences prévues. Les tests au niveau des logiciels sont essentiels, car ils garantissent que ce dernier fonctionne selon les attentes, qu'il répond aux besoins des utilisateurs et finalement, qu'il soit fiable et efficace. Il existe différents types de tests de logiciel et chacun d'eux correspond à une fonction spécifique. On

2. La boucle DevOps est un cycle continu qui vise à améliorer l'efficacité et la qualité du développement logiciel en intégrant les processus de développement (Dev) et d'exploitation (Ops). Elle repose sur les principes d'automatisation, de collaboration et de rétroaction rapide.

3. La figure est basée sur une création provenant de https://freepik.com

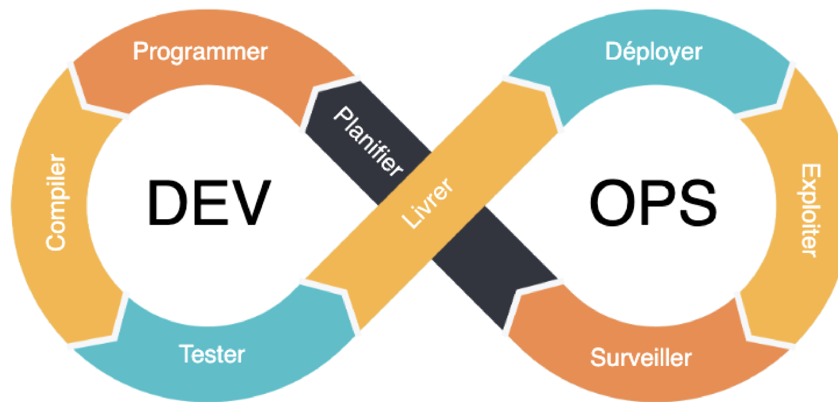

FIGURE 1.2 : Cycle DevOps [3]

retrouve notamment les tests unitaires, les tests d'intégration, les tests de système, les tests d'acceptation, les tests de régression, les tests de performance et les tests de sécurité. Chacune de ces catégories de tests a un objectif précis et peut être effectuée à différentes étapes du cycle de vie du développement du logiciel précis (Leloudas, 2023).

Dans le cadre de ce projet de recherche, nous avons concentré nos efforts sur les tests unitaires. Même s'ils ne constituent qu'une étape dans un processus plus large, nous sommes convaincus de leur impact significatif sur la qualité globale de la solution développée, aussi bien en termes de code source que de fonctionnalités. Nous constatons actuellement l'existence d'un mouvement nommé *Shift Left Security* dans les communautés de pratiques et dans l'industrie de la sécurité informatique (Pitchford, 2021). Un concept qui souhaite voir la sécurité être intégrée très tôt dans le processus de création d'une solution technologique. Au niveau du domaine du génie logiciel, il s'agit d'intégrer les concepts de sécurité applicative au tout début du processus de création et de maintenance. En dépit de ce qui a été réalisé jusqu'à maintenant, l'utilisation de tests unitaires dans un contexte de sécurité logiciel semble peu commune, néanmoins Kolawa (2005) et Gonzalez (2021) décrivent théoriquement les avantages que nous pourrions en tirer.

Selon un rapport produit en 2022 (IBM, 2023), les comportements négligents d'employés ou de sous-traitants sont responsables de 21% des brèches de sécurité subies par les organisations. Les pannes informatiques causées par des interruptions de service ou des défaillances de systèmes, entraînant une perte de données, sont responsables quant à elles de 24% des brèches. Ces problématiques comprennent les erreurs dans le code source ou encore les défaillances dans les processus de communication automatisées, comme indiqué dans la figure 1.3

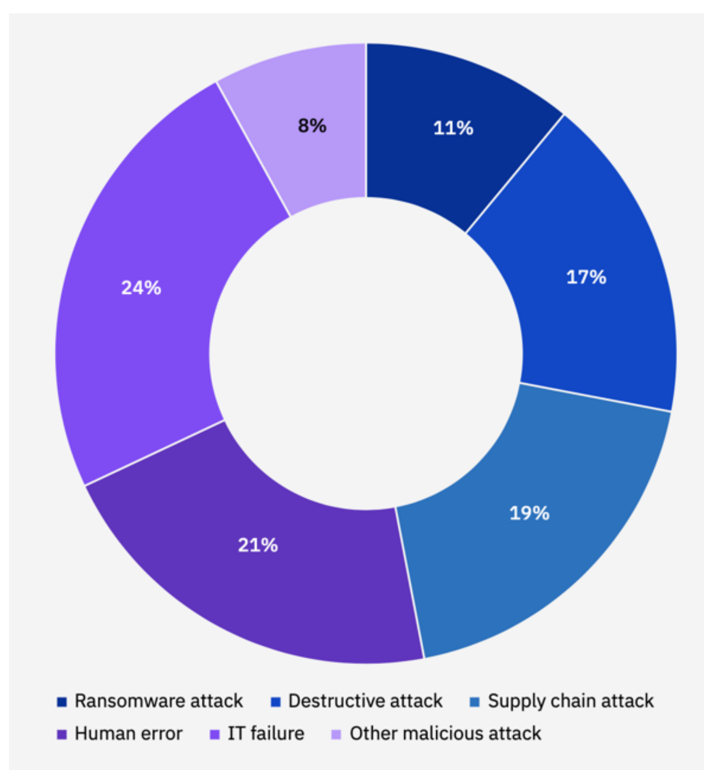

FIGURE 1.3 : Brèches subies par les organisations en 2022.
Source : IBM

1.2 MOTIVATION

En juillet 2005, un texte d'opinion est paru dans le magasine *Military and Aerospace Electronics* intitulé *Reducing software security vulnerabilies through unit testing* (Kolawa, 2005). Cet article a été écrit par le Dr Adam Kolawa, PDG et Co-fondateur de l'entreprise *Parasoft*, une société spécialisée en outils de développement logiciel. Dans son article, Kolawa explique les avantages d'effectuer des tests unitaires dans un contexte de sécurité. Il soutient que grâce à cette méthode, les développeurs peuvent repérer et rectifier les vulnérabilités dès le début du processus de développement. En améliorant la qualité du code source et ses fonctionnalités, et en réduisant les vulnérabilités de l'application, les développeurs contribuent à diminuer les risques de compromission. Selon lui, les études démontrent que la détection précoce d'une faille rend la correction plus simple, plus rapide et moins coûteuse.

En 2021, Danielle Nicole Gonzalez publie une thèse de doctorat intitulée *The State of Practice for Security Unit Testing : Towards Data Driven Strategies to Shift Security into Developer's Automated Testing Workflflows* (Gonzalez, 2021). Cet ouvrage est venu renforcer notre perception qu'une approche par des tests unitaires dans un contexte de sécurité pourrait offrir un excellent moyen de protection contre les vulnérabilités au niveau des logiciels. Dans sa thèse, Gonzalez détaille l'état actuel de cette pratique. Selon elle, les chercheurs en génie logiciel ne semblent pas avoir intégré cette technique dans leurs études sur le développement sécuritaire de logiciel. Elle ajoute que la déconnexion entre la promotion des tests unitaires dans un contexte de sécurité et le manque de preuves empiriques sur la nature et l'application de ces tests est ce qui motive sa thèse. Il semble donc y avoir une évidence que les tests unitaires, plus spécifiquement de sécurité, pourraient offrir un excellent moyen de protection contre les vulnérabilités des logiciels. Un moyen de contrôle, orienté sur la prévention, destiné aux équipes de développement, pouvant les utiliser à des fins de vérification et ainsi garantir la mise en oeuvre des composants et la sécurité des données (Gonzalez *et al.*, 2021). En somme,

un outil qui confirme que des composants logiciels répondent aux exigences de sécurité et qui aide à prévenir l'inclusion de vulnérabilités exploitables par des adversaires malveillants.

1.3 OBJECTIFS

L'objectif de ce projet de recherche vise à poser les bases relatives aux tests unitaires dans un contexte de sécurité. En sécurité applicative, le spectre de composants pouvant influencer la sécurité d'un logiciel est vaste. Les défaillances peuvent toucher autant les composantes logicielles et les paramètres de configuration, que le code source lui-même. Dans cette étude, nous souhaitons concentrer nos efforts au niveau du code source et des vulnérabilités pouvant être introduites lors du développement, plus précisément au niveau des vulnérabilités qui se rapportent aux injections SQL. Ceci étant dit, nous savons qu'il existe des ouvrages sur les tests automatisés relatifs aux injections de code JavaScript (XSS), tel que décrit par Mohammadi *et al.* (2018).

Ce projet de recherche répond à deux questions :

1. Est-ce qu'une approche par tests unitaires dans un contexte de sécurité des systèmes d'information peut éviter l'introduction de vulnérabilités sensibles aux attaques par injection SQL ?
2. Comment créer des scénarios de tests unitaires qui permettent de couvrir l'ensemble de toutes les déclinaisons connues d'attaques par injection SQL pouvant affecter un système d'information ?

En résumé, nos motivations visent à répondre à un besoin persistant en matière de sécurité, d'introduire une nouvelle approche pour résoudre cette problématique, et sensibiliser les principaux acteurs tout en adoptant une perspective interdisciplinaire qui allie la sécurité informatique et les principes fondamentaux du développement logiciel. Ainsi, l'objectif de ce projet de recherche est de démontrer qu'il est possible, en dépit de l'absence de documentation

et de méthodologie concernant une approche de sécurité fondée sur les tests unitaires pour prévenir les attaques par injection SQL, de concevoir des scénarios de tests adéquats qui répondent aux exigences de protection, tout en fournissant une meilleure visibilité sur les vulnérabilités susceptibles d'être introduites au cours du développement. L'objectif est de prévenir l'intégration involontaire de code vulnérable aux injections SQL durant la phase de développement.

1.4 CONTEXTE DE LA RECHERCHE

Dans l'industrie, il existe des solutions technologiques basées sur les analyses statiques (*Static Application Security Testing (SAST)*) (Dencheva, 2022), lesquelles permettent de faire des analyses au niveau du code source. Ces solutions commerciales détectent les mauvaises portions de code source qui ne respectent pas les gabarits standards et proposent des corrections, afin de respecter les bonnes pratiques de sécurité. D'autres solutions, quant à elles, orientent leurs efforts sur des mécanismes d'analyse dynamique (*Dynamic Application Security Testing (DAST)*), lesquelles permettent d'analyser le comportement d'une application au moment de son exécution (Dencheva, 2022). Elles analysent donc les entrées-sorties des processus et leurs empreintes comportementales, et par la suite proposent des méthodes de remédiation lorsque des faiblesses sont détectées. Dans la pratique, ces outils d'analyses statiques et d'analyses dynamiques sont exécutés durant la phase de construction (*Build*) et la phase de *Test* du cycle *DevOps* (Figure 1.4).

4. La figure est basée sur une création provenant de https://freepik.com

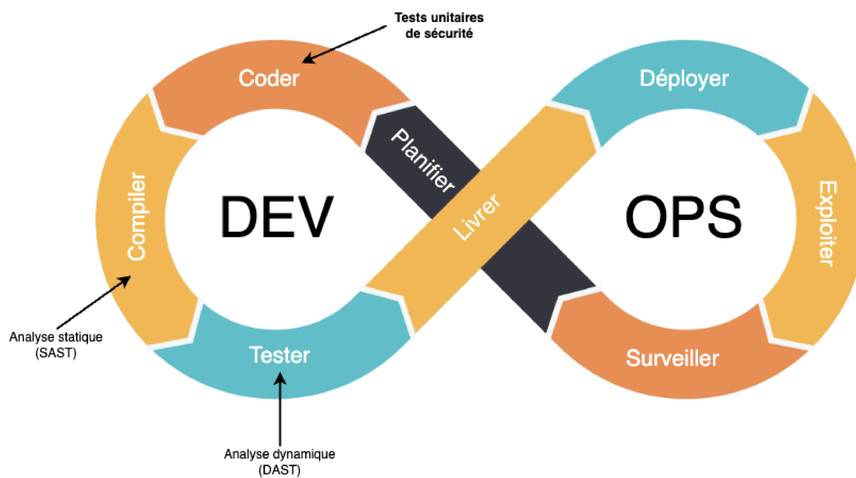

FIGURE 1.4 : Cycle DevOps [4]

Les tests unitaires se concentrent sur le comportement des unités isolées, c'est-à-dire les classes et méthodes, tandis que les tests d'intégration vérifient les interactions entre plusieurs unités ou composants. Une collection de tests unitaires et d'intégration peut être exécutée automatiquement dans le cadre d'un pipeline d'intégration continue (*Continuous Integration (CI)*) afin de tester les modifications du code source au fur et à mesure qu'il est soumis. Ils peuvent être également exécutés manuellement par les développeurs au fur et à mesure de la mise en oeuvre de la portion de code (Gonzalez *et al.*, 2021). Il est donc approprié d'admettre que les tests unitaires de sécurité peuvent être exécutés durant la phase de rédaction du code (Figure 1.4).

Comme mentionné précédemment, peu de recherches semblent se concentrer sur les tests unitaires dans un contexte de sécurité. Cela semble également le cas au niveau de l'industrie. Par conséquent cette nouvelle approche pourrait ouvrir de nouveaux horizons de prévention et de protection. Contrairement aux solutions de protection actuellement disponibles, lesquelles peuvent également être dispendieuses, notre nouvelle méthode concentrerait la responsabilité sur les équipes de développement, durant la phase de programmation, en offrant des méthodes

et des outils permettant d'assurer un meilleur contrôle du code source avant le passage aux stades suivant du cycle *DevOps*. Professionnellement parlant, les développeurs qui rédigent des scénarios et exécutent des tests unitaires représentent la première, et la plus efficace, ligne de défense (Bland, 2014).

1.5 CONTRIBUTION

Les projets de recherche jouent un rôle essentiel dans l'avancement des connaissances et la résolution de problèmes complexes. Leurs contributions se manifestent à plusieurs niveaux, tant sur le plan académique que pratique. Ce projet contribue à l'expansion des frontières du savoir en explorant une nouvelle idée, en développant une approche novatrice. Sur le plan pratique, la démarche apporte une solution concrète aux défis rencontrés en sécurité du développement.

Ce projet de recherche présente les contributions suivantes :

1. Une revue de la littérature sur les pratiques au niveau des tests unitaires et la sécurité informatique ;
2. Une démarche pour prévenir les attaques par injection SQL, en utilisant les tests unitaires ;
3. Une série de scénarios de validation visant à prévenir ces types d'attaques

1.6 ORGANISATION DU DOCUMENT

Ce document est divisé en six chapitres. Ce chapitre d'introduction montre l'état actuel de la sécurité, des menaces et des enjeux relatifs au problème de protection des logiciels. Le second chapitre présente la revue de la littérature en abordant l'état de l'art dans le domaine de la sécurité et des tests au niveau des logiciels. Le troisième chapitre couvre la

démarche au niveau de la méthodologie et des stratégies à apporter dans un contexte de protection par des tests unitaires. Le quatrième chapitre couvre les résultats et l'efficacité sur les différentes techniques de prévention. Le cinquième chapitre couvre l'implication de cette nouvelle approche pour les équipes de développement logiciel, l'apport au niveau de la pratique et les limitations de la recherche. Finalement, le sixième chapitre résumera le projet de recherche et la contribution au domaine du développement logiciel.

CHAPITRE II

REVUE DE LA LITTÉRATURE

Depuis 2007, l'*Open Web Application Security Project* (OWASP) a identifié les injections de code malveillant comme les vulnérabilités les plus fréquemment exploitées lors des attaques. De 2010 à 2017, les injections de code ont occupé la première place du classement [5]. Plus récemment, dans l'édition de 2021 du *Top 10 OWASP*, les injections de code malveillant sont encore dans le classement des vulnérabilités les plus exploitées. Malgré la descente de celles-ci au troisième range du classement (A3 :2021), en dessous des *Violation des contrôles d'accès* (A1 :2021) et des *Défaillances cryptographiques* (A2 :2021) [6], elles représentent une défaillance importante au niveau des systèmes d'information (Aljabri *et al.*, 2022).

Un système est attaqué par l'exploitation de ses vulnérabilités. Une vulnérabilité se définit comme une faiblesse ou une faille présente dans un système, un logiciel, ou une infrastructure, susceptible d'être exploitée à des fins d'attaque. L'assaillant se sert donc de ces vulnérabilités comme d'un point d'entrée pour compromettre le système et ainsi accéder à des informations confidentielles ou encore infliger divers autres types de dommages. De façon générale, une attaque par injection se caractérise par une action malveillante visant à exploiter les failles d'un logiciel en injectant du code, ce qui entraîne la génération ou la modification de programmes en sortie. En somme, en insérant une entrée malveillante dans un logiciel, un attaquant peut amener ce dernier à produire un comportement inadéquat (Ray & Ligatti, 2014).

---

5. History of OWASP TOP 10. Récupéré en septembre 2023 de https://www.hahwul.com/cullinan/history-of-owasp-top-10/

6. OWASP Top10. Récupéré en septembre 2023 de https://owasp.org/Top10/

Il existe plusieurs catégories d'injection [7]. Certaines s'exécutent au niveau de la couche de l'interface utilisateur et d'autres sur des couches inférieures du système. On retrouve principalement les injections sous cinq catégories :

1. *CrossSite Scripting* ou XSS : Une *XSS* est une forme d'attaque où un attaquant injecte et exécute du code malveillant, généralement du code JavaScript, dans des pages HTML consultées par des utilisateurs depuis des navigateurs internet (Hydara *et al.*, 2015).
2. *Server-Side Includes Injection* ou SSI : Les *SSI* sont des directives présentes sur les applications internet utilisées pour alimenter une page HTML avec un contenu dynamique. Ils sont similaires aux *Common Gateway Interface* (CGI), cependant ils sont utilisés pour exécuter certaines actions avant le chargement de la page ou durant sa visualisation. L'attaque par injection *SSI* permet l'exploitation d'une application internet en injectant des scripts dans des pages HTML [8].
3. SQL injection : Une attaque par injection SQL [9] consiste à l'insertion ou « l'injection » d'une requête SQL malveillante à partir d'un processus de saisie de données provenant d'une entrée d'un utilisateur vers un système d'information ou une application, comme un formulaire électronique par exemple. Une exploitation réussie permet, entre autres, de lire des données sensibles de la base de données, de modifier les données de la base de données, d'exécuter des opérations d'administration sur la base de données (comme la création ou la destruction d'une table), de récupérer le contenu d'un fichier de données présent sur le système d'exploitation, de générer un déni de service ou dans certain cas, d'envoyer des commandes au système d'exploitation qui supporte le système de gestion de la base de données (Halfond *et al.*, 2006).

---

7. A03 :2021 – Injection. Récupéré en septembre 2023 de https://owasp.org/Top10/A03_2021-Injection/

8. Server-Side Includes (SSI) Injection. Récupéré en mai 2023 de https://owasp.org/www-community/attacks/Server-Side_Includes_(SSI)_Injection

9. SQL Injection. Récupéré en mai 2023 de https://owasp.org/www-community/attacks/SQL_Injection

4. XML ou XPath Injection : Similaires aux injections SQL, les attaques par injection *XPath* se produisent lorsqu'un site internet utilise des données fournies par l'utilisateur pour construire une requête *XPath* à partir de données XML. En envoyant intentionnellement des informations mal formées au site internet, un attaquant peut découvrir comment les données XML sont structurées ou accéder à des données auxquelles il n'a normalement pas accès. Il est également possible d'effectuer une escalade de privilèges si les données XML sont utilisées pour l'authentification d'un utilisateur [10].

5. Injection de commandes (*Commands Injection*) : L'injection de commandes est une attaque dont le but est l'exécution de commandes arbitraires sur le système d'exploitation hôte, en passant par le système d'information ou l'application. Les attaques par injection de commande sont possibles lorsqu'une application transmet des données fournies par l'utilisateur à un système *shell* (formulaires, témoins, en-têtes HTTP, etc.) [11].

2.1 RISQUE ET APERÇU DES ATTAQUES PAR INJECTION SQL

Les injections SQL ont été documentées pour la première fois en décembre 1998, par Jeff Forristal, un spécialiste en cybersécurité [12]. Forristal avait découvert celles-ci et publié un article dans le magazine *Phrack*. Forristal décrivait un serveur SQL récupérant des données potentiellement sensibles grâce à l'utilisation de commandes spécifiques au niveau des entrées des utilisateurs, lesquelles permettaient d'altérer le comportement normal de la requête. C'est en 2002 que les injections SQL ont semblé être portées à l'attention de la communauté de la sécurité et que celles-ci sont devenues une vulnérabilité connue (Hyslip, 2017). Durant cette même année, Jeremiah Jacks, découvre que Guess.com était vulnérable à une attaque par

10. XPATH Injection. Récupéré en mai 2023 de https://owasp.org/www-community/attacks/XPATH_Injection

11. Command Injection. Récupéré en avril 2023 de https://owasp.org/www-community/attacks/Command_Injection

12. J Forristal LLC. 2023. Récupéré de https://www.forristal.com

injection SQL, permettant à toute personne capable de créer une *URL* adéquatement forgée, d'extraire plus de 200 000 noms, numéros de carte de crédit et dates d'expiration depuis la base de données des clients du site internet (Poulsen, 2002).

Il ne semble pas exister de réponse évidente à la question suivante : Pourquoi les injections SQL sont apparues sur la place publique en 2002 ? Cependant, il serait juste de croire que des publications comme celle de Jeff Forristal, en 1998 et Jeremiah Jacks, en 2002 ont suscités les intérêts de la communauté.

L'*OWASP* affirme qu'une attaque par injection SQL consiste à insérer, ou plutôt injecter, une portion de code SQL malveillant à partir d'une entrée de données provenant normalement d'un utilisateur [13]. Selon Ray et Ligatti, la définition d'une injection de code, dont une injection SQL, est basée sur la propriété *Noncode Insertions or Expansions*, qui stipule que les entrées non fiables d'une application ne doivent produire que des insertions ou des extensions non codées (NIE) au niveau de la sortie des programmes. Autrement dit, lorsque les applications génèrent des sorties, telles que des requêtes SQL, basées sur des entrées non fiables, la propriété NIE exige que les entrées n'affectent que la sortie du programme en insérant ou en prolongeant des symboles non codés (Ray & Ligatti, 2014). Lors d'une saisie d'information dans un formulaire électronique, tel qu'un formulaire de recherche ou un formulaire d'authentification par exemple, l'utilisateur malveillant insère une portion de code SQL qui brise ou détourne la requête normale et modifie ainsi le comportement légitime du formulaire. De cette manière, si aucun mécanisme de protection n'est présent dans les différentes couches du logiciel pour valider les intrants de l'utilisateur, le code malveillant est transmis au travers du système jusqu'à la base de données et modifie l'exécution de la requête. À terme, dans notre exemple, le code injecté permet de contourner le mécanisme naturel d'une authentification d'un usager

13. A03 2021 - Injection. Récupéré en janvier 2024 de https://owasp.org/Top10/fr/A03_2021-Injection/

ou encore, permet de récupérer des informations qui normalement ne devraient pas être présentes dans les résultats attendus d'une recherche d'information.

On retrouve également une définition des injections SQL dans la *Common Weakness Enumeration* (CWE). CWE est un référentiel de vulnérabilités logicielles et matériels maintenu par une communauté de spécialistes. Ce dernier sert de base de connaissances, de mécanisme de mesure pour les outils de sécurité et de référence au niveau de l'identification, de l'atténuation et de la prévention des vulnérabilités [14]. Sous CWE, la définition des injections SQL se retrouve sous l'identification *CWE-89 : Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')* [15], laquelle énumère les faiblesses d'un système qui permettent l'injection de code SQL. Selon la description de *CWE-89*, la vulnérabilité résulte de la construction complète ou partielle d'une commande SQL à l'aide d'une entrée provenant de l'extérieur (du système) à partir d'un composant en amont, qui n'a pas neutralisé adéquatement les caractères spéciaux, lesquelles peuvent modifier la requête SQL lorsqu'elle est envoyée à un composant en aval (Figure 2.1). Également, selon le *TOP 25 des erreurs logicielles les plus dangereuses* produit par la *SANS Institute* [16], la vulnérabilité *CWE-89* est au troisième rang des erreurs de logiciels, juste au-dessus du *CWE-20 : Improper Input Validation* [17].

---

14. Mitre CWE. 2023. Récupéré de https://cwe.mitre.org/index.html

15. CWE 89. Récupéré en avril 2023 de https://cwe.mitre.org/data/definitions/89.html

16. CWE TOP 25 Most Dangerous Software Errors. Récupéré en mars 2023 de https://www.sans.org/top25-software-errors/

17. CWE 20. Récupéré en avril 2023 de https://cwe.mitre.org/data/definitions/20.html

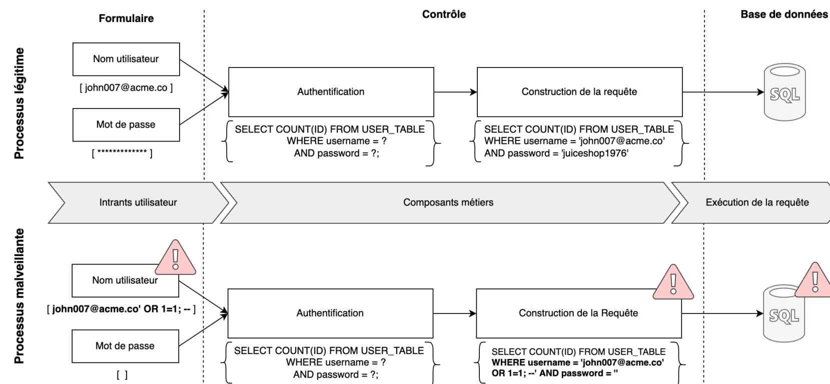

FIGURE 2.1 : Exemple d'un processus de création d'une requête légitime versus la création d'une requête malveillante contenant une injection SQL.

En plus d'OWASP, il existe une seconde liste de vérification (*Checklist*) mettant en lumière les vulnérabilités les plus courantes et leurs méthodes de mitigation. Similaire au *Top 10 OWASP*, cette liste est issue de la *SANS Institute* et se nomme la *Securing Web Application Technologies Checklist* ou plus couramment utilisée sous l'acronyme *SWAT Checklist*. On retrouve cette liste sous la rubrique *Cloud Security* de leur site internet [18]. Tout comme OWASP, cette liste se veut être *un ensemble de bonnes pratiques faciles à consulter qui sensibilisent et aident les équipes de développement à créer des applications plus sécurisées*. Toujours selon la *SANS Institute*, *il s'agit d'une première étape vers la constitution d'une base de connaissances en matière de sécurité autour de la sécurité des applications*. Sous la *SWAT Checklist*, on retrouve les injections SQL sous deux sections, c'est-à-dire sous *Input and Output Handling* et sous *Configuration and Operations*.

Il est essentiel de noter que ce type d'attaque est indépendant du langage utilisé lors de la rédaction du logiciel. Une attaque par injection SQL peut être effectuée au travers des logiciels rédigés sous différents langages. Notamment en Java, en PHP, en JavaScript ou en ṅet. Dès lors que le système de gestion de bases de données (SGBD) est de type SQL, le

18. Securing Web Application Technologies [SWAT] Checklist. Récupéré en avril 2023 de https://www.sans.org/cloud-security/securing-web-application-technologies/

code malveillant peut s'exécuter. Néanmoins, quelques variations peuvent exister en fonction de l'éditeur du langage SQL. Par exemple, il est possible qu'une attaque par injection SQL rédigée pour une base de données sous Transac-SQL (Microsoft) ne soit pas fonctionnelle sous PL-SQL (Oracle).

Voici quelques exemples d'injection SQL d'ordre générique [19] :

Injection SQL - Tautologie : `' OR 1=1`

```
SELECT COUNT(ID) FROM USER_TABLE
    WHERE usename = 'john007@acme.co'
    OR 1=1; --' AND password = '';
```

Listing 2.1 – Contournement d'une connexion

Dans cet exemple, cette injection de code SQL permet de contourner le mécanisme normal d'authentification d'un système d'information ou d'une application internet, en forçant la clause *WHERE* à retourner toujours vraie, peu importe la condition normale (tautologie).

Injection SQL - Destruction d'une table : `'; DROP TABLE USER_TABLE`

```
SELECT * FROM MUSIC_TABLE
    WHERE music_title = ''; DROP TABLE USER_TABLE;
```

Listing 2.2 – Destruction d'une table

Dans cet exemple, l'injection de code SQL permet de détruire la table nommée *USER_TABLE* de la base de données. Par une requête qui suit immédiatement la requête normale (*Piggy-*

19. Il ne semble pas exister de définition ni de référence claire d'une requête dite générique d'injection SQL. Cependant, nous pourrions la définir comme étant une requête qui est en mesure de s'exécuter avec succès sous n'importe quel type de système de gestion de base de données (SGBD), indépendamment de la structure du langage utilisé pour effectuer l'action malveillante

backed), cette dernière vient altérer la structure de la base de données en supprimant une table. Cette attaque pourrait causer un déni de service du système, puisque les données des utilisateurs deviennent introuvables.

Injection SQL - Union :`' UNION SELECT username FROM USER_TABLE;--`

```
SELECT title FROM MUSIC_TABLE
    WHERE music_author_name LIKE '%elvis%'
    UNION username FROM USER_TABLE;
```

Listing 2.3 – Saisir la liste des utilisateurs

Dans cet exemple, l'injection de code SQL permet non seulement de récupérer la liste des titres de chansons dont le nom de l'auteur contient le mot *elvis*, mais il permet également de récupérer la liste des noms d'utilisateurs de l'application.

Selon *CVE Details* (*Common Vulnerabilities and Exposures*), un portail en ligne créé par Serkan Özkan qui héberge une base de données sur les vulnérabilités (Özkan, 2017), il y aurait plus de 12 000 vulnérabilités de type *injection SQL* identifiées et répertoriées jusqu'à maintenant [20]. La Figure 2.2 montre un diagramme démontrant l'évolution du nombre de vulnérabilités par injections SQL qui ont été répertoriées depuis 2013 [21].

---

20. CVEs (Sql injection). Récupéré en juin 2024 de https://www.cvedetails.com/vulnerability-list/opsqli-1/sql-injection.html

21. CVE Details. 2023. Récupéré de https://www.cvedetails.com/

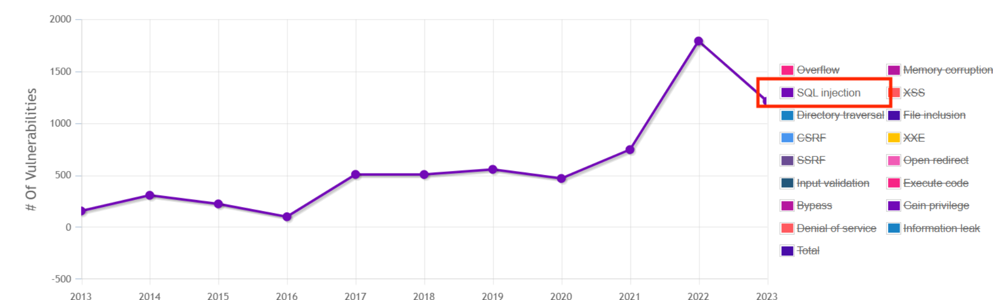

FIGURE 2.2 : Diagramme indiquant le nombre de vulnérabilités liées aux injections SQL de 2013 à 2023. Source : CVE Details

2.2 TECHNIQUES ET MEILLEURES PRATIQUES POUR CONTRER LES INJECTIONS SQL

Les attaques informatiques par injection SQL peuvent être classifiées par intention (ou motif), c'est-à-dire en fonction des objectifs et de la motivation de l'attaquant. Dans un article de conférence paru en novembre 2015, Yasser Gomaa et al. nous font part des résultats d'un sondage intitulé *Survey on Securing a Querying process by Blocking SQL injection*, dans lequel on retrouve l'inventaire des intentions des assaillants relatifs aux attaques par injection SQL, ainsi que les techniques d'attaques et les contre-mesures (Gomaa *et al.*, 2015). À noter que lors d'une attaque, les assaillants peuvent avoir une ou plusieurs intentions.

Les auteurs ont défini les intentions comme suit :

1. Identification des champs vulnérables : Il s'agit de déterminer quels champs d'un formulaire sont vulnérables à des injections de code malveillant. Cela consiste principalement à tester manuellement les différents champs, en injectant des requêtes ayant des impacts mineurs.
2. Déterminer le ou les schémas de la base de données : L'attaquant tente de récupérer des informations sur la base de données, ainsi que les tables et relations qui la composent ;

3. Contournement de l'authentification : Tel que le titre le mentionne, l'attaquant tente de s'authentifier sans devoir connaître le mot de passe de l'utilisateur. Ainsi l'attaquant accède au système d'information, à partir d'un accès non autorisé ;
4. Extraction de données : L'attaquant tente d'extraire des données. Il s'agit de l'attaque par injection SQL la plus commune ;
5. Ajout ou modification de données : On tente ici d'ajouter des informations dans des tables ou encore d'altérer leur contenu ;
6. Deni de service : L'attaquant souhaite rendre les interrogations à la base de données difficiles, voire impossibles. En conséquence, le système d'information ne répond plus aux requêtes des utilisateurs ;
7. Évitement des mécanismes de détection : Ce type d'attaque tente d'éviter de se faire détecter par des systèmes de détection d'intrusion ;
8. Escalade de privilège : L'attaquant tente de récupérer des droits supérieurs en personnifiant (*Impersonate*) un utilisateur à haut privilège ;
9. Téléchargement de fichier : Cette méthode d'attaque tente de récupérer un fichier stocké sur le serveur hôte. Ce fichier pourrait contenir des informations techniques pertinentes pouvant servir à une autre attaque.
10. Téléversement de fichier : Cette méthode d'attaque tente d'envoyer un fichier du côté du serveur hôte, lequel pourrait être utilisé lors d'une prochaine étape.

Lorsque les intentions sont définies, il est nécessaire d'identifier les techniques d'attaques pouvant être utilisées par l'assaillant afin d'exploiter les vulnérabilités du système et ainsi répondre à l'une ou l'autre de ses intentions. Les méthodes et techniques d'attaques établissent la structure du code à composer pour créer une charge utile (*Payload*) à envoyer au système ciblé, afin de répondre aux objectifs souhaités.

Les auteurs ont défini ces techniques comme suit :

1. Par tautologie : Il s'agit de faire en sorte qu'une requête normale ou légitime soit toujours vraie, peu importe la condition. Cette technique est principalement utilisée pour contourner les mécanismes d'authentification des utilisateurs.
2. Par commentaire : L'attaquant s'assure qu'une fois que son code malveillant est injecté, s'il reste du code légitime à exécuter, ce dernier sera tout simplement placé sous la forme d'un commentaire, de sorte que l'interpréteur du langage SQL ne considère pas celui-ci lors de son exécution.
3. Par une requête d'union : En ajoutant la commande *UNION* à une requête légitime de saisie d'information, il pourrait être possible de récupérer des données supplémentaires, qui normalement ne devraient pas être récupérées. Ces données supplémentaires seraient ajoutées aux résultats de la requête.
4. Une requête adossée ou *PiggyBacking* : L'attaquant ajoute une requête malveillante à la suite de la requête légitime. Ainsi, il est possible de faire en sorte que cette requête malveillante s'exécute à la suite de la requête légitime, sans pour autant interrompre celle-ci.
5. Les procédures stockées du système : Il s'agit d'exploiter des procédures stockées venant par défaut avec le SGBD. Ces procédures peuvent à leur tour contenir des vulnérabilités exploitables par une attaque de type injection SQL. À noter que selon les auteurs, une procédure stockée pourrait même générer une attaque de type *BufferOverflow* [22] ou une attaque de type *Escalade de privilèges* si celle-ci n'est pas adéquatement protégée.

Dans une modélisation de menaces, lorsque l'intention d'un attaquant et les techniques pour effectuer une attaque sont déterminées, il est impératif d'établir les contre-mesures néces-

22. Méthode de surcharge d'une partie prédéfinie de la mémoire tampon, permettant potentiellement d'écraser et de corrompre la mémoire au-delà des limites de ce tampon (Deckard, 2005)

saires afin d'éviter que les charges utiles (code malveillant) n'atteignent la base de données et que celles-ci soient exécutées. Les contre-mesures sont des mécanismes de protection qui empêchent qu'une attaque se solde par un succès. Dans le sondage, les auteurs énoncent une liste de contre-mesures, lesquelles minimisent ou annulent complètement l'effet de la charge utile. Il est possible de les appliquer en partie ou en totalité. À noter que ces contre-mesures peuvent être appliquées non seulement au niveau du code source du logiciel, mais également au niveau des paramètres de la base de données.

Les auteurs ont défini les contre-mesures comme suit. À noter que la validation des entrées utilisateurs a été ajoutée à cette liste :

1. Paramétrisation des requêtes : Il s'agit d'utiliser les fonctions propres au langage de programmation qui passent les intrants utilisateurs sous forme de paramètres à la requête et non sous forme de chaîne de caractères. Selon Galluccio *et al.* (2020), un moyen de contrer les injections SQL consiste à utiliser des requêtes dites paramétrées. La principale raison derrière cela est que l'entrée n'est jamais envoyée à la base de données en l'état, c'est-à-dire sous forme de chaîne, comme c'est le cas dans la création de chaînes dynamiques, mais elle est plutôt sérialisée et stockée dans des paramètres séparés (d'où le nom). Cela se fait en utilisant des variables lors de la construction de l'instruction SQL, en utilisant des identifiants comme espaces réservés afin que la chaîne réelle puisse être construite en toute sécurité.
2. Moindre privilège : L'approche vise uniquement à fournir les accès nécessaires à la réalisation de la ou les tâches au niveau de la base de données.
3. Séparation des tâches : L'approche vise à séparer les tâches et fonctions d'un rôle. Par exemple, un compte de service qui permet la saisie d'informations dans la base de données ne pourrait pas, du même coup, effectuer un ou des modifications sur cette même base de données.

4. Personnalisation des messages d'erreur : Éviter de transmettre les messages d'erreur détaillée provenant du SGBD. Puisque l'objectif est de minimiser les informations révélées à l'utilisateur, il est souhaitable d'indiquer que le système d'information a rencontré une erreur et qu'il est nécessaire de communiquer avec l'administrateur du système.
5. Retrait des procédures stockées du système [23] : Il s'agit de retirer ou de désactiver les procédures stockées présentes par défaut dans la base de données, mais non nécessaires. Cela réduit les risques que celles-ci soient utilisées à des fins malveillantes.
6. Échappement des mots clés SQL (*Escaping*) : Cette approche permet de retirer la possibilité aux utilisateurs d'insérer des mots-clés ou des caractères clés appartenant typiquement au langage SQL. Il s'agit donc d'invalider ces mots, s'ils ne respectent pas le patron souhaité.
7. Validation des intrants des utilisateurs : Il s'agit de valider systématiquement les intrants provenant des utilisateurs en couvrant à la fois la forme et le contenu. Par exemple, un intrant n'ayant pas la forme attendue (telle que la dimension) ou le contenu (tel que le type de valeur comme un nombre entier) serait invalide et rejeté par le système.

La revue de code est également une mesure de sécurité, contre les vulnérabilités. Ces dernières sont des évaluations méthodiques du code source, conçues pour identifier les anomalies et améliorer la qualité de la solution. Ils sont préférablement effectués par des personnes, autres que les développeurs. Tout comme les tests, la revue de code source est un processus de contrôle qualité qui assure la conformité d'une solution. Selon le *Code Review Guide* d'OWASP (Conklin & Robinson, 2017), lors d'une revue de code, il est nécessaire d'effectuer les tâches relatives aux injections de code suivantes :

23. Une procédure stockée est un code SQL préparé qu'il est possible d'enregistrer dans la base de données, afin que celui-ci puisse être réutilisé (Foggon, 2006)

1. Valider systématiquement une entrée provenant d'un utilisateur en testant le type, la longueur, le format et la portée (*Range*).
2. Tester la taille et le type de données de l'entrée et s'assurer des limites appropriées.
3. Tester le contenu des variables de type chaîne de caractères *String* et autoriser uniquement les valeurs qui sont attendues. Rejeter les entrées contenant des données binaires, échapper les séquences et les caractères qui sont utilisés pour faire des commentaires dans le code source.
4. Lorsque des documents XML sont impliqués, il faut valider toutes les données par rapport à leur schéma lors de leur saisie.
5. Ne pas créer d'instructions SQL directement à partir d'une entrée utilisateur.
6. Utiliser des procédures stockées pour valider les entrées provenant des utilisateurs.
7. Implémenter plusieurs couches de validation au niveau du système d'information (Défense en profondeur).
8. Ne pas concaténer les chaînes de caractères provenant d'une entrée utilisateur qui n'a pas été validée.
9. Invalider toute instruction, depuis une entrée utilisateur, qui appelle des commandes telles que *EXECUTE*, *EXEC* ou toutes autres instructions SQL qui pourraient appeler des ressources du système, tel que des lignes de commandes du système d'exploitation par exemple.

2.3 LES TESTS UNITAIRES ET LEURS UTILITÉS DANS UN CONTEXTE DE PRÉVENTION CONTRE LES INJECTIONS SQL

Les tests (ou essais) sur les logiciels sont une étape nécessaire dans le processus de développement. Ces derniers représentent les méthodes et mécanismes d'évaluation des

systèmes d'information. Ils repèrent notamment les défauts, les erreurs et les bogues afin de s'assurer que les systèmes répondent aux exigences prévues et fonctionnent selon les attentes. Malgré que la détection et la découverte de ces derniers soient une partie essentielle de ce processus, ce n'est pas le seul objectif. Les tests s'assurent également que le logiciel soit évolutif, sécuritaire et efficace (Leloudas, 2023).

Voici quelques catégories de tests effectués dans le cadre du développement d'un logiciel :

1. Tests de non-régression : Les tests de non-régression s'assurent que les modifications apportées au logiciel n'affectent pas la fonctionnalité des composants qui ont été altérés.
2. Tests de sécurité : Les tests de sécurité prennent normalement la forme de test d'intrusion, de tests statiques qui valident les vulnérabilités dans le code source et les tests dynamiques qui confirment que le comportement du composant n'a pas de vulnérabilité et n'est pas malveillant.
3. Tests unitaires : Comme mentionné dans le Chapitre 1, ils permettent de valider la fonctionnalité d'une portion de code en se concentrant sur le comportement des unités isolées (les classes et méthodes).
4. Tests d'intégration : Quant aux tests d'intégration, ils valident les interactions entre plusieurs unités et/ou composants. Par exemple, un test d'intégration pourrait combiner une suite de tests unitaires, permettant ainsi de tester de bout en bout un processus métier.

Les publications sur les tests unitaires ou sur les injections SQL semblent courantes. Des travaux académiques et professionnels sur l'un ou l'autre de ces sujets sont présents dans plusieurs plateformes de publications. Cependant, il semble en être tout autrement au niveau

de l'unification des deux sujets, c'est-à-dire de l'utilisation des tests unitaires dans un contexte de prévention contre les injections SQL.

Dans son livre intitulé *Unit Testing - Principles, Practices, and Patterns*, Vladimir Khorikov définit les tests unitaires comme étant des tests automatisés ayant trois attributs (Khorikov, 2020) :

1. Ils doivent vérifier des petites portions de code source appelées *unités*
2. Ils doivent s'effectuer rapidement
3. Ils doivent se produire de manière isolée

Les tests unitaires représentent un moyen pour les développeurs de logiciel de s'assurer que les fonctions et portions de codes qui ont été produites répondent aux besoins pour lesquels elles ont été créées. En somme, il s'agit de tester la fonctionnalité de la portion de code qui a été rédigée. Essentiellement, les tests unitaires peuvent être créés afin de vérifier qu'une chaîne de caractère provenant d'une entrée utilisateur respecte une certaine longueur. Ils peuvent également s'assurer qu'un calcul complexe réponde aux attentes, ou plus simplement s'assurer que le contenu d'une variable respecte une certaine structure pour être valide. Toujours selon Khorikov, l'objectif des tests unitaires est de permettre une croissance durable du projet de développement du système. La Figure 2.3 présente la différence de croissance entre un projet avec et sans tests unitaires (Khorikov, 2020).

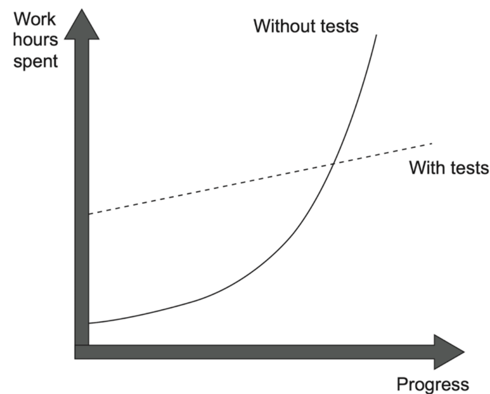

FIGURE 2.3 : Croissance d'un projet avec et sans tests unitaires. Reproduit avec la permission de Manning. Source : Khorikov

De plus, Khorikov mentionne dans son livre qu'un test unitaire est efficace lorsqu'il est basé sur quatre valeurs fondamentales, c'est-à-dire :

1. La protection contre la régression : Une régression survient lorsqu'une fonctionnalité cesse de fonctionner suite à un événement, tel que la modification du code source.

2. La résistance au réusinage du code source (*Refactoring*) : Il s'agit d'effectuer des modifications au niveau du code source, sans changer le comportement de la portion de code modifié. Cette pratique est utilisée, entre autres, pour réduire la complexité et améliorer la clarté du code.

3. La rétroaction rapide : il est possible de réduire significativement le temps de réponse et de correction, lorsqu'une défaillance (bogue) survient au moment de la construction et de l'exécution du test, puisqu'elle signale très tôt dans le processus qu'une anomalie est survenue.

4. La facilité de la maintenance du code source : puisque les tests unitaires sont exécutés à chaque fois que des modifications sont effectuées, il devient aisé d'effectuer des

améliorations sans défaillances fonctionnelles, telles que l'ajout de nouvelles spécificités par exemple.

Dans une perspective de sécurité, les deux valeurs de Khorikov qui nous paraissant les plus appropriées dans le cadre de ce projet sont la **protection contre la régression** et la **rétroaction rapide**. D'une part, puisque le développeur est en mesure de s'assurer que le code source qui a été ajouté ou modifié préserve la contre-mesure, et d'une autre part, puisqu'il est possible de recevoir rapidement un retour sur l'état de la composante testée, il est concevable qu'il continue de répondre aux attentes de sécurité. Il n'en demeure pas moins que les deux autres valeurs sont également pertinentes, mais moins significatives dans l'objectif de notre démarche.

CHAPITRE III
MÉTHODOLOGIE SCIENTIFIQUE

3.1 JUSTIFICATION DE LA DÉMARCHE ET SYNTHÈSE DE LA MÉTHODE

Au fil des années, les organisations telles que les institutions académiques ont développé des idées, des approches et des mesures afin d'améliorer la sécurité des logiciels. Sur le plan de la sécurité, des outils d'encadrement tels que le *Secure Software Development LifeCycle (SSDLC)* [24], le *Secure Development LifeCycle (SDL)* [25], le standard *ISO/IEC 27034* [26], le *NIST Secure Software Development Framework (SSDF)* [27] et le *Software Assurance Maturity Model (SAMM)* [28] pour n'en nommer que quelques-uns, ont su offrir un appui pour améliorer la pratique du domaine. De plus, en fonction des nouvelles tendances technologiques et des environnements opérationnels tels que le *Cloud Computing* et l'intelligence artificielle, nous avons vu émerger de nouveaux cadres tels que le *CSA Security Guidance* (Guidance, 2017) et le standard *ISO/IEC 27090 - Guidance for addressing security threats and failures in artificial intelligence systems* [29].

Il est communément admis que l'amélioration de la qualité des logiciels est nécessaire. À partir d'approches basées sur les tests, des projets de recherche en génie logiciel ont pu offrir aux praticiens de nouvelles techniques qui améliorent la qualité et la productivité des

---

24. Secure Software Development LifeCycle (SSDLC). 2024. Récupéré de https://sdlc.uconn.edu/

25. Secure Development LifeCycle (SDL). 2024. Récupéré de https://www.microsoft.com/en-us/securityengineering/sdl/practices

26. ISO/IEC 27034. 2024. Récupéré de https://www.iso27001security.com/html/27034.html

27. NIST Secure Software Development Framework (SSDF). 2024. Récupéré de https://csrc.nist.gov/Projects/ssdf

28. OWASP SAMM. 2024. Récupéré de https://owasp.org/www-project-samm/

29. ISO/IEC CD 27090. 2024. Récupéré de https://www.iso.org/standard/56581.html

solutions (Daka & Fraser, 2014). Par ailleurs, l'amélioration des contrôles de la qualité par les tests unitaires fait partie de ces techniques. Tout comme les mesures de contrôle de la qualité destinées à assurer notamment, la maintenance, l'extensibilité et la clarté. La mise en place de mécanismes de contrôle pour préserver la sécurité au niveau du code source des systèmes d'information représente également une démarche d'assurance de cette qualité. Par ailleurs, il existe un standard couvrant spécifiquement les tests logiciels, il s'agit du standard *ISO 29119*[30]. Ce dernier collige plusieurs normes, dont la *Standard for Software Unit Testing (1008)* de la IEEE, parue pour la première fois en 1987 (277, 1986).

Notre méthode est autonome par rapport aux cadres existants, car elle constitue un niveau de contrôle intégré directement au processus de rédaction du code source. Une démarche par tests de sécurité, lesquels confirment que le code est exempt de vulnérabilités de type injection SQL, peut aisément s'insérer dans les pratiques, normes et standards actuels. Dans la littérature, nous remarquons qu'il existe peu d'informations qui combinent à la fois les tests unitaires et la protection contre les attaques aux injections. Comme indiqué précédemment, quelques ouvrages existent sur l'utilisation des tests unitaires dans un contexte de vulnérabilités de type *Cross-site Scripting (XSS)* (Mohammadi *et al.* (2018), Mohammadi *et al.* (2016)), mais aucun ne semble couvrir les injections SQL.

Nous pouvons établir, d'ordre général, que les diverses pratiques de test mettant à l'épreuve les systèmes d'information offrent une grande valeur au niveau du génie logiciel. Malgré le fait que les tests unitaires ne couvrent qu'une partie de ces pratiques, ils offrent pourtant une méthode de protection additionnelle contre les erreurs de programmation au moment de la rédaction du code. Même si certains contestent l'effet des tests unitaires sur la qualité du code en soi (Gren & Antinyan, 2017), il n'en reste pas moins que l'utilisation de ceux-ci demeure une pratique courante et valorisée par l'industrie. Selon Martin Fowler,

30. ISO/IEC/IEEE 29119-1 :2022. 2024. Récupéré de https://www.iso.org/standard/81291.html

un spécialiste du domaine : *Des tests [unitaires] imparfaits, exécutés fréquemment, sont bien meilleurs que des tests parfaits qui ne sont jamais écrits du tout* (Fowler, 2024).

On peut observer que certaines pratiques de tests sont utilisées pour valider des fonctionnalités de composants logiciels relatifs à la sécurité (Saad & Mitchell (2020), Gonzalez *et al.* (2021)).

Prenons par exemple les formulaires de connexion (*Login Box*) d'un système d'information. Des tests unitaires peuvent être mis en place pour tester le nombre de tentatives de connexions infructueuses, le nombre minimal et maximal de caractères à insérer dans les champs ou encore de valider que le mot de passe respecte la robustesse imposée. Bien qu'il s'agit de tests servant à vérifier que le composant respecte les exigences de fonctionnalité vis-à-vis la sécurité, ces derniers ne sont pas, à notre avis, des tests unitaires de sécurité en soi. Ils représentent plutôt des tests fonctionnels qui s'assurent que les règles métiers qui encadrent les fonctionnalités du composant soient conformes à la mission de ce dernier. Puisqu'il s'agit d'un module d'authentification d'un système d'information, il va de soi que les fonctionnalités sont attachées à la protection du système. Dans notre approche, les tests unitaires de sécurité viennent agir sur différents mécanismes de vérification des vulnérabilités afin de prévenir toutes tentatives d'attaque (par injection SQL), indépendamment de la nature et de la mission du composant du système d'information.

En établissant un plan de test, nous avons l'intention de créer une série de scénarios de test pour examiner divers cas d'injection SQL. Notre objectif est de mettre à l'épreuve les contrôles de sécurité visant à contrer ce type d'attaque. Nous cherchons à assurer une couverture suffisante pour évaluer la robustesse de pratique de sécurité mise en place.

Dans un premier ordre, nous suggérons d'adopter les recommandations proposées par le *Top10 OWASP A03 :2021* [31]. Il est impératif de mettre en place de manière systématique ces mesures de sécurité sur chaque composant chargé de recueillir des données entrées par les utilisateurs avant de les transmettre à la base de données (Figure 3.1). Ces mesures de sécurité peuvent être considérées comme les quatre (4) lignes de défense, protégeant le système contre d'éventuelles attaques par injection SQL.

1. La validation des intrants.
2. L'assainissement des mots-clés SQL.
3. L'échappement de caractères spéciaux (*Escaping*).
4. Le passage des requêtes paramétrées.

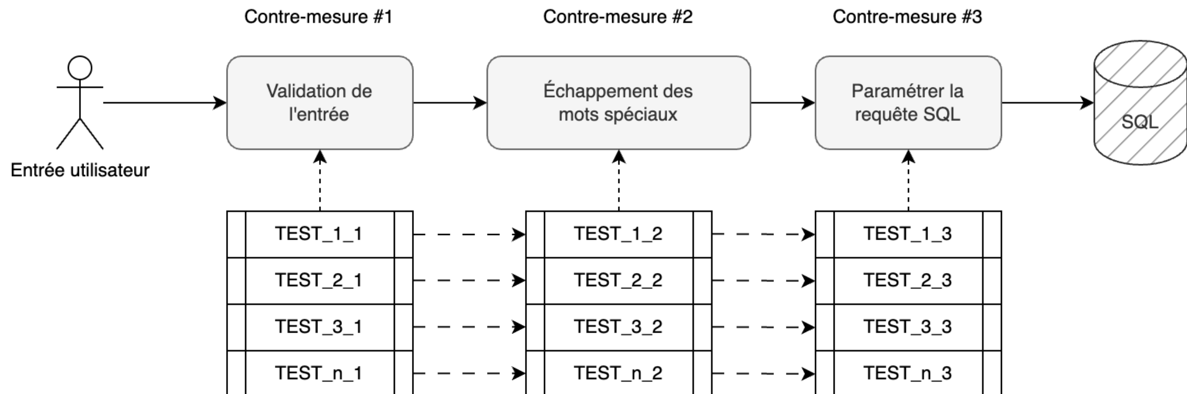

FIGURE 3.1 : Schéma des contre-mesures.

31. A03 2021 - Injection. Récupéré en janvier 2024 de https://owasp.org/Top10/A03_2021-Injection/

Dans un second ordre, nous proposons d'appliquer des contre-mesures de sécurité qui effectueront des vérifications uniquement au niveau de la paramétrisation de la base de données. Au besoin, des contre-mesures pourront également être définies pour tester les paramètres de sécurité sur une infrastructure infonuagique. Ci-dessous, les principales contre-mesures implémentées dans le projet, basées sur les travaux de (Gomaa *et al.*, 2015).

1. Principes de moindre privilège.
2. Séparation des tâches au niveau des comptes de service.
3. Retrait des procédures stockées non nécessaires.

En combinant ces contre-mesures aux motifs de l'attaquant, nous pouvons établir une stratégie de tests rigoureux et adéquats. Toutefois, il est à noter que même si des scénarios sont manquants, il sera possible de bonifier ces derniers avec de nouveaux cas d'utilisations, en fonction des besoins qui surviendront au fil du temps.

3.2 SYNTHÈSE DE LA MÉTHODE ET STRATÉGIE DE TEST

Une stratégie de tests implique, d'ordre général, un plan de test. C'est-à-dire un document qui décrit l'ensemble des tests à effectuer, leurs objectifs, les tâches, les étapes et les techniques nécessaires à leur réalisation. De plus, le plan permet d'identifier la portée des tests, de décrire les scénarios, d'établir l'horaire d'exécution, de prévoir les ressources et les budgets nécessaires (O'Regan, 2019). En ce qui nous concerne, notre approche ne vise pas à couvrir l'ensemble des pratiques de test au niveau des logiciels, mais uniquement de démontrer la faisabilité en ce qui a trait à l'utilisation des tests unitaires, dans un contexte de sécurité. C'est pourquoi, dans le cadre du projet, le plan de test s'articulera uniquement autour des trois éléments suivants :

1. La description de chacun des scénarios de test unitaire (cas d'usage).
2. L'identification de la portée du test unitaire (les intrants).
3. Les résultats attendus pour chacun des tests unitaires effectués.

3.3 PLAN DE TEST

Le plan de tests est un cadre méthodologique non seulement pour vérifier le bon fonctionnement des unités de code, mais également pour simuler des conditions diverses et évaluer la résilience de l'ensemble du système. Chacun des scénarios de tests unitaires de ce plan de test sera effectué sur une portion de code source et chacun de ceux-ci aura deux objectifs précis :

1. Tester une portion de code vulnérable.
2. Tester une portion de code sécurisée.

Les scénarios de tests unitaires seront effectués sur un modèle de données nommé *Chinook*. Ce modèle de données est disponible pour des systèmes de gestion de bases de données tels que *SQL Server*, *Oracle*, *MySQL* et *PostgreSQL*. Il peut être créé en exécutant un seul script SQL. La base de données *Chinook* est une alternative à la base de données *Northwind*, idéale pour les démonstrations et les tests d'outils [32]. Ce modèle de données met en scène une boutique de vente de musique numérique. Il inclut notamment les artistes, les albums de musiques, les pièces de musique, les clients et les commandes.

Dans le contexte de cette initiative, des modifications ont été effectuées sur le modèle de données afin de satisfaire nos besoins spécifiques. Par conséquent, nous avons élargi les

32. Github. Lerocha - Chinook Database. Récupéré en mai 2023 de https://github.com/lerocha/chinook-database

données en incluant des détails supplémentaires, tels que les comptes utilisateurs liés aux clients et les journaux d'événements (*logs*).

3.3.1 MÉTHODE DE CRÉATION DE SCÉNARIOS DE TESTS

Pour concevoir des scénarios de tests unitaires de sécurité, il est nécessaire d'adopter une approche basée sur les risques et les menaces. L'objectif étant de couvrir tous les potentiels vecteurs d'attaques.

Les étapes suivantes sont ainsi proposées :

Établir les points d'entrée

- Identifier les points d'entrée : Il est nécessaire de commencer par identifier et répertorier tous les points d'entrée du système par lesquels les données utilisateur sont saisies et transmises. Cela inclut par exemple les formulaires, les paramètres d'URL, les API ainsi que tout autre élément susceptible de recevoir des données injectées.
- Évaluer les points d'entrée identifiés : Il convient de classer ces points d'entrée en fonction de leur niveau de risque et de leur importance. Certains points d'entrée peuvent être directement exposés depuis le périmètre externe, tandis que d'autres peuvent nécessiter une connexion au système ou des droits d'accès administratifs. Par exemple, lors du processus d'authentification à partir d'un formulaire de connexion (*Login Box*) ou l'appel à des API uniquement accessible depuis des ressources provenant du périmètre interne d'un réseau. Il est ensuite essentiel de prioriser les points d'entrée les plus critiques ou les plus vulnérables aux attaques.

Intégrer les paramètres valides et non valides

- Scénarios avec paramètres valides : Il s'agit de rédiger des tests qui utilisent des entrées conformes aux attentes. Ces tests permettent de vérifier que le code réponde correctement lorsque les données saisies sont valides.
- Scénarios avec paramètres non valides : Il s'agit de créer des tests qui introduisent des données non valides ou malformées, comme des caractères spéciaux, des chaînes trop longues, des groupes de mots ou des types de données invalides. Ces tests sont conçus pour évaluer la résilience du code face à des entrées inattendues et potentiellement dangereuses.

Prévoir les comportements malveillants et non malveillants attendus

- Comportements malveillants et code non sécurisé : Si le code n'est pas sécurisé, les tests doivent vérifier si les entrées non valides peuvent provoquer des comportements malveillants, tels que des comportements liés aux attaques par injections SQL. Ces tests cherchent à identifier les vulnérabilités dans le code source.
- Comportements malveillants et code sécurisé : Si le code est sécurisé, les tests doivent s'assurer que les entrées non valides sont adéquatement filtrées et le cas échéant, rejetées. Empêchant ainsi toute exploitation malveillante. Le test doit retourner des informations appropriées en fonction des résultats des obtenus.
- Comportements non malveillants et code non sécurisé : Pour un code non sécurisé, même les entrées valides peuvent parfois provoquer des erreurs ou des comportements inattendus.
- Comportements non malveillants et code sécurisé : Les tests doivent confirmer que les entrées valides sont traitées correctement, sans générer de faux positifs en termes de sécurité. Le système doit réagir normalement et de manière prévisible lorsqu'il reçoit des données valides.

Rédaction des scénarios de tests

- Définition des scénarios de test : Pour chaque scénario de test, il s'agit de rédiger un ou des cas d'usage en incluant les paramètres d'entrée (valides et non valides) ainsi que les réponses ou les comportements attendus. Les tests seront effectués sur du code sécurisé et non sécurisé.
- Structure des scénarios : Décrire et classer les scénarios de manière claire, en séparant les tests destinés à évaluer la sécurité du code de ceux qui testent la fonctionnalité en général (ex. : test fonctionnel). De plus, nous proposons de rassembler les tests sous des catégories d'attaques, afin de faciliter l'analyse et le partage des résultats, en plus d'aligner ceux-ci sous les standards requis.

3.3.2 LES SCÉNARIOS DE TESTS UNITAIRES

Nous avons choisi de concevoir nos scénarios de tests unitaires en fonction des principales catégories d'attaques par injection SQL. Cette approche permet de structurer les tests de manière à couvrir efficacement les différentes techniques d'injection SQL identifiées (Halfond *et al.*, 2006).

CLASSE *BOOLEAN-BASED* (TAUTOLOGIE)

Scénario 1.1

Nom : Contournement du mécanisme d'authentification.

Description : Un utilisateur invité tente de se connecter à la boutique. Le test unitaire doit vérifier qu'il n'est pas possible qu'un utilisateur puisse se connecter à son environnement en contournant les contrôles de sécurité. Le test devra être effectué sur la portion de code source qui prend le nom de l'utilisateur et le mot de passe en paramètre. Ensuite, il doit

valider l'identité de l'utilisateur auprès de la base de données et s'assurer que l'authentification s'exécute sans contournement (*Auth Bypass*).

Résultat attendu :

- Code vulnérable : Absence de contrôle de sécurité. Un attaquant peut être en mesure de forger une requête qui lui permet de contourner le processus d'authentification à partir d'une tautologie. Selon les intrants et les attentes, une requête de la sorte est en mesure de se rendre au niveau de la base de données. Le test doit se conclure par un échec.
- Code sécurisé : Tous les contrôles de sécurité ont été implémentés au niveau du code source. Un attaquant qui tente de forger une requête malveillante verra celle-ci être interrompue au niveau de la validation du nom de l'utilisateur, puisque celui-ci ne respecte pas le format souhaité par les barèmes de contrôle. Le test doit s'assurer que la requête malveillante ne se rend pas à la base de données. Le test doit se conclure par un succès.

Prendre note qu'en cas de requête légitime, une valeur de zéro (0) ou un (1) sera retournée par la base de données.

Scénario 1.2

Nom : Recherche dans la liste d'achats d'un utilisateur et tous les autres utilisateurs.

Description : Un utilisateur peut effectuer une recherche dans sa liste d'achats de musique. Le test unitaire fait appel aux mécanismes qui retournent la liste complète des achats effectués par cet utilisateur. En aucun cas cette requête ne doit permettre de récupérer d'autres informations, par altération la requête. Le test doit être effectué sur la portion de code source qui prend le nom de l'utilisateur en paramètre.

Résultat attendu :

- Code vulnérable : Absence de contrôle de sécurité. Un attaquant peut être en mesure de forger une requête qui lui permet de récupérer l'ensemble de tous les achats de musiques de tous les utilisateurs présents dans la table *Orders*. Selon les intrants et les attentes, une requête de la sorte est en mesure de se rendre au niveau de la base de données. Le test doit se conclure par un échec.
- Code sécurisé : L'ensemble des contrôles de sécurité ont été mis en place au niveau du code source. Un attaquant qui tente de forger une requête malveillante verra celle-ci être interrompue au niveau de la validation du nom de l'utilisateur, puisque celui-ci ne respecte pas le format souhaité par les barèmes de contrôle. Le test doit s'assurer que la requête malveillante ne se rend pas à la base de données.

Prendre note qu'en cas de requête légitime, une valeur de zéro ou plus devrait être retournée par la base de données. Chacune des valeurs attendues doit être identifiée dans les intrants du test, pour chacun des utilisateurs testés. Le test doit se conclure par un succès.

Scénario 1.3

Nom : Recherche du profil d'un utilisateur.

Description : Un utilisateur peut consulter les informations attachées à son profil. Le test unitaire fait appel aux mécanismes de recherche qui retourne le profil d'un utilisateur à partir de son adresse courriel. En aucun cas, cette requête ne doit permettre de récupérer d'autres informations. Le test doit être effectué sur la portion de code source qui prend le nom de l'utilisateur en paramètre et retourne les informations de son profil.

Résultat attendu :

- Code vulnérable : Absence de contrôle de sécurité. Un attaquant peut être en mesure de forger une requête qui lui permet de récupérer l'ensemble de toutes les informations de tous les utilisateurs présents dans la base de données. Le test doit se conclure par un

échec, puisque selon les intrants et les attentes, une requête de la sorte est en mesure de se rendre au niveau de la base de données.

- Code sécurisé : L'ensemble des contrôles de sécurité ont été mis en place au niveau du code source. Un attaquant qui tente de forger une requête malveillante verra celle-ci être interrompue au niveau de la validation du nom de l'utilisateur, puisque celui-ci ne respecte pas le format souhaité par les barèmes de contrôle. Le test doit s'assurer que la requête malveillante ne se rende pas à la base de données. Le test doit se conclure par un succès.

Prendre note qu'en cas de requête légitime, seules les informations attachées à l'utilisateur, passées en paramètre, doivent normalement être retournées par la base de données.

CLASSE *ERROR-BASED*

L'injection SQL nommée *Error-based* tente d'exploiter des faiblesses pouvant retourner des messages d'erreur au lieu de résultats (Galluccio *et al.*, 2020).

Scénario 2.1

Nom : Tenter de briser le logiciel, en affichant la liste de chansons.

Description : Un utilisateur peut effectuer une requête sur un module de recherche, lequel permet de récupérer une liste d'albums de musique présents dans la base de données. En aucun cas, cette requête ne doit être valide. Le test doit être effectué sur une portion du code source qui prend en paramètre des mots-clés. Le test doit créer un bris au niveau du logiciel et générer une exception.

Résultat attendu :

- Code vulnérable : Absence de contrôle de sécurité. Un attaquant peut être en mesure de forger une requête qui lui permet de tester le champ en injectant diverses requêtes

illégitimes, lesquelles pourraient conduire à un bris de logiciel. Le test doit se conclure par un échec, puisque selon les intrants et les attentes, une requête de la sorte doit soulever une exception depuis la base de données. Cependant, une exception non gérée peut retourner des informations techniques sur la base de données.

- Code sécurisé : L'ensemble des contrôles de sécurité ont été mis en place au niveau du code source. Un attaquant qui tente de forger une requête malveillante verra celle-ci être interrompue par une exception adéquatement gérée par le système. Cette requête malveillante sera non conforme au niveau des contrôles de sécurité et sera interrompue en soulevant une exception, laquelle est gérée adéquatement par le logiciel. À terme, aucune information technique n'est divulguée.

CLASSE *UNION-BASED*

Scénario 3.1

Nom : Afficher la liste des artistes et des employés.

Description : Un utilisateur peut effectuer une requête sur un module de recherche, lequel permet de retourner une liste d'artistes. Le test unitaire prend en paramètre des mots-clés et doit s'assurer qu'il n'est pas possible de créer une attaque à partir d'une syntaxe qui permet d'insérer une requête de type *Union*. Une syntaxe de la sorte permet d'ajouter une seconde requête et y récupérer les résultats. Dans ce cas-ci, en interrogeant la table des artistes, l'attaquant tente de récupérer également la liste des employés.

Résultat attendu :

- Code vulnérable : Absence de contrôle de sécurité. Un attaquant peut être en mesure de forger une requête qui lui permet de récupérer non seulement la liste des artistes relatifs aux mots-clés, mais également des informations provenant de la table des employés. Le

test doit se conclure par un échec, puisque selon les intrants et les attentes, une requête de la sorte est en mesure de se rendre au niveau de la base de données.

- Code sécurisé : L'ensemble des contrôles de sécurité ont été mis en place au niveau du code source. Un attaquant qui tente de forger une requête malveillante lui permettant de récupérer des informations provenant d'une autre table verra celle-ci être interrompue au niveau de la détection d'un *pattern* d'injection SQL. Le test doit s'assurer que la requête malveillante ne se rende pas à la base de données.

Prendre note qu'en cas de requête légitime, seule une liste des artistes relatifs aux mots-clés soumis doit être retournée par la base de données.

Scénario 3.2

Nom : Afficher une liste de chanson et la structure d'une table.

Description : Un utilisateur peut effectuer une requête sur un module de recherche, lequel permet de retourner une liste de chansons. Le test unitaire prend en paramètre des mots-clés et doit s'assurer qu'il n'est pas possible de créer une attaque par injection SQL de type *Union*. L'attaquant peut tenter de récupérer de tester la validité d'une table ou du nombre de colonnes d'une autre table. Cette approche peut fournir des données utiles à un attaquant cherchant à approfondir sa compréhension de la base de données visée.

Résultat attendu :

- Code vulnérable : Absence de contrôle de sécurité. Un attaquant peut être en mesure de forger une requête qui lui permet de récupérer non seulement la liste des chansons relatives aux mots-clés, mais également des informations techniques provenant d'autres tables. Le test doit se conclure par un échec, puisque selon les intrants et les attentes, une requête de la sorte est en mesure de se rendre au niveau de la base de données.
- Code sécurisé : L'ensemble des contrôles de sécurité ont été mis en place au niveau du code source. Un attaquant qui tente de forger une requête malveillante qui lui permet de récupérer des informations provenant d'autres tables verra celle-ci être interrompue au niveau d'une non-conformité des paramètres ou de la détection d'un *pattern* d'injection SQL. Le test doit s'assurer que la requête malveillante ne se rende pas à la base de données.

Scénario 3.3

Nom : Afficher une *Playlists* et des informations provenant des journaux du logiciel.

Description : Un utilisateur peut effectuer une requête sur un module de recherche, lequel permet de retourner une liste de *Playlists*. Le test unitaire prend en paramètre des mots-clés et doit s'assurer qu'il n'est pas possible de créer une attaque par injection SQL de type *Union*. L'attaquant peut tenter de récupérer les activités du logiciel depuis la table des journaux (*logs*). Cette table peut fournir des informations pertinentes pour un attaquant, lequel pourrait les utiliser par la suite, pour effectuer d'autres types d'attaques.

Résultat attendu :

- Code vulnérable : Absence de contrôle de sécurité. Un attaquant peut être en mesure de forger une requête qui lui permet de récupérer non seulement une liste de *Playlists* relatives aux mots-clés, mais également des informations provenant de la table des journaux. Le test doit se conclure par un échec, puisque selon les intrants et les attentes, une requête de la sorte est en mesure de se rendre au niveau de la base de données.
- Code sécurisé : L'ensemble des contrôles de sécurité ont été mis en place au niveau du code source. Un attaquant qui tente de forger une requête malveillante qui lui permet de récupérer des informations provenant de la table des journaux verra celle-ci être interrompue au niveau d'une non-conformité des paramètres ou de la détection d'un *pattern* d'injection SQL. Le test doit s'assurer que la requête malveillante ne se rende pas à la base de données. À noter que dans ce cas précis, et en supposant que la requête se rende à la base de données, celle-ci serait rejetée par la base de données puisque les requêtes effectuées auprès de cette table demandent un autre type d'accès utilisateur que celui utilisé pour interroger la table des chansons. Le test doit s'assurer que la requête malveillante ne se rende pas à la base de données.

CLASSE *PIGGY BACKING*

Prendre note que les attaques de type *Piggy Backing* composées de requêtes multiples (*Stacked*), envoyées à MySQL par un connecteur *JDBC* ne sont pas fonctionnelles [33]. Les requêtes multiples génèrent des exceptions SQL.

Scénario 4.1

Nom : Insertion dans les journaux, à partir de la création d'un utilisateur dans l'application.

Description : Un utilisateur peut effectuer une requête sur un module de recherche, lequel permet de retourner une liste d'artistes. Le test unitaire prend en paramètre des mots-clés et doit s'assurer qu'il n'est pas possible d'insérer une requête malveillante complète à la suite d'une requête SQL légitime. Dans quel cas, il sera possible d'exécuter une requête d'insertion de données, de mise à jour ou encore de suppression de données. Dans ce cas-ci, une requête d'ajout est utilisée pour insérer une information dans la table des journaux (*logs*). À terme, si des informations malveillantes sont insérées dans la table des journaux, une seconde attaque pourrait en découler, comme une attaque de type XSS.

Résultat attendu :

- Code vulnérable : Absence de contrôle de sécurité. Un attaquant peut être en mesure de forger une requête qui lui permet d'insérer des informations malveillantes. Le test doit se conclure par un échec, puisque selon les intrants et les attentes, une requête de la sorte est en mesure de se rendre au niveau de la base de données et être exécutée sans restriction.

33. PortSwigger SQL injection cheat sheet. Récupéré en janvier 2024 de https://portswigger.net/web-security/sql-injection/cheat-sheet

- Code sécurisé : L'ensemble des contrôles de sécurité ont été mis en place au niveau du code source. Un attaquant qui tente de forger une requête malveillante qui lui permet d'insérer des informations dans la table des journaux verra celle-ci être interrompue au niveau de la détection de la non-conformité des barèmes ou d'un *pattern* d'injection SQL. Le test doit s'assurer que la requête malveillante ne se rende pas à la base de données.

Il convient de noter que dans cette situation, et en supposant que la requête parvienne à la base de données, elle serait refusée car des mesures de sécurité ont été implémentées. Les requêtes adressées à la table *logs* exigent un niveau d'accès utilisateur spécifique.

Scénario 4.2

Nom : Création d'un nouvel utilisateur applicatif, à partir de la mise à jour d'une adresse courriel.

Description : Un utilisateur peut effectuer une requête de mise à jour de son adresse courriel en tant que client. Le test unitaire, qui prend en paramètre l'ancienne et la nouvelle adresse courriel doit s'assurer qu'il n'est pas possible d'insérer une requête malveillante complète à la suite de la requête légitime. Dans quel cas, il serait possible d'exécuter une requête d'insertion de données, de mise à jour ou encore de suppression de données. Dans ce cas-ci, une requête malveillante sera utilisée pour insérer un nouvel utilisateur dans la table des utilisateurs.

Résultat attendu :

- Code vulnérable : Absence de contrôle de sécurité. Un attaquant peut être en mesure de forger la requête qui lui permet d'insérer des informations dans une table. Le test doit se conclure par un échec, puisque selon les intrants et les attentes, une requête de la sorte est en mesure de se rendre au niveau de la base de données.
- Code sécurisé : L'ensemble des contrôles de sécurité ont été mis en place au niveau du code source. Un attaquant qui tente de forger une requête malveillante qui lui permet

d'insérer un nouvel utilisateur dans la table verra celle-ci être interrompue au niveau de la détection d'un *pattern* d'injection SQL. Le test doit s'assurer que la requête malveillante ne se rende pas à la base de données.

Scénario 4.3

Nom : Destruction d'une table, à partir de la mise à jour d'une adresse courriel.

Description : Un utilisateur peut effectuer une requête de mise à jour de son adresse courriel en tant que client. Le test unitaire qui prend en paramètre l'ancienne adresse courriel et une nouvelle adresse courriel doit s'assurer qu'il n'est pas possible d'insérer une requête malveillante complète, à la suite de la requête légitime. Dans quel cas, il serait possible d'exécuter une requête de destruction d'une table.

Résultat attendu :

- Code vulnérable : Absence de contrôle de sécurité. Un attaquant peut être en mesure de forger une requête qui lui permet d'altérer la base de données. Le test unitaire qui prend en paramètre l'ancienne et la nouvelle adresse courriel, doit s'assurer qu'il n'est pas possible d'insérer une requête malveillante complète, à la suite de la requête légitime. Le test doit se conclure par un échec, puisque selon les intrants et les attentes, une requête de la sorte est en mesure de se rendre au niveau de la base de données.
- Code sécurisé : L'ensemble des contrôles de sécurité ont été mis en place au niveau du code source. Un attaquant qui tente de forger une requête malveillante qui lui permet d'altérer la base de données verra celle-ci être interrompue au niveau de la non-conformité des barèmes ou la détection d'un *pattern* d'injection SQL. Le test doit s'assurer que la requête malveillante ne se rende pas à la base de données.

À noter que dans ce cas précis, et en supposant que la requête se rende à la base de données, celle-ci serait rejetée par la base de données puisque les requêtes effectuées auprès de la base de données demandent un haut niveau d'accès.

Scénario 4.4

Nom : Exécuter la fonction *Sleep()*, dans une requête qui récupère les informations d'un utilisateur.

Description : Un utilisateur peut effectuer une requête qui retourne les informations d'un utilisateur. Le test unitaire qui prend en paramètre une adresse courriel doit s'assurer qu'il n'est pas possible d'insérer une requête malveillante complète à la suite de la requête légitime. Dans quel cas, il serait possible pour un attaquant de vérifier si une vulnérabilité aux injections SQL est exploitable en exécutant une fonction SQL. Il pourrait également être possible de créer une attaque de type *DoS* en monopolisant l'ensemble des connexions à la base de données.

Résultat attendu :

- Code vulnérable : Absence de contrôle de sécurité. Un attaquant est en mesure de forger une requête qui lui permet d'exécuter la fonction *Sleep()* auprès de la base de données. Le test qui prend en paramètre une adresse courriel doit se conclure par un échec, puisque selon les intrants et les attentes, une requête de la sorte est en mesure de se rendre au niveau de la base de données. À noter qu'une injection SQL qui tente d'injecter une requête *SELECT SLEEP(1000);* est en mesure de monopoliser une connexion durant 1000 secondes.
- Code sécurisé : L'ensemble des contrôles de sécurité ont été mis en place au niveau du code source. Un attaquant qui tente de forger une requête malveillante qui lui permet d'insérer cette fonction verra celle-ci être interrompue au niveau de la détection des commandes non autorisées. Le test doit s'assurer que la requête malveillante ne se rend pas à la base de données.

3.3.3 TESTS UNITAIRES - TRANSVERSAUX AU SYSTÈME

Les tests unitaires transversaux du système valident la configuration de certains paramètres de sécurité établis selon une bonne pratique de sécurité. Ces derniers ne sont pas attachés aux contrôles de sécurité dans le code source. Ils visent principalement à tester les configurations de sécurité attribuées.

PROCÉDURE STOCKÉE

Scénario 5.1

Nom : Procédures stockées.

Description : Le test unitaire doit s'assurer que les procédures stockées sont inexistantes (ou inaccessibles). Deux types de vérifications sont possibles. Une première vérification qui confirme que celles-ci sont inexistantes au niveau de la base de données *Chinook*. Une seconde qui s'assure que seul un compte de service à haut privilège est autorisé à les exécuter. À noter que pour les besoins de ce test, nous vérifierons l'accessibilité d'une seule procédure stockée.

Résultat attendu :

- Au moment de l'appel du test unitaire, la procédure stockée ne doit pas être accessible dans la base de données *Chinook*. Le test doit retourner une exception si la procédure stockée est inaccessible, sinon il doit retourner une valeur de type *Integer* qui confirme son existence. Il est important de noter que dans ce cas particulier, le test doit générer une exception pour être considéré comme réussi.

SÉPARATION DES TÂCHES (*SEPARATION OF DUTIES*)

Scénarios 6.1 à 6.6

Nom : Les comptes de service

Description : Il s'agit d'exécuter plusieurs tests unitaires qui vérifient les autorisations de chacun des comptes de service existants relatifs à la base de données. Par exemple, un compte de services qui a des droits standards d'accès ne doit pas avoir les droits pour accéder à la table des journaux.

Résultat attendu :

- Différents scénarios de tests seront effectués afin de valider les accès des comptes de service. Par exemple, la table de journaux (*Logs*) est uniquement accessible par le compte *log*. La table contenant les utilisateurs applicatifs est accessible par les comptes *auth* (authentification) et *std* (standard). Toutes les autres tables sont uniquement accessibles par le compte *std* (standard). Quant au compte *root*, il a accès à l'ensemble des tables, fonctions et commandes de la base de données *Chinook*.

MOINDRE PRIVILÈGE

Scénario 7.1

Nom : La suppression de table est réservée uniquement au compte à haut privilège

Description : Exécuter un test unitaire qui vérifie que seul le compte *root* a les droits de suppression des tables. Tous les autres comptes de services attitrés à *Chinook* n'ont aucun droit à ce niveau.

Résultat attendu :

- Différents intrants de tests seront appliqués afin de valider les accès des comptes. Sous la base de données *Chinook*, l'exécution de la requête de suppression d'une table par les comptes *log*, *std* et *auth*, doit générer une exception SQL liée à un refus d'exécution.

Scénario 7.2

Nom : Afficher la liste des tables accessibles pour chacun des comptes

Description : Exécuter un test unitaire qui valide le nombre de tables accessibles pour chacun des comptes de service. Il s'agit ici d'une forme simple de validation pour les besoins de la démonstration. Un test plus adéquat pourrait valider chacun des noms des tables attachées à chacun des comptes de service.

Résultat attendu :

- Différents intrants de tests seront appliqués afin de vérifier les accès des comptes de service de la base de données *Chinook*. L'exécution de la requête *Show Tables* doit retourner une (1) table pour *auth*, une (1) table pour *log*, douze (12) tables pour *std* et finalement, treize (13) tables pour *root*. Toutes autres valeurs que celles mentionnées démontrent qu'une mauvaise configuration est présente au niveau des droits d'accès aux tables.

Ce chapitre expose les scénarios de tests unitaires que nous devrons mettre en pratique pour garantir la conformité aux normes et aux exigences de qualité établies. Chaque scénario aborde un aspect spécifique, mettant en lumière l'importance des tests unitaires dans la prévention des injections SQL et la sécurisation des systèmes. Dans le prochain chapitre, nous approfondirons ces scénarios en analysant en détail les résultats obtenus à partir de tests unitaires effectués sur du code source sécurisé par rapport à du code source non sécurisé. De plus, nous examinerons les résultats des scénarios de tests effectués sur certains paramètres de sécurité liés aux systèmes de gestion de base de données.

CHAPITRE IV
RÉSULTATS

4.1 INTRODUCTION

Dans ce chapitre, nous présentons en détail les résultats de notre recherche, mettant en pratique les stratégies que nous proposons dans le *Chapitre 3*. Nous discutons également des implications de nos résultats sur la prévention des injections SQL.

La prévention des attaques par injections SQL à travers l'implémentation de tests unitaires contribue à minimiser les risques de compromission des données, tout en transférant la responsabilité de cette prévention aux équipes de développement de logiciels. En agissant sur le code source et le traitement des intrants, la plupart des attaques par injections SQL peuvent être contrecarrées, en garantissant que les entrées et les commandes sont structurées et formatées selon les pratiques de développement sécuritaire (Galluccio *et al.*, 2020).

En combinant les principes de tests unitaires avec les mesures de protection contre les menaces par injections SQL, notre objectif est de fusionner le concept de test unitaire à la prévention des injections SQL, comme mesure de protection. Ainsi donc, en établissant des scénarios de tests unitaires qui ciblent les vulnérabilités sensibles aux injections SQL, nous sommes d'avis qu'il est possible de rehausser le niveau de sécurité du système d'information qui bénéficie de ces tests.

Les contrôles de sécurité que nous souhaitons mettre en place sont bien connus dans la pratique de développement sécuritaire. Notre approche par tests unitaires reprend simplement chacun des contrôles de sécurité et évalue systématiquement l'efficacité du code source à partir des scénarios de tests.

Pour détailler notre approche, exposer les résultats de notre projet de recherche et enfin répondre à nos hypothèses, ce chapitre est divisé en cinq sections :

1. Exposer le jeu de données que nous avons utilisées pour mener à bien notre projet de recherche. Nous allons non seulement identifier leur source, mais également mettre en lumière leur pertinence par rapport à nos objectifs.
2. Détailler notre processus menant aux résultats, en examinant chacune des étapes de contrôle proposées tout en fournissant des exemples concrets.
3. Exposer les résultats obtenus, lesquels permettent de comprendre les avantages et les inconvénients que pourrait avoir cette démarche. Les tests unitaires étant naturellement utilisés pour des besoins fonctionnels, une approche orientée sur la sécurité demande une vision quelque peu différente.
4. Conclure en effectuant un retour sur la démarche et les résultats obtenus, tout en offrant des perspectives d'approches pour des projets de recherche. Le tout basé sur les résultats obtenus et les lacunes observées.

4.2 JEU DE DONNÉES

Tel que nous l'avons mentionné au **Chapitre 3**, nous avons dû identifier un jeu de données sous forme de base de données relationnelle. Ce dernier devant satisfaire la réalisation de nos scénarios de tests. Ce jeu de données, nommé *Chinook*, offre non seulement un ensemble de données qui répond à la mission de ce projet, mais qui est également disponibles sous plusieurs formes de système de gestion de base de données, nous permettant ainsi de tester notre approche sous différents langages SQL. Dans le cadre de ce projet, nous avons effectué nos tests unitaires principalement sur *MySQL* et contre-validé sur *Microsoft SQL Server*. La Figure 4.1 démontre de façon détaillée la structure de la base de données utilisée pour la création et l'exécution des tests.

Deux tables ont été ajoutées pour répondre à nos besoins. Nous avons ajouté une première table contenant la liste des utilisateurs (*UserAccount*) tirée de la table des clients (*Customer*). La table *UserAccount* est composée de cinq champs, soit un champ contenant l'identifiant de l'utilisateur, un champ contenant le nom de l'utilisateur sous forme d'adresse courriel, un champ contenant le mot de passe et finalement un champ de commentaire. Le cinquième champ est la clé d'appariement avec la table *Customer*. Nous avons ajouté une seconde table afin de simuler la notion de journaux pouvant contenir les événements et activités reliés au logiciel. La table *Logs* est composée d'un champ contenant l'identifiant du journal, un champ contenant l'activité journalisée et un champ *TimeStamp* contenant la date et l'heure de l'activité journalisée.

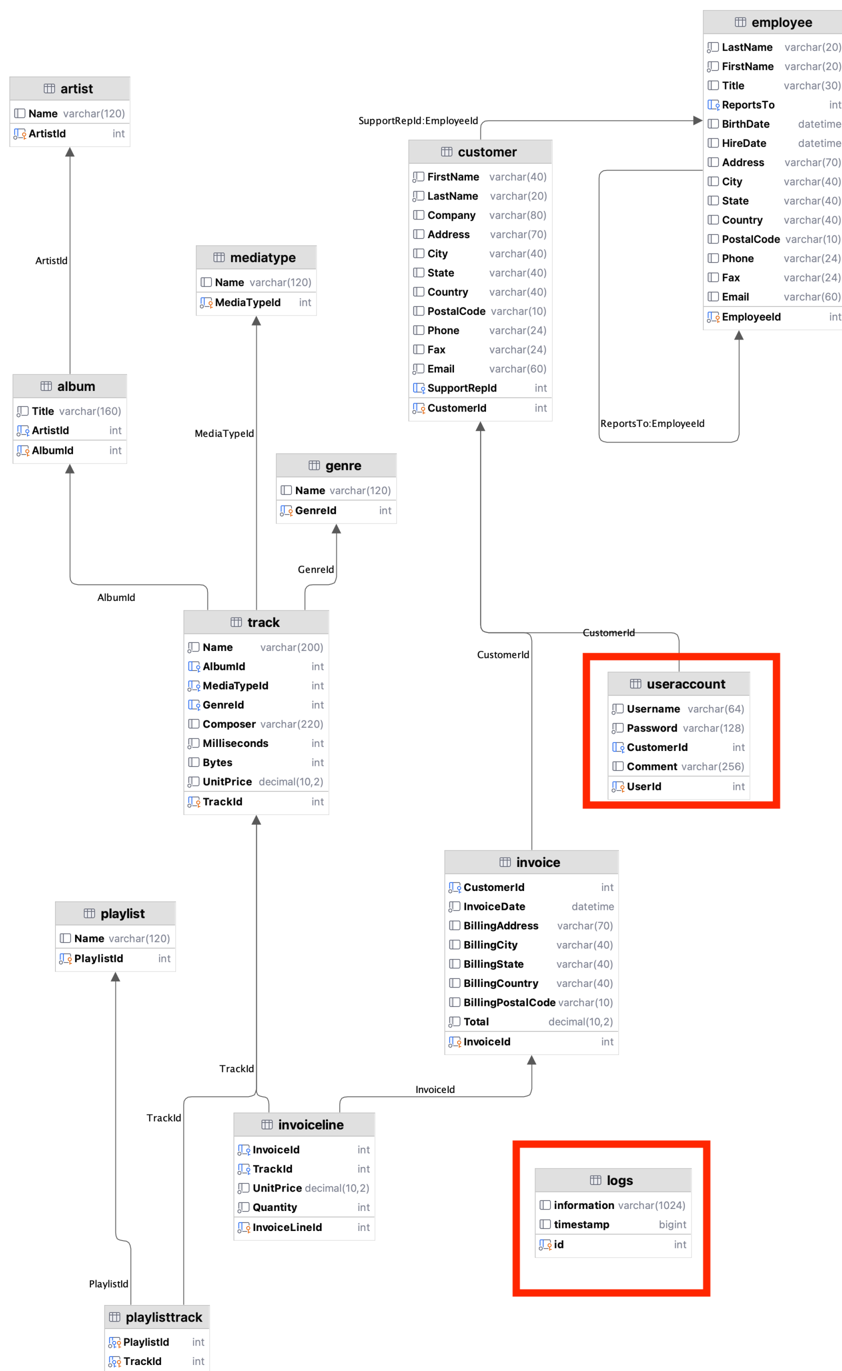

FIGURE 4.1 : Schéma de la base de données *Chinook* sous MySQL et MSSQL, incluant les tables *UserAccount* et *Logs*.

De plus, nous avons créé et attribué différents rôles et accès aux comptes de services de la base de données *Chinook*. L'attribution des droits, pour chacun des comptes de services, est détaillée dans la Figure 4.2 :

1. **log** : Compte de service qui a uniquement accès à la table *Log*.
2. **auth** : Compte de service qui a uniquement accès à la table *UserAccount*.
3. **std** : Compte de service qui a accès à toutes les tables, sauf la table *Log*.
4. **root** : Compte de service par défaut, il a accès à toutes les tables et les commandes de la base de données.

```
-- Rights for 'log' user account
GRANT SELECT, INSERT ON Chinook.logs TO 'log'@'%';
-- Rights for 'auth' user account
GRANT SELECT ON Chinook.useraccount TO 'auth'@'%';
-- Rights for 'std' user account
GRANT SELECT, INSERT, UPDATE, DELETE ON Chinook.album TO 'std'@'%';
GRANT SELECT, INSERT, UPDATE, DELETE ON Chinook.artist TO 'std'@'%';
GRANT SELECT, INSERT, UPDATE ON Chinook.customer TO 'std'@'%';
GRANT SELECT, INSERT, UPDATE, DELETE ON Chinook.employee TO 'std'@'%';
GRANT SELECT, INSERT, UPDATE, DELETE ON Chinook.genre TO 'std'@'%';
GRANT SELECT, INSERT, UPDATE, DELETE ON Chinook.invoice TO 'std'@'%';
GRANT SELECT, INSERT, UPDATE, DELETE ON Chinook.invoiceline TO 'std'@'%';
GRANT SELECT, INSERT, UPDATE, DELETE ON Chinook.mediatype TO 'std'@'%';
GRANT SELECT, INSERT, UPDATE, DELETE ON Chinook.playlist TO 'std'@'%';
GRANT SELECT, INSERT, UPDATE, DELETE ON Chinook.playlisttrack TO 'std'@'%';
GRANT SELECT, INSERT, UPDATE, DELETE ON Chinook.track TO 'std'@'%';
GRANT SELECT, INSERT, UPDATE ON Chinook.useraccount TO 'std'@'%';
```

FIGURE 4.2 : Règles d'accès attribuées aux comptes de services.

Depuis ce jeu de données, nous avons créé des intrants typiques représentant des entrées provenant d'utilisateurs, lesquels seront testés sur les bases de données. Ainsi pour chacun des scénarios de tests proposés, des intrants pourront prendre différentes formes, telles que :

1. Des intrants valides et non malveillants
2. Des intrants non valides et non malveillants
3. Des intrants valides et malveillants
4. Des intrants non valides et malveillants

Nous entendons par *intrants valides* des entrées utilisateurs qui respectent les barèmes et paramètres souhaités dans le cadre de notre logiciel fictif. Il s'agit notamment du format, de la longueur et du type de valeur d'une entrée. Dans le Tableau 4.1, nous retrouvons une suite d'intrants destinés à tester la validité d'une requête qui retourne le nombre d'achats d'un client, depuis l'adresse courriel de ce dernier. Chacun de ces intrants contient plusieurs variables dont l'identifiant séquentiel du test, l'adresse courriel à contrôler, le résultat attendu de la requête SQL qui retourne le nombre d'achats effectués et finalement, le statut attendu du test.

TestID	Adresse courriel	Résultat	Statut
1	h@g.com	0	FAIL
2	hholy.gmail.com	0	FAIL
3	astrid_.gruber@apple.at	0	PASS
4	astrid.gruber@apple.at	7	PASS
5	hholy@gmail.com	7	PASS
6	' OR true=true ; –	0	FAIL
7	hholy@gmail.com' OR (('2'='2')) ; –	0	FAIL

TABLEAU 4.1 : Exemple d'intrants composés de paramètres.

Si un test dont le statut attendu est *PASS* et que le résultat du test est *PASS*, alors le test est un succès *PASS (1)*. Si un test dont le statut attendu est *FAIL* et que le résultat du test est *FAIL*, alors le test est un succès *PASS (1)*. En contre-parti, si un test dont le statut attendu est *PASS* et que le résultat du test est *FAIL* alors le test est un échec *FAIL (0)*. Et finalement, si un test dont le statut attendu est *FAIL* et que le résultat du test est *PASS*, alors le test est un échec *FAIL (0)*. Le Tableau 4.2 contient les résultats des tests en fonction des attentes et des résultats :

	PASS (A)	**FAIL (A)**
PASS (R)	1	0
FAIL (R)	0	1

TABLEAU 4.2 : Matrice des attentes (A) et des résultats (R).

Le détail des différents paramètres de chacun des intrants du Tableau 4.1 est énoncé dans la liste suivante :

1. Un intrant contenant une adresse courriel qui ne respecte pas la longueur minimale.
2. Un intrant contenant une adresse courriel dont le format ne respecte pas les barèmes.
3. Un intrant contenant une adresse courriel dont le format respecte les barèmes, pour laquelle il n'existe aucun achat.
4. Un intrant contenant une adresse courriel dont le format respecte les barèmes, pour laquelle il existe sept (7) achats.
5. Un intrant contenant une adresse courriel dont le format respecte les barèmes, pour laquelle il existe sept (7) achats.
6. Un intrant ne contenant pas d'adresse courriel, mais qui contient une injection SQL.
7. Un intrant contenant une adresse courriel dont le format respecte les barèmes et contient une injection SQL.

Dans un code source sécurisé, les intrants invalides ou malveillants ne doivent pas atteindre la base de données. Prenons comme exemple le quatrième intrant de la liste ci-dessus, qui contient une adresse courriel valide selon son format et sa longueur. Le résultat attendu est la valeur 7, c'est-à-dire qu'il existe 7 achats pour cet utilisateur et le statut attendu du test est *PASS* (Succès). Prenons le dernier paramètre qui tente d'effectuer une injection SQL, dont la valeur de l'adresse courriel respecte le format, mais un *pattern* d'injection de code est présent. Le retour attendu est 0 et le statut du test est *FAIL* (Échec). En résumé, si l'élément testé satisfait aux critères de sécurité définis, que l'adresse courriel du client soit présente ou non dans la table des achats, le résultat attendu du test est *PASS*. Cependant, si l'entrée testée ne respecte pas les critères de sécurité et semble contenir une injection SQL, cette entrée déclenchera une exception et le résultat attendu sera *FAIL*.

Finalement, si nous effectuons un test avec le sixième intrant et que ce dernier retourne un résultat supérieur à zéro, cela signifiera que l'injection SQL a été exécutée avec succès et que le statut attendu qui devait être *FAIL* est désormais *PASS*. Nous avons donc un échec au niveau du test. Des corrections au niveau du code source s'imposent, puisqu'une injection SQL est passée au travers des contrôles de sécurité.

SIMULATION LOGICIELLE

Afin de simuler une séquence de tests relatifs à la sécurité, nous avons décomposé notre application sous une architecture *n-tier* limitée à deux couches, soit la couche métier (*Business*) qui contient les règles de validation et les règles d'affaires, et la couche d'accès aux données (*Data*) qui fait appel aux bases de données, en fonction du système de gestion de base de données souhaitées au moment de l'exécution des tests unitaires. Ainsi, les tests unitaires sont effectués depuis la couche métier, qui doit effectuer tous les contrôles nécessaires avant

de passer la requête à la couche inférieure, soit la couche d'accès aux données (Figure 4.3). À noter qu'aucun test n'est effectué directement sur la couche d'accès aux données.

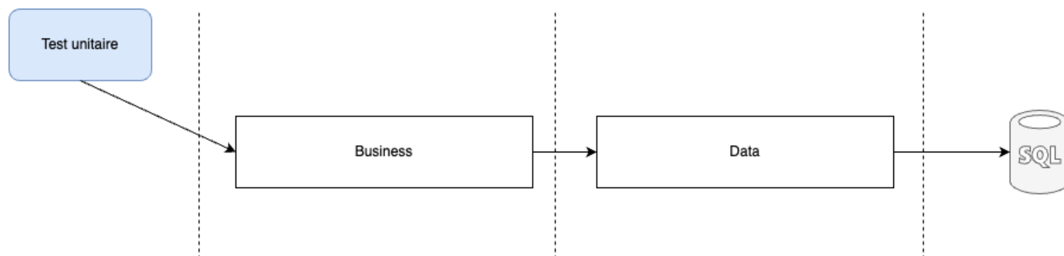

FIGURE 4.3 : Test unitaire à partir de la couche métier.

Il convient de mentionner que nous n'avons pas estimé nécessaire de créer une couche d'interface graphique (*UI*) dans le cadre de ce projet, cependant nous avons créé une suite d'API Web (*Application Programming Interface*) qui font appel aux diverses fonctions présentes sous la couche métier dans une section que nous détaillons plus loin dans ce chapitre (Figure 4.4).

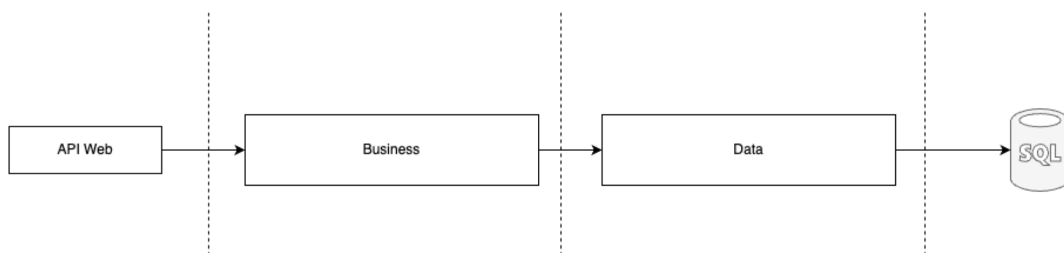

FIGURE 4.4 : Architecture de la simulation du logiciel.

4.3 APPROFONDISSEMENT DE LA MÉTHODE

Comme indiqué dans le **Chapitre 3**, notre méthode repose sur deux paradigmes de tests : le premier vise à évaluer la qualité des contrôles de sécurité au niveau du code source. Le second s'assure que les configurations de sécurité de la base de données respectent les attentes.

Basés sur les scénarios, nous avons programmé deux séries de tests qui font appel à l'une ou l'autre des bases de données. Une première série de tests qui effectuent des essais sur du code source sécurisé (aussi appelé *tests sécurisés*) et une seconde série de tests qui effectuent des essais sur du code source non sécurisé (aussi appelé des *tests non sécurisés*). Les deux séries de tests font usage des mêmes suites d'intrants, incluant les mêmes valeurs de retour et les mêmes attentes. Les deux séries de tests peuvent être exécutées indépendamment ou simultanément. Lors de l'exécution de la série de tests sur le code source sécurisé, tous les tests réalisés se soldent avec succès. Par contre, les tests effectués sur du code source non sécurisé peuvent générer des échecs.

Dans un second temps, des tests transversaux effectués sur le système de gestion de la base de données permettent d'assurer une configuration adéquate des mesures de sécurité, telles que la séparation des tâches, la gestion des accès aux procédures stockées et la mise en œuvre du principe du privilège minimal pour les comptes de service. À titre d'exemple, il est essentiel de veiller à ce qu'un compte de service lié à de la base de données ne dispose pas des autorisations nécessaires pour supprimer une table en exécutant la requête *DROP TABLE UserAccount*. Si par inadvertance cette injection SQL parvient à passer au travers des contrôles de sécurité dans le code source, le compte de service ne disposant pas des autorisations nécessaires pour exécuter la requête, se verra refuser l'exécution de la requête SQL en raison des restrictions d'autorisation.

4.4 RÉSULTATS

Nous allons examiner les résultats des différents scénarios et leurs paramètres, ainsi que chaque variante du code source, sécurisée ou non. En d'autres termes, nous allons passer en revue les performances de chaque combinaison de scénarios, paramètres et versions de code source pour en comprendre les résultats et ainsi démontrer l'efficacité des tests unitaires comme moyen de prévention. De plus, nous allons également mettre en parallèle les résultats de chaque version de systèmes de gestion de base de données que nous avons utilisés, c'est-à-dire *MySQL* et *SQL Server (MSSQL)*. Dans un premier temps, nous examinons les résultats des tests unitaires qui confirment les mesures de sécurité mises en œuvre, et par la suite les résultats des tests unitaires qui confirment que les systèmes de gestion de bases de données sont adéquatement sécurisés.

RÉSULTATS - CONTRÔLES DE SÉCURITÉ

Avant de débuter, il est essentiel de saisir la différence entre le concept de tests unitaires qui valide la fonctionnalité d'une fonction, et celui des tests unitaires qui valide la sécurité d'une fonction. Un test unitaire qui valide la fonctionnalité s'assure que la fonction répond à la mission pour laquelle elle a été créée. Par exemple, si une méthode d'authentification valide le nom d'un utilisateur et son mot de passe, un test qui se solde par un succès (*PASS*) signifie que le nom de l'utilisateur est présent dans la base de données et que le mot de passe correspond à ce dernier. Ainsi, la valeur de retour de la fonction sera *TRUE*. Cependant, si un test se solde par un échec (*FAIL*), cela signifie que le nom de l'utilisateur n'est pas présent dans la base de données ou que le mot de passe ne correspond pas avec ce dernier. Alors, la valeur de retour de la méthode est *FALSE*.

Dans un test unitaire de sécurité, qui vise à garantir l'absence de vulnérabilités d'une portion de code, la présence ou l'absence du nom d'utilisateur dans la base de données et l'adéquation ou non du mot de passe n'ont aucune incidence. Si la valeur de retour du test est *TRUE* ou *FALSE*, cela indique que le test a réussi (*PASS*), puisque la méthode a répondu aux critères d'un test visant à garantir la mise en œuvre des contrôles de sécurité. Cependant, si un test ne répond pas aux attentes de sécurité (*FAIL*), cela signifie que la valeur de retour est ni *TRUE* ni *FALSE*.

Voici une synthèse des résultats des tests pour les scénarios 1 à 4, concernant les portions de code sécurisé et non sécurisé (Figure 4.5). À gauche, les résultats réussis des tests effectués sur le code sécurisé et à droite, les résultats infructueux des tests effectués sur le code non sécurisé.

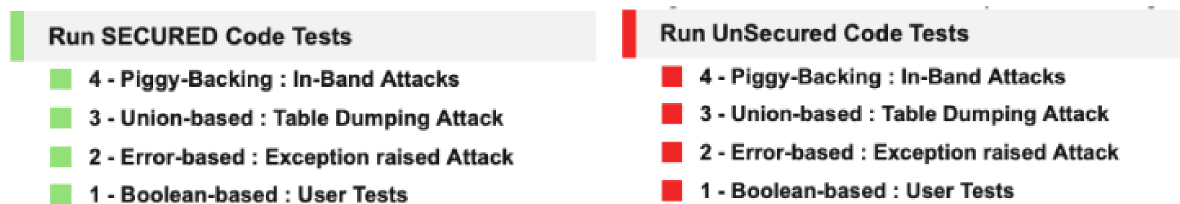

FIGURE 4.5 : Sommaire des résultats des tests sécurisés et non sécurisés pour la suite de scénarios 1 à 4

Scénario 1.1

Rappel du scénario

Un utilisateur invité tente de se connecter à la boutique. Le test unitaire doit vérifier qu'il n'est pas possible qu'un utilisateur puisse se connecter en contournant les contrôles de sécurité au moment de l'authentification.

Résultats

```
Code sécurisé

Scenario #1.1 - Security Test for User Login (Auth Bypass)
[1] [1], h@g.com, secret, false, FAIL
[2] [2], hholy@gmail.com' UNION SELECT 1; -- , secret, false, FAIL
[3] [3], hholy.gmail.com, secret, false, FAIL
[4] [4], hholy@gmail.com, notsecret, false, PASS
[5] [5], hholy@gmail.com, secret, true, PASS
[6] [6], ' OR 1=1; -- , secret_dumb, false, FAIL
[7] [7], hholy@gmail.com' OR 1=1; -- , secret_dumb, false, FAIL
[8] [8], hholyAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA@gmail.com, secret, false, FAIL
[9] [9], hholy@gmail.com, secret__________________, false, FAIL
[10] [10], hholy@gmail.com' -- , secret, false, FAIL
[11] [11], donot@existsemail.com' -- , secret, false, FAIL
[12] [12], hholy.gmail.com, s, false, FAIL
```

```
Code non sécurisé

Scenario #1.1 - Security Test for User Login (Auth Bypass)
[1] [1], h@g.com, secret, false, FAIL
[2] [2], hholy@gmail.com' UNION SELECT 1; -- , secret, false, FAIL
[3] [3], hholy.gmail.com, secret, false, FAIL
[4] [4], hholy@gmail.com, notsecret, false, PASS
[5] [5], hholy@gmail.com, secret, true, PASS
[6] [6], ' OR 1=1; -- , secret_dumb, false, FAIL
[7] [7], hholy@gmail.com' OR 1=1; -- , secret_dumb, false, FAIL
[8] [8], hholyAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA@gmail.com, secret, false, FAIL
[9] [9], hholy@gmail.com, secret__________________, false, FAIL
[10] [10], hholy@gmail.com' -- , secret, false, FAIL
[11] [11], donot@existsemail.com' -- , secret, false, FAIL
[12] [12], hholy.gmail.com, s, false, FAIL
```

FIGURE 4.6 : Résultats du test 1.1

Sous le code sécurisé, à gauche, le test est conforme aux attentes (Figure 4.6).

Sous le code non sécurisé, à droite, les intrants 4 et 5 sont conformes aux attentes du test. Les intrants 2, 6, 7, 10 et 11 contiennent des injections SQL et elles ont été exécutées sans obstruction sur la base de données. Bien que les attentes de ces derniers sont de valeur *FAIL*, nous avons eu une valeur *PASS* comme résultat. Les intrants 1, 3, 8, 9 et 12 ont également été exécutés sans obstruction sur la base de données. Bien que ces derniers ne sont pas conformes aux attentes, ils ne représentent pas une menace (Figure 4.6).

Scénario 1.2

Rappel du scénario

Un utilisateur peut effectuer une recherche dans sa liste d'achat de musique. Le test unitaire fait appel aux mécanismes qui retournent la liste complète des achats effectués par l'utilisateur. En aucun cas cette requête ne doit permettre de récupérer d'autres informations, par altération la requête légitime.

Résultats

Code sécurisé

Scenario #1.2 - Security Test for Getting the User Orders List

[1] [1], h@g.com, 0, FAIL
[2] [2], hholy.gmail.com, 0, FAIL
[3] [3], astrid_.gruber@apple.at, 0, PASS
[4] [4], astrid.gruber@apple.at, 7, PASS
[5] [5], hholy@gmail.com, 7, PASS
[6] [6], ' OR true=true; -- , 0, FAIL
[7] [7], hholy@gmail.com' OR (('2'='2')); -- , 0, FAIL

Code non sécurisé

Scenario #1.2 - Security Test for Getting the User Orders List

[1] [1], h@g.com, 0, FAIL
[2] [2], hholy.gmail.com, 0, FAIL
[3] [3], astrid_.gruber@apple.at, 0, PASS
[4] [4], astrid.gruber@apple.at, 7, PASS
[5] [5], hholy@gmail.com, 7, PASS
[6] [6], ' OR true=true; -- , 0, FAIL
[7] [7], hholy@gmail.com' OR (('2'='2')); -- , 0, FAIL

FIGURE 4.7 : Résultats du test 1.2

Sous le code sécurisé, à gauche, le test est conforme aux attentes. Il n'est pas possible de retourner d'autres informations que celles attachées à la requête légitime (Figure 4.7).

Sous le code non sécurisé, à droite, les intrants 3, 4 et 5 sont conformes aux attentes du test. Les intrants 6 et 7 contiennent des injections SQL et elles ont été exécutées sans obstruction sur la base de données. Bien que les attentes de ces derniers sont de valeur *FAIL*, nous avons eu une valeur *PASS* comme résultat. Les intrants 1 et 2 ont également été exécutés sans obstruction sur la base de données. Bien que ces derniers ne sont pas conformes aux attentes, ils ne représentent pas une menace (Figure 4.7).

Scénario 1.3

Rappel du scénario

Un utilisateur peut effectuer la recherche de son profil. Le test unitaire fait appel aux mécanismes de recherche qui retourne le profil d'un utilisateur à partir de son adresse courriel, sans récupérer d'autres informations.

Résultats

Code sécurisé	Code non sécurisé
Scenario #1.3 - Security Test to get The User profile	Scenario #1.3 - Security Test to get The User profile
[1] [1], h@g.com, false, FAIL	[1] [1], h@g.com, false, FAIL
[2] [2], hholy.gmail.com, false, FAIL	[2] [2], hholy.gmail.com, false, FAIL
[3] [3], astrid.gruber@apple.at, true, PASS	[3] [3], astrid.gruber@apple.at, true, PASS
[4] [4], hholy@gmail.com, true, PASS	[4] [4], hholy@gmail.com, true, PASS
[5] [5], hholy__@gmail.com, false, PASS	[5] [5], hholy__@gmail.com, false, PASS
[6] [6], ' OR true=true; -- , false, FAIL	[6] [6], ' OR true=true; -- , false, FAIL
[7] [7], hholy@gmail.com' OR (('2'='2')); -- , false, FAIL	[7] [7], hholy@gmail.com' OR (('2'='2')); -- , false, FAIL

FIGURE 4.8 : Résultats du test 1.3

Sous le code sécurisé, le test est conforme aux attentes (Figure 4.8).

Sous le code non sécurisé, les intrants 3, 4 et 5 sont conformes aux attentes du test. Les intrants 6 et 7 contiennent des injections SQL et elles ont été exécutées sans obstruction sur la base de données. Bien que les attentes sont de valeur *FAIL*, nous avons eu une valeur *PASS* comme résultat. Les intrants 1 et 2 ont également été exécutés sans obstruction sur la base de données. Bien que ces derniers ne sont pas conformes aux attentes, ils ne représentent pas une menace (Figure 4.8).

Scénario 2.1

Rappel du scénario

Un utilisateur peut effectuer une requête sur un module de recherche qui permet de récupérer une liste d'albums de musique présents dans la base de données. Le test devra être effectué sur la portion de code source qui prend des mots clés en paramètre et celui-ci doit impérativement briser le système, c'est-à-dire générer une exception.

Résultats

Code sécurisé

Scenario #2.1 - Security Test for Searching Songs and Make it Crashed!

[1] [1], Dog Eat Dog, true, PASS
[2] [2], eat, true, PASS
[3] [3], aaaaaaa, false, PASS
[4] [4], AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA, false, FAIL
[5] [5], A, false, FAIL
[6] [6], %'; DELETE FROM DUAL; !-, false, FAIL
[7] [7], ', false, FAIL

Code non sécurisé

Scenario #2.1 - Security Test for Searching Songs and Make it Crashed!

[1] [1], Dog Eat Dog, true, PASS
[2] [2], eat, true, PASS
[3] [3], aaaaaaa, false, PASS
[4] [4], AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA, false, FAIL
[5] [5], A, false, FAIL
[6] [6], %'; DELETE FROM DUAL; !-, false, FAIL
[7] [7], ', false, FAIL

FIGURE 4.9 : Résultats du test 2.1

Sous le code sécurisé, en haut, le test est conforme aux attentes (Figure 4.9).

Sous le code non sécurisé, en bas, les intrants 1, 2 et 3 sont conformes aux attentes du test. Les intrants 4 et 5 ont également été exécutés sans obstruction sur la base de données et bien que les attentes sont de valeur *FAIL*, nous avons eu des valeurs *PASS* comme résultat. Les intrants 6 et 7 contiennent des injections SQL menant à des erreurs et ils ont été exécutés sans obstruction sur la base de données (Figure 4.9).

À l’égard de l’intrant numéro 6, une exception est soulevée (Figure 4.10), laquelle confirme que la requête a été exécutée.

```
S2.1 Test[6] Exception raised : Incorrect syntax near ';'.
```

FIGURE 4.10 : Exception - Intrant 6

À l’égard de l’intrant numéro 7, une exception est soulevée également (Figure 4.11), laquelle confirme que la requête a été exécutée.

```
S2.1 Test[7] Exception raised : Unclosed quotation mark after the character string ')'.
```

FIGURE 4.11 : Exception - Intrant 7

Scénario 3.1

Rappel du scénario

Un utilisateur effectue une requête sur un module de recherche qui permet de retourner une liste d’artistes. Le test unitaire prend en paramètre des mots clés et doit s’assurer qu’il n’est pas possible de créer une attaque qui permet d’ajouter une seconde requête et y récupérer la liste des employés dans les résultats.

Résultats

Code sécurisé

Scenario #3.1 - Security Test for Searching artists

[1] [1], black, true, PASS
[2] [2], gilbert, true, PASS
[3] [3], bbbbbb, false, PASS
[4] [4], AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA, false, FAIL
[5] [5], A, false, FAIL
[6] [6], %') UNION SELECT 0, Email FROM employee; -- , false, FAIL

Code non sécurisé

Scenario #3.1 - Security Test for Searching Artists

[1] [1], black, true, PASS
[2] [2], gilbert, true, PASS
[3] [3], bbbbbb, false, PASS
[4] [4], AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA, false, FAIL
[5] [5], A, false, FAIL
[6] [6], %') UNION SELECT 0, Email FROM employee; -- , false, FAIL

FIGURE 4.12 : Résultats du test 3.1

Sous le code sécurisé, le test est conforme aux attentes (Figure 4.12).

Sous le code non sécurisé, les intrants 1, 2 et 3 sont conformes aux attentes du test. L’intrant 6 contient une injection SQL qui est exécutée sans obstruction sur la base de données. Bien que nos attentes soient de valeur *FAIL*, nous avons eu des résultats de valeur *PASS*. Les intrants 4 et 5 ont également été exécutés sans obstruction sur la base de données. Bien que ces derniers ne sont pas conformes aux attentes, ils ne représentent pas une menace (Figure 4.12).

Scénario 3.2

Rappel du scénario

Un utilisateur effectue une requête sur un module de recherche qui permet de retourner une liste de chansons. Le test unitaire prend en paramètre des mots clés et doit s'assurer qu'il n'est pas possible de créer une attaque qui tente de récupérer le nombre de colonnes de la table, depuis la requête légitime.

Résultats

Code sécurisé

Scenario #3.2 - Security Test for Searching Songs

[1] [1], black, true, PASS
[2] [2], Grunge, false, PASS
[3] [3], bbbbbbbb, false, PASS
[4] [4], AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA, false, FAIL
[5] [5], A, false, FAIL
[6] [6], %') UNION ALL SELECT 1,2,3,4,5,6; -- , false, FAIL
[7] [7], %') UNION ALL SELECT 1,2,3,4; -- , false, FAIL

Code non sécurisé

Scenario #3.2 - Security Test for Searching Songs

[1] [1], black, true, PASS
[2] [2], Grunge, false, PASS
[3] [3], bbbbbbbb, false, PASS
[4] [4], AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA, false, FAIL
[5] [5], A, false, FAIL
[6] [6], %') UNION ALL SELECT 1,2,3,4,5,6; -- , false, FAIL
[7] [7], %') UNION ALL SELECT 1,2,3,4; -- , false, FAIL

FIGURE 4.13 : Résultats du test 3.2

Sous le code sécurisé, en haut, le test est conforme aux attentes (Figure 4.13).

Sous le code non sécurisé, en bas, les intrants 1, 2 et 3 sont conformes aux attentes du test. Les intrants 6 et 7 contiennent des injections SQL, cependant l'intrant 6 a levé une exception au moment de son exécution (Figure 4.14). Tandis que le paramètre 7 a été exécuté sans obstruction sur la base de données. Malgré que les attentes pour 6 et 7 sont de valeur *FAIL*, seul le résultat de l'intrant 7 a eu une valeur *PASS*. Les intrants 4 et 5 ont également été

exécutés sans obstruction sur la base de données. Bien que ces derniers ne sont pas conformes aux attentes, ils ne représentent pas une menace (Figure 4.13).

```
S3.2 Test[6] Exception raised : All queries combined using a UNION, INTERSECT or EXCEPT
 operator must have an equal number of expressions in their target lists.
```

FIGURE 4.14 : Exception - Intrant 6

Scénario 3.3

Rappel du scénario

Un utilisateur effectue une requête sur un module de recherche qui permet de retourner une *Playlists* de chansons. Le test unitaire prend en paramètre des mots clés et doit s'assurer qu'il n'est pas possible de créer une attaque qui tente de récupérer les événements et activités présents dans la table des journaux (*logs*) de notre logiciel fictif.

Résultats

Code sécurisé

Scenario #3.3 - Security Test for Searching Playlists
[1] [1], blackbox, false, PASS
[2] [2], Grunge, true, PASS
[3] [3], bbbbbbb, false, PASS
[4] [4], AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA, false, FAIL
[5] [5], A, false, FAIL
[6] [6], %' UNION SELECT 0, information FROM logs; --, false, FAIL

Code non sécurisé

Scenario #3.3 - Security Test for Searching Playlists
[1] [1], blackbox, false, PASS
[2] [2], Grunge, true, PASS
[3] [3], bbbbbbb, false, PASS
[4] [4], AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA, false, FAIL
[5] [5], A, false, FAIL
[6] [6], %') UNION SELECT 0, information FROM logs; -- , false, FAIL

FIGURE 4.15 : Résultats du test 3.3

Sous le code sécurisé, en haut, le test est conforme aux attentes (Figure 4.15).

Sous le code non sécurisé, en bas, les intrants 1, 2 et 3 sont conformes aux attentes du test. L'intrant 6 contient une injection SQL qui est exécutée sans obstruction sur la base de données. Bien que les attentes ont une valeur *FAIL*, nous avons eu une valeur *PASS* comme résultat. Les intrants 4 et 5 ont également été exécutés sans obstruction sur la base de données. Bien que ces derniers ne sont pas conformes aux attentes, ils ne représentent pas une menace (Figure 4.15).

Scénario 4.1

Rappel du scénario

Un utilisateur peut effectuer une requête sur un module de recherche qui permet de retourner une liste d'artistes. Le test unitaire prend en paramètre une activité sous forme de phrase et doit s'assurer qu'il n'est pas possible d'insérer une information dans la table des journaux (*logs*).

Résultats

```
Code sécurisé
Scenario #4.1 - Security Test for Adding a Log Input to Log Table by Using Create New User
[1] [1], 297@gmail.com, secret, This is a new User, true, PASS
[2] [2], gilbert.gmail.com, secret, No Comment, false, FAIL
[3] [3], gilbert@gmail.com, secret, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA, false, FAIL
[4] [4], A, secret, No Comment, false, FAIL
[5] [5], gilbert@gmail.com, secret, ');INSERT INTO Logs (information) VALUES('Malicious Log'); -- , false, FAIL
[6] [6], ','secret','NoComment');INSERT INTO Logs (information) VALUES('Malicious Log'); -- , Secret, Please see the Username, false, FAIL

Code non sécurisé
Scenario #4.1 - Security Test for Adding a Log Input to Log Table by Using Create New User
[1] [1], 887@gmail.com, secret, This is a new User, true, PASS
[2] [2], gilbert.gmail.com, secret, No Comment, false, FAIL
[3] [3], gilbert@gmail.com, secret, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA, false, FAIL
[4] [4], A, secret, No Comment, false, FAIL
[5] [5], gilbert@gmail.com, secret, ');INSERT INTO Logs (information) VALUES('Malicious Log'); -- , false, FAIL
[6] [6], ','secret','NoComment');INSERT INTO Logs (information) VALUES('Malicious Log'); -- , Secret, Please see the Username, false, FAIL
```

FIGURE 4.16 : Résultats du test 4.1

Sous le code sécurisé, en haut, le test est conforme aux attentes. Il n'est pas possible d'insérer des données dans la table des journaux (Figure 4.16).

Sous le code non sécurisé, en bas, l'intrant 1 est conforme aux attentes du test. Les intrants 5 et 6 contiennent des injections SQL valides qui sont exécutées sans obstruction sur la base de données. Cependant, puisqu'il n'est pas possible d'exécuter des requêtes empilées (*Stacked*) à partir du connecteur *MySQL*, des exceptions SQL ont été levées (Figures 4.17 et 4.18). Puisque les attentes ont une valeur *FAIL*, nous avons un test valide. Les intrants 2, 3 et 4

ont également été exécutés sans obstruction sur la base de données. Bien que ces derniers ne sont pas conformes aux attentes, ils ne représentent pas une menace (Figure 4.16).

```
S4.1 Test[5] Exception raised : You have an error in your SQL syntax; check the manual
 that corresponds to your MySQL server version for the right syntax to use near 'INSERT
 INTO logs (information) VALUES('Malicious Log'); -- ')' at line 1
```

FIGURE 4.17 : Exception - Intrant 5

```
S4.1 Test[6] Exception raised : You have an error in your SQL syntax; check the manual
 that corresponds to your MySQL server version for the right syntax to use near 'INSERT
 INTO logs (information) VALUES('Malicious Log'); -- ','7e32a729b1226ed127' at line 1
```

FIGURE 4.18 : Exception - Intrant 6

Code non sécurisé sous *MSSQL* : Tout comme sous *MySQL*, l'intrant 1 est conforme aux attentes du test. Les paramètres 5 et 6 qui comportent des injections SQL valides sont exécutés sans obstruction sur la base de données. De plus, puisque les requêtes empilées (*Stacked*) sont fonctionnelles à partir du connecteur *MSSQL*, celles-ci se sont complétées avec succès. Ainsi, bien que les attentes sont de valeur *FAIL*, nous avons eu une valeur *PASS* comme résultat. Les intrants 2, 3 et 4 ont également été exécutés sans obstruction sur la base de données. Bien que ces derniers ne sont pas conformes aux attentes, ils ne représentent pas une menace (Figure 4.19).

Code non sécurisé

Scenario #4.1 - Security Test for Adding a Log Input to Log Table by Using Create New User

[1] [1], 256@gmail.com, secret, This is a new User, true, PASS

[2] [2], gilbert.gmail.com, secret, No Comment, false, FAIL

[3] [3], gilbert@gmail.com, secret, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA, false, FAIL

[4] [4], A, secret, No Comment, false, FAIL

[5] [5], gilbert@gmail.com, secret, ');INSERT INTO logs (information) VALUES('Malicious Log'); -- , false, FAIL

[6] [6], ','secret','NoComment');INSERT INTO logs (information) VALUES('Malicious Log'); -- , Secret, Please see the Username, false, FAIL

FIGURE 4.19 : Résultats du test 4.1 non sécurisé sous MSSQL

Scénario 4.2

Rappel du scénario

Un utilisateur souhaite effectuer une requête de mise à jour de son adresse courriel en tant que client. Le test unitaire prend en paramètre l'ancienne et la nouvelle adresse courriel et doit s'assurer qu'il n'est pas possible d'exécuter une requête d'insertion de données, de mise à jour ou encore de suppression de données.

Résultats

Code sécurisé

Scenario #4.2 - Security Test for Creating a new application user, by using email update

[1] [1], daan_peeters@apple.be, daan_peeters@apple.be, true, PASS
[2] [2], dummy@gmail.com, dummy@gmail.com, false, PASS
[3] [3], suzz@AAAAAAAAAAAAAAAAAAAAAAAAA.com, dummy@gmail.com, false, FAIL
[4] [4], h@g.com, h@g.com, false, FAIL
[5] [5], '; INSERT INTO useraccount (Username, Password) VALUES('admin@acme.com','2bb80d537b1da3e38bd30361aa855686bde0eacd7162fef6a25fe97bf527a25b'); -- , dummy@gmail.com, false, FAIL

Code non sécurisé

Scenario #4.2 - Security Test for Creating a new application user, by using email update

[1] [1], daan_peeters@apple.be, daan_peeters@apple.be, true, PASS
[2] [2], dummy@gmail.com, dummy@gmail.com, false, PASS
[3] [3], suzz@AAAAAAAAAAAAAAAAAAAAAAAAA.com, dummy@gmail.com, false, FAIL
[4] [4], h@g.com, h@g.com, false, FAIL
[5] [5], ';INSERT INTO useraccount (Username, Password) VALUES('admin@acme.com','2bb80d537b1da3e38bd30361aa855686bde0eacd7162fef6a25fe97bf527a25b'); -- , dummy@gmail.com, false, FAIL

FIGURE 4.20 : Résultats du test 4.2

Sous le code sécurisé, en haut, le test est conforme aux attentes (Figure 4.20).

Sous le code non sécurisé, en bas, les intrants 1 et 2 sont conformes aux attentes du test. L'intrant 5 contient une injection SQL valide qui est exécutée sans obstruction sur la base de données. Cependant, puisqu'il n'est pas possible d'exécuter des requêtes empilées (*Stacked*) à partir du connecteur *MySQL*, une exception SQL a été levée (Figure 4.21). Puisque les attentes sont de valeur *FAIL*, nous avons néanmoins un test valide pour ce dernier. Les intrants 3 et 4 ont également été exécutés sans obstruction sur la base de données. Bien que ces derniers ne sont pas conformes aux attentes, ils ne représentent pas une menace (Figure 4.20).

```
S4.2 Test[5] Exception raised : You have an error in your SQL syntax; check the manual
 that corresponds to your MySQL server version for the right syntax to use near 'INSERT
 INTO useraccount (Username, Password) VALUES('admin@acme.com','2bb80d537b' at line 1
```

FIGURE 4.21 : Exception - Intrant 5

Code non sécurisé sous *MSSQL* : Le seul résultat qui diffère de ceux sous *MySQL* est celui sous l'intrant 5. Ce dernier comporte une injection SQL et a été exécuté sans obstruction sur la base de données *MSSQL*. Puisque les requêtes empilées (*Stacked*) sont fonctionnelles à partir du connecteur *MSSQL*, celle-ci s'est complétée avec succès. Malgré nos attentes de valeur *FAIL*, nous avons eu une valeur *PASS* comme résultat, confirmant que l'injection SQL s'est complétée avec succès (Figure 4.22).

Code non sécurisé

Scenario #4.2 - Security Test for Creating a new application user, by using email update

[1] [1], daan_peeters@apple.be, daan_peeters@apple.be, true, PASS

[2] [2], dummy@gmail.com, dummy@gmail.com, false, PASS

[3] [3], suzz@AAAAAAAAAAAAAAAAAAAAAAAA.com, dummy@gmail.com, false, FAIL

[4] [4], h@g.com, h@g.com, false, FAIL

[5] [5], ';INSERT INTO useraccount (Username, Password) VALUES('admin@acme.com','2bb80d537b1da3e38bd30361aa855686bde0eacd7162fef6a25fe97bf527a25b'); -- , dummy@gmail.com, false, FAIL

FIGURE 4.22 : Résultats du test 4.2 non sécurisé sous MSSQL

Scénario 4.3

Rappel du scénario

Un utilisateur peut effectuer une requête de mise à jour de son adresse courriel en tant que client. Le test unitaire prend en paramètre une nouvelle adresse et une adresse courriel existante. Le test doit s'assurer qu'il n'est pas possible d'insérer une requête malveillante complète à la suite de la requête légitime, laquelle permet l'exécution de la destruction d'une table.

Résultats

Code sécurisé

Scenario #4.3 - Security Test for Deleting a DB Table, by using Email update (DB User privilege validation)

[1] [1], daan_peeters@apple.be, daan_peeters@apple.be, true, PASS
[2] [2], dummy@gmail.com, dummy@gmail.com, false, PASS
[3] [3], suzz@AAAAAAAAAAAAAAAAAAAAAAAAA.com, dummy@gmail.com, false, FAIL
[4] [4], h@g.com, h@g.com, false, FAIL
[5] [5], ';DROP TABLE dummy; -- , dummy@gmail.com, false, FAIL

Code non sécurisé

Scenario #4.3 - Security Test for Deleting a DB Table, by using Email update (DB User privilege validation)

[1] [1], daan_peeters@apple.be, daan_peeters@apple.be, true, PASS
[2] [2], dummy@gmail.com, dummy@gmail.com, false, PASS
[3] [3], suzz@AAAAAAAAAAAAAAAAAAAAAAAAA.com, dummy@gmail.com, false, FAIL
[4] [4], h@g.com, h@g.com, false, FAIL
[5] [5], ';DROP TABLE dummy; -- , dummy@gmail.com, false, FAIL

FIGURE 4.23 : Résultats du test 4.3

Sous le code sécurisé, en haut, le test est conforme aux attentes (Figure 4.23).

Sous le code non sécurisé, en bas, les intrants 1 et 2 sont conformes aux attentes du test. L'intrant 5 contient une injection SQL valide qui est exécutée sans obstruction sur la base de données. Cependant, puisqu'il n'est pas possible d'exécuter des requêtes empilées (*Stacked*) à partir du connecteur *MySQL*, une exception SQL a été levée (Figure 4.24). Puisque les attentes sont de valeur *FAIL*, nous avons néanmoins un test valide pour ce dernier. Les intrants 3 et 4

ont également été exécutés sans obstruction sur la base de données. Bien que ces derniers ne sont pas conformes aux attentes, ils ne représentent pas une menace (Figure 4.23).

```
S4.3 Test[5] Exception raised : You have an error in your SQL syntax; check the manual
 that corresponds to your MySQL server version for the right syntax to use near 'DROP TABLE
  dummy; -- '' at line 1
```

FIGURE 4.24 : Exception - Intrant 5

Code non sécurisé sous *MSSQL* : Le seul résultat sous *MySQL* qui diffère des autres est celui sous l'intrant 5. Ce dernier comporte une injection SQL et a été exécuté sans obstruction sur la base de données *MSSQL*. Puisque les requêtes empilées (*Stacked*) sont fonctionnelles à partir du connecteur *MSSQL*, celle-ci s'est complétée avec succès. Malgré nos attentes de valeur *FAIL*, nous avons eu une valeur *PASS* comme résultat, confirmant que l'injection SQL s'est complétée avec succès. Bien que la table nommée *dummy* n'existe pas, la requête SQL malveillante a été exécutée avec succès. (Figure 4.25)

Code non sécurisé

Scenario #4.3 - Security Test for Deleting a DB Table, by using Email update (DB User privilege validation)

[1] [1], daan_peeters@apple.be, daan_peeters@apple.be, true, PASS
[2] [2], dummy@gmail.com, dummy@gmail.com, false, PASS
[3] [3], suzz@AAAAAAAAAAAAAAAAAAAAAAAA.com, dummy@gmail.com, false, FAIL
[4] [4], h@g.com, h@g.com, false, FAIL
[5] [5], ';DROP TABLE dummy; -- , dummy@gmail.com, false, FAIL

FIGURE 4.25 : Résultats du test 4.3 non sécurisé sous MSSQL

Scénario 4.4

Rappel du scénario

Un utilisateur peut effectuer une requête qui retourne les informations d'un utilisateur. Le test unitaire prend en paramètre une adresse courriel et doit s'assurer qu'il n'est pas possible pour un attaquant d'effectuer une injection SQL en exécutant une procédure SQL.

Résultats

Code sécurisé

Scenario #4.4 - Security Test for Executing a Sleep of 5 seconds

[1] [1], bjorn.hansen@yahoo.no, true, PASS

[2] [2], bjorn@yahoo.no, false, PASS

[3] [3], bjorn.hansen.yahoo.no, false, FAIL

[4] [4], '; SELECT SLEEP(5); -- , false, FAIL

Code non sécurisé

Scenario #4.4 - Security Test for Executing a Sleep of 5 seconds

[1] [1], bjorn.hansen@yahoo.no, true, PASS

[2] [2], bjorn@yahoo.no, false, PASS

[3] [3], bjorn.hansen.yahoo.no, false, FAIL

[4] [4], '; SELECT SLEEP(5); -- , false, FAIL

FIGURE 4.26 : Résultats du test 4.4

Sous le code sécurisé, le test est conforme aux attentes (Figure 4.26).

Sous le code non sécurisé, les intrants 1 et 2 sont conformes aux attentes du test. L'intrant 4 contient une injection SQL valide qui est exécutée sans obstruction sur la base de données. Cependant, puisqu'il n'est pas possible d'exécuter des requêtes empilées (*Stacked*) à partir du connecteur *MySQL*, une exception SQL a été levée (Figure 4.27). Puisque les attentes sont de valeur *FAIL*, nous avons un test valide. Concernant les paramètres 3 et 4, ceux-ci ont

également été exécutés sans obstruction sur la base de données. Bien que ces derniers ne sont pas conformes aux attentes, ils ne représentent pas une menace (Figure 4.26).

```
S4.3 Test[5] Exception raised : You have an error in your SQL syntax; check the manual
 that corresponds to your MySQL server version for the right syntax to use near 'DROP TABLE
  dummy; -- '' at line 1
```

FIGURE 4.27 : Exception - Intrant 4

Code non sécurisé sous *MSSQL* : Les résultats sont identiques à *MySQL*. Cependant, puisque les requêtes empilées (*Stacked*) sont fonctionnelles à partir du connecteur *MSSQL*, la procédure *SLEEP()* qui est inexistante lève une exception SQL (Figure 4.29). Au lieu d'utiliser la procédure *SLEEP()*, nous aurions dû faire appel à la procédure *WAITFOR DELAY '0 :0 :5' ;* (Figure 4.28).

Code non sécurisé

Scenario #4.4 - Security Test for Executing a Sleep of 5 seconds

[1] [1], bjorn.hansen@yahoo.no, true, PASS

[2] [2], bjorn@yahoo.no, false, PASS

[3] [3], bjorn.hansen.yahoo.no, false, FAIL

[4] [4], '; SELECT SLEEP(5); -- , false, FAIL

FIGURE 4.28 : Résultats du test 4.4 non sécurisé sous MSSQL

```
S4.4 Test[4] Exception raised : 'SLEEP' is not a recognized built-in function name.
```

FIGURE 4.29 : Exception - Intrant 4

RÉSULTATS - PARAMÉTRISATION DE LA BASE DE DONNÉES

Dans les scénarios de tests 5 à 7, nous exposons les scénarios et nous comparons les résultats selon les différences des systèmes de gestion de base de données (*SGBD*). Nous vérifions si une configuration de sécurité appropriée sous *MySQL* est également valide sous *MSSQL* (Figure 4.30).

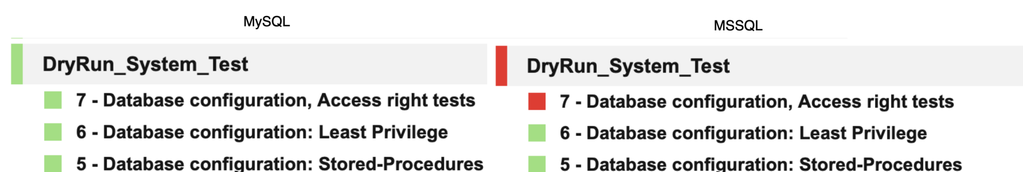

FIGURE 4.30 : Aperçu des résultats des tests 5 à 7

Scénario 5.1

Rappel du scénario : Le test unitaire doit s'assurer que les procédures stockées sont inexistantes (ou inaccessibles) par les comptes de service autres que *root*.

FIGURE 4.31 : Résultats des tests sous MySQL et MSSQL

Le test effectue une simple validation en exécutant l'appel d'une procédure stockée, normalement disponible sous *MySQL*. Cet appel vérifie si la table *UserAccount* existe dans la base de données *Chinook*.

```
CALL Chinook.table\_exists('Chinook', 'UserAccount');
```

Listing 4.1 – Procédure de validation de l'existance de la table UserAccount sous MySQL

Résultats

Sous *MySQL*, l'appel soulève une exception indiquant que la procédure est inexistante (Figure 4.32) se soldant ainsi par un succès (*PASS*). Tandis que sous *MSSQL* une exception est levée indiquant qu'il ne reconnaît pas l'appel de la procédure (Figure 4.33).

```
S5.1 Exception raised : PROCEDURE Chinook.table_exists does not exist
```

FIGURE 4.32 : Exception sous MySQL

À noter que du côté *MSSQL*, l'appel soulève une également une exception, puisque le format de l'appel n'est pas valide (Figure 4.33).

```
S5.1 Exception raised : Incorrect syntax near '.'.
```

FIGURE 4.33 : Exception sous MSSQL

Néanmoins, il aurait été plus approprié d'utiliser l'appel suivant :

```
EXEC sp_tables \@table_name = 'UserAccount',
               \@table_owner = 'dbo',
               \@table_qualifier = 'Chinook';
```

Listing 4.2 – Procédure de validation de l'existance de la table UserAccount sous MSSQL

Scénarios 6.1 à 6.6

Rappel des scénarios

Il s’agit d’exécuter plusieurs tests unitaires qui valident les autorisations pour chacun des comptes de service de la base de données *Chinook* sous chacun des systèmes de gestion de base de données (Figure 4.34).

Résultats

MySQL

6 - Database configuration: Least Privilege

Scenario #6.3 - Security Test for Validating of DB Log User with the PlayList Table
Scenario #6.2 - Security Test for Validating of DB Log User to a Standard Table
Scenario #6.6 - Security Test for Validating that Log User can not Delete the content of the table Logs
Scenario #6.4 - Security Test for Validating of DB Auth User to a Standard Table
Scenario #6.5 - Security Test for Validating of DB Auth User to the Logs Table
Scenario #6.1 - Security Test for Validating of DB Standard User to Logs Table

MSSQL

6 - Database configuration: Least Privilege

Scenario #6.3 - Security Test for Validating of DB Log User with the PlayList Table
Scenario #6.2 - Security Test for Validating of DB Log User to a Standard Table
Scenario #6.6 - Security Test for Validating that Log User can not Delete the content of the table Logs
Scenario #6.4 - Security Test for Validating of DB Auth User to a Standard Table
Scenario #6.5 - Security Test for Validating of DB Auth User to the Logs Table
Scenario #6.1 - Security Test for Validating of DB Standard User to Logs Table

FIGURE 4.34 : Résultats des tests sous MySQL et MSSQL

Les tests de 6.1 à 6.6 valident les autorisations de chacun des comptes de service relatifs à la base de données, c’est-à-dire les comptes **auth**, **std** et **log**. Il est important de mentionner que nous n’avons pas testé le compte **root**, car il s’agit d’un compte à haut privilège qui est présent par défaut.

Nous nous attendons, pour chacun des tests, à la levée d'exceptions comme refus d'accès. Ce que les résultats nous ont confirmé (Figure 4.35).

MySQL

```
S6.3 Exception raised : UPDATE command denied to user 'log'@'192.168.65.1' for table 'playlist'
S6.2 Exception raised : SELECT command denied to user 'log'@'192.168.65.1' for table 'useraccount'
S6.6 Exception raised : DELETE command denied to user 'log'@'192.168.65.1' for table 'logs'
S6.4 Exception raised : SELECT command denied to user 'auth'@'192.168.65.1' for table 'playlist'
S6.5 Exception raised : SELECT command denied to user 'auth'@'192.168.65.1' for table 'logs'
S6.1 Exception raised : SELECT command denied to user 'std'@'192.168.65.1' for table 'logs'
```

MSSQL

```
S6.3 Exception raised : The SELECT permission was denied on the object 'Playlist', database 'Chinook', schema 'dbo'.
S6.2 Exception raised : The SELECT permission was denied on the object 'UserAccount', database 'Chinook', schema 'dbo'.
S6.6 Exception raised : The DELETE permission was denied on the object 'Logs', database 'Chinook', schema 'dbo'.
S6.4 Exception raised : The SELECT permission was denied on the object 'Playlist', database 'Chinook', schema 'dbo'.
S6.5 Exception raised : The SELECT permission was denied on the object 'Logs', database 'Chinook', schema 'dbo'.
S6.1 Exception raised : The SELECT permission was denied on the object 'Logs', database 'Chinook', schema 'dbo'.
```

FIGURE 4.35 : Exceptions des tests sous MySQL et MSSQL, pour les scénarios 6

Scénarios 7.1 et 7.2

Rappel des scénarios

Les tests unitaires valident les droits d'accès et d'exécution de chacun des comptes de service de la base de données (Figure 4.36).

MySQL

7 - Database configuration, Access right tests

Scenario #7.1 - Security Test for Validating that other DB Users than ROOT can not DROP tables.

[1] AUTH_USER, useraccount
[2] LOG_USER, logs
[3] STD_USER, customer

Scenario #7.2 - Security Test for Validating the Number of tables to show, for each DB User.

[1] AUTH_USER, 1
[2] LOG_USER, 1
[3] STD_USER, 12
[4] ROOT_USER, 13

MSSQL

7 - Database configuration, Access right tests

Scenario #7.1 - Security Test for Validating that other DB Users than ROOT can not DROP tables.

[1] AUTH_USER, useraccount
[2] LOG_USER, logs
[3] STD_USER, customer

Scenario #7.2 - Security Test for Validating the Number of tables to show, for each DB User.

[1] AUTH_USER, 1
[2] LOG_USER, 1
[3] STD_USER, 12
[4] ROOT_USER, 13

FIGURE 4.36 : Résultats des tests des scénarios 7

Résultats

Les tests 7.1 et 7.2 valident les droits de chacun des comptes de service de la base de données. Le scénario 7.1 s'assure que les comptes de service qui ne sont pas à hauts privilèges (**auth**, **std** et **log**) n'ont pas les droits de suppression de tables (*DROP Table*). Quant au scénario 7.2, il s'assure que les comptes de service sont limités uniquement aux tables dont ils ont normalement accès.

Afin d'assurer la validité du test 7.1, il est nécessaire que l'appel de la commande *DROP Table* lève une exception relative aux droits d'accès (Figure 4.37). Les résultats pour *MySQL* et *MSSQL* répondent à nos attentes. Concernant le test 7.2, nous utilisons la procédure *SHOW Tables* pour afficher la liste des tables selon les comptes de service. Sous *MySQL*, les résultats

sont concluants. Cependant sous *MSSQL*, les tests lèvent des exceptions SQL puisque la fonction *SHOW Tables* est inexistante sous *MSSQL* (Figure 4.37).

MySQL

```
S7.1 Exception raised : DROP command denied to user 'auth'@'192.168.65.1' for table 'useraccount'
S7.1 Exception raised : DROP command denied to user 'log'@'192.168.65.1' for table 'logs'
S7.1 Exception raised : DROP command denied to user 'std'@'192.168.65.1' for table 'customer'
```

MSSQL

```
S7.1 Exception raised : Cannot drop the table 'UserAccount', because it does not exist or you do not have permission.
S7.1 Exception raised : Cannot drop the table 'Logs', because it does not exist or you do not have permission.
S7.1 Exception raised : Cannot drop the table 'Customer', because it does not exist or you do not have permission.
S7.2 Exception raised : Could not find stored procedure 'SHOW'.
S7.2 Exception raised : Could not find stored procedure 'SHOW'.
S7.2 Exception raised : Could not find stored procedure 'SHOW'.
S7.2 Exception raised : Could not find stored procedure 'SHOW'.
```

FIGURE 4.37 : Exceptions des tests sous MySQL et MSSQL, pour les scénarios 7

SIMULATION D'UNE ATTAQUE

Les tests unitaires ont permis de confirmer le fonctionnement attendu du code source et d'assurer la conformité aux normes de sécurité. Cependant, nous avons également voulu tester la validité de notre approche en évaluant l'efficacité de nos contrôles de sécurité avec un outil externe tel que *SQLMap* [34]. Nous avons donc créé une suite d'APIs, qui font appel à chacune des fonctions de la couche métier de notre logiciel (Figure 4.38).

34. SQL Map. 2024. Récupéré de https://sqlmap.org

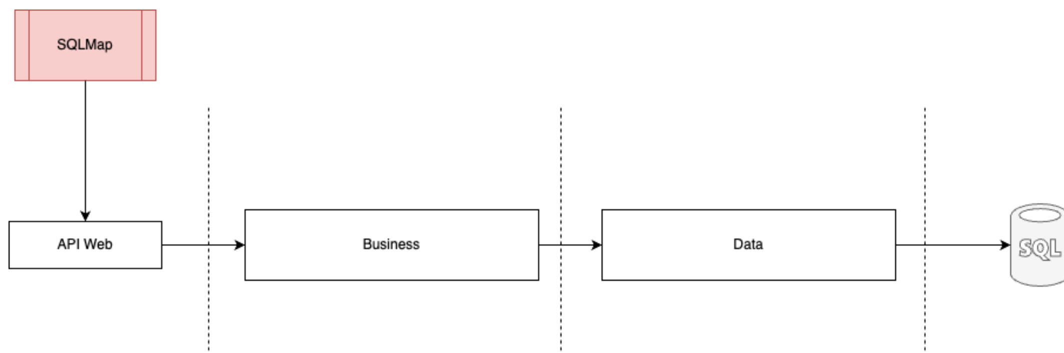

FIGURE 4.38 : Attaque par SQLMap sur les APIs.

Nous savons que l'utilisation des requêtes paramétrées (*Prepared Statements*) bloque toute tentative d'attaque par injection SQL. Bien que nous anticipions des résultats négatifs d'une attaque par SQLMap sur le code sécurisé, il est intéressant de constater à quel niveau de contrôle, la portion de code source sécurisé a interrompu l'attaque.

Ci-dessous, la commande *SQLMap* lance une attaque contre l'API qui invoque la fonction d'authentification sécurisée de notre application.

```
sqlmap -u http://localhost:4567/secure/user/login
--data "{\"user\":\"*\",\"pwd\":\"*\"}" -p "user,pwd"
--method POST --level=3
```

Listing 4.3 – Appel de la fonction sécurisée d'authentification

Ci-dessous, une capture d'écran des résultats de l'attaque (Figure 4.39). *SQLMap* n'a pas été en mesure d'exécuter une attaque par injection SQL.

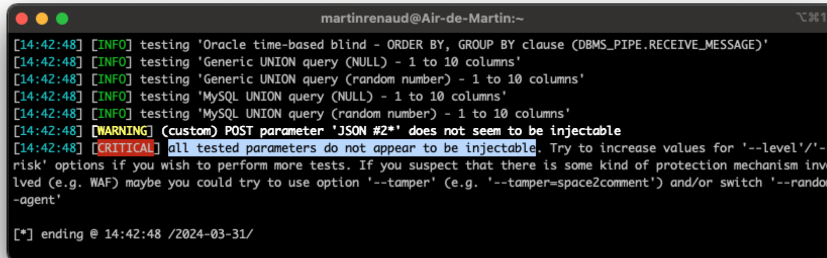

FIGURE 4.39 : Résultat non concluant, provenant de l'attaque SQLMap

Ci-dessous, la commande *SQLMap* lance une attaque contre l'API qui invoque la fonction d'authentification non sécurisée de notre application.

```
sqlmap -u http://localhost:4567/user/login
--data "{\"user\":\"*\",\"pwd\":\"*\"}" -p "user,pwd"
--method POST --level=3
```

Listing 4.4 – Appel de la fonction non sécurisée d'authentification

Il convient de noter que faute de contrôle de sécurité dans le code non sécurisé, *SQLMap* a réussi à injecter du code SQL au travers des APIs. Nous constatons que l'attaque a réussi à exploiter avec succès des vulnérabilités sensibles aux injections SQL. Ci-dessous, une capture d'écran des résultats de l'attaque (Figure 4.40).

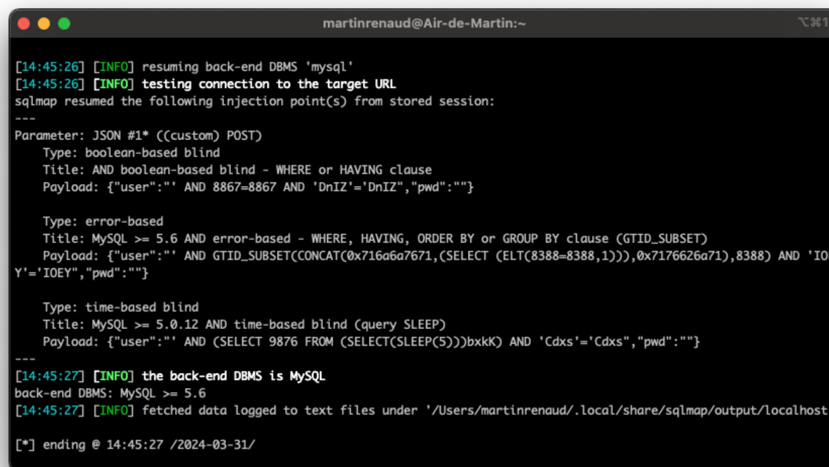

FIGURE 4.40 : Résultat concluant, provenant de l'attaque SQLMap

Il convient de souligner que dans cette simulation, notre objectif n'est pas d'exposer les résultats des attaques réalisées sur chacune des APIs. Nous souhaitons plutôt présenter un aperçu de l'efficacité de notre approche en simulant une attaque à l'aide d'un outil existant et de déterminer à quel moment les contrôles de sécurité ont interrompu les attaques. En examinant la Figure 4.41, nous réalisons que l'attaque a été interrompue au tout début du processus de contrôle de sécurité, au moment de la vérification de la longueur des chaînes de caractères.

```
Call to /secure/user/login
Parameters [user,pwd]: [" AND 3793=(SELECT (CASE WHEN (3793=3793) THEN 3793 ELSE (SELECT 8088 UNION
 SELECT 7251) END))-- cafo,])
Returned msg : String length does not meet the requirements. Must be between 10 and 30 characters.

Call to /secure/user/login
Parameters [user,pwd]: []) AND 2262=6278-- dAUn,])
Returned msg : String length does not meet the requirements. Must be between 6 and 20 characters.

Call to /secure/user/login
Parameters [user,pwd]: []) AND 2831=2831-- nouL,])
Returned msg : String length does not meet the requirements. Must be between 6 and 20 characters.
```

FIGURE 4.41 : Exceptions levées par nos contrôles, lors de la simulation de l'attaque

4.5 CONCLUSION

En comparant les résultats entre le code sécurisé et le code non sécurisé, nous constatons dans cette démarche, un avantage pour les équipes de développement, lequel devra être confirmé avec une étude empirique dans des travaux futurs. Malgré que certains tests peuvent paraître élémentaires ou redondants, l'enjeu était de démontrer notre processus de réflexion en exposant nos idées depuis des scénarios clairs et aisés à comprendre. En utilisant les techniques selon les bonnes pratiques de l'industrie et en les assemblant dans une approche méthodique de tests, nous avons démontré qu'il est possible d'automatiser des tests et ainsi établir une défense

au niveau du code source (*Code-Level Defense*) et une défense au niveau de la plateforme [35] (*Platform-Level Defense*) (Galluccio *et al.*, 2020).

De plus, nous sommes en mesure de constater la valeur de notre démarche, en comparant les résultats provenant des tests sur du code sécurisé et du code non sécurisé. Les tests effectués sur du code source sécurisé représentent notre base de validation. Les mesures de contrôles mises en place doivent impérativement traiter les intrants et s'assurer que les injections SQL soient interrompues. Les scénarios ont donc été créés dans ce sens. La pleine justification de la démonstration se manifeste lorsque nous testons les segments de code non sécurisés en utilisant les mêmes intrants de test que ceux appliqués aux segments de code sécurisé. En comparant les résultats, nous pouvons établir la différence et démontrer l'efficacité. Le bénéfice de l'utilisation des tests unitaires réside dans notre capacité à contrôler les paramètres d'entrée ainsi que les résultats anticipés. Si le résultat obtenu ne correspond pas à celui prévu, le test est considéré comme infructueux, exigeant ainsi une rectification et une nouvelle exécution du test.

Ainsi, à notre première question *Est-ce qu'une approche par tests unitaires, dans un contexte de sécurité des systèmes d'information, peut permettre d'éviter l'introduction de vulnérabilités sensibles aux attaques par injection SQL ?*, la réponse est affirmative. Les tests unitaires permettent d'anticiper les erreurs dans le code source susceptible de permettre des injections SQL. À la seconde question *Comment créer des scénarios de tests unitaires qui permettent de couvrir l'ensemble de toutes les déclinaisons connues d'attaques par injection SQL pouvant affecter un système d'information ?*, nous avons développé des scénarios qui vérifient non seulement les parties du code source vulnérables, mais aussi de contrôler et de

35. Dans le cadre de ce projet, la défense au niveau de la plateforme se situe au niveau de la paramétrisation des systèmes de gestion de base de données. Toutefois, des règles de défense auraient également pu être utilisées dans un contexte de validation des paramètres d'un serveur internet ou d'une infrastructure infonuagique.

garantir l'intégrité au fil du temps de certains paramètres de sécurité associés aux bases de données.

Aurait-on pu atteindre un niveau d'exhaustivité supérieur des scénarios ? Très probablement. Nous aurions pu concevoir des scénarios de tests plus élaborés, capables de valider de manière systématique chaque paramètre d'entrée issu d'intrants potentiels manipulés par les utilisateurs, en explorant différentes variations d'une même injection SQL. En dépit de l'intégration de quelques contrôles de sécurité utilisant des expressions régulières, il aurait été possible d'élargir ces mesures en incluant des validations de motifs plus sophistiquées. Une initiative, avec le support de l'intelligence artificielle, pourrait représenter une piste d'exploration intéressante. Néanmoins, les objectifs de notre projet étaient davantage axés sur la démonstration du réalisme de notre approche, ainsi que sur la valeur qu'elle ajoute en termes de prévention des vulnérabilités aux injections SQL.

Finalement, il est important de souligner que notre approche est intimement liée aux ensembles de données et au contexte que nous avons sélectionné pour illustrer notre démonstration, et potentiellement jeter les bases de cette approche. Nous croyons qu'il est possible de standardiser et généraliser notre méthode au travers de nouvelles expérimentations, voire même en établissant un nouveau cadre de travail (*Framework*).

CHAPITRE V
DISCUSSION

5.1 IMPLICATION DE L'APPROCHE POUR LES DÉVELOPPEURS

L'adoption d'une approche axée sur la vérification du bon fonctionnement des différentes portions du code source, à travers des tests unitaires, revêt une importance déterminante dans le domaine du développement logiciel. Pour les développeurs, cette méthodologie représente bien plus qu'une simple validation de la conformité du code source. En effet, celle-ci incarne une démarche **proactive** visant à garantir la qualité du logiciel en identifiant les éventuelles anomalies. Dans cette perspective, cette section explore les limites de cette approche sur le processus de développement, soulignant son rôle dans la diminution des risques relatifs aux défaillances dans le code source.

Dans le cadre de ce projet, il est observé que les *développeurs* jouent un rôle essentiel. Leur responsabilité ne se limite pas seulement à la rédaction du code source conformément aux exigences définies, mais s'étend également à la modélisation et à la rédaction des tests unitaires. En plus des tests fonctionnels, il est impératif que les tests soient conçus pour garantir l'absence de vulnérabilités aux injections SQL. Il convient de noter que, dans le cas de projets de grande envergure, les développeurs peuvent collaborer avec des spécialistes de l'assurance qualité (*Quality Assurance*) pour concevoir et rédiger les cas d'usages qui seront traduits en code par la suite.

En somme, notre approche implique la participation quasi entière des développeurs. Puisqu'il s'agit d'une méthode qui encourage ces derniers à appréhender les défis de sécurité liés aux menaces directes sur les sources de données, celle-ci propose une approche préventive

dans le processus de rédaction du code source qui permet d'anticiper les erreurs et les pièges associés aux menaces.

5.1.1 QUELLES CONNAISSANCES LES DÉVELOPPEURS ONT BESOIN POUR CRÉER LES TESTS?

Pour créer des tests unitaires visant à prévenir les injections SQL, les développeurs doivent avoir une compréhension approfondie de plusieurs concepts et techniques liés aux tests unitaires :

1. Les concepts de validation et de filtrage des données utilisateur : Les développeurs doivent mettre en œuvre les mécanismes de validation et de filtrage appropriés des intrants provenant des utilisateurs pour s'assurer que les entrées ne contiennent pas d'instructions malveillantes.
2. Le langage SQL : La maîtrise des langages SQL est nécessaire. Les développeurs doivent être en mesure de construire et exécuter des requêtes SQL, ainsi que de comprendre l'impact des injections lorsque la requête dépend d'entrées provenant d'utilisateurs.
3. La compréhension des injections SQL : Les développeurs doivent être conscients de ce qu'est une injection SQL et comment elle peut être exploitée. Cela inclut également la connaissance des différentes formes d'injections SQL, ainsi que les subtilités reliées aux éditeurs des différents systèmes de gestion de bases de données existants (SGBD).
4. Les pratiques de programmation sécurisées : Les développeurs doivent connaître et comprendre les concepts de meilleures pratiques de programmation (*Best Practices*) pour éviter les pièges. Cela inclut l'utilisation de requêtes paramétrées (*Prepared Statements*) ou des procédures stockées (*StoredProc*) plutôt que la simple concaténation de chaînes de caractères pour construire des requêtes SQL, avant de les envoyer vers la base de données.

5. Les *Frameworks* de tests unitaires : Les développeurs doivent être familiers avec les *frameworks* de tests unitaires attachés aux langages de programmation qu'ils utilisent, tels que *JUnit* pour Java ou *NUnit* pour .NET. En ce sens, ceux-ci permettront d'adapter les tests en fonction du langage et des fonctionnalités offertes.
6. La gestion des données de test : Les développeurs doivent s'assurer que les données de test sont correctement gérées afin de garantir la reproductibilité des tests. Cela peut inclure l'utilisation de jeux de données de test spécifiques ou la mise en place de données de tests temporaires.

En maîtrisant chacun des concepts et techniques précédents, les développeurs seront en mesure de rédiger non seulement du code source sécuritaire, mais également des tests unitaires spécifiques et adaptés pour prévenir les vulnérabilités relatives aux injections SQL.

5.1.2 QUELLES CONNAISSANCES LES DÉVELOPPEURS ONT BESOIN POUR EXÉCUTER LES TESTS?

Il peut arriver que les développeurs qui exécutent les tests diffèrent de ceux qui les ont rédigés. Une fois que les tests unitaires sont créés et opérationnels, il ne nous parait pas nécessaire que les développeurs aient une compréhension approfondie de leur fonctionnement, à condition qu'ils aient une compréhension de leur utilité et sachent comment apporter des corrections au niveau du code source, le cas échéant.

Les connaissances suivantes sont nécessaires :

1. L'environnement d'exécution : Les développeurs doivent connaître l'environnement dans lequel les tests unitaires seront exécutés. Cela comprend, entre autres, la configuration du système, la base de données, les dépendances et les variables.

2. Les paramètres des tests : Les développeurs doivent être capables de spécifier et de comprendre les différents paramètres de test. Cela peut inclure les paramètres de configuration et les options de test spécifiques relatifs aux contextes de la mission du logiciel (ex. : le contrôle des accès).
3. L'analyse des résultats : Les développeurs devront analyser les résultats provenant des tests unitaires. Ils doivent être en mesure d'interpréter les rapports de test générés par les outils, d'identifier les tests en échecs et diagnostiquer les problèmes.
4. Le débogage des tests échoués : En cas d'échec d'un test unitaire, les développeurs doivent être compétents dans le débogage pour identifier la cause sous-jacente du problème et être en mesure de localiser l'emplacement de la vulnérabilité.
5. La remédiation : Le développeur doit être en mesure de comprendre le problème relatif à l'échec du test et effectuer la correction qui s'impose afin de remédier à la situation.

En maîtrisant ces aspects, les développeurs peuvent exécuter leurs tests unitaires de manière efficace, identifier les problèmes potentiels dans le code source et maintenir la qualité du logiciel tout au long du cycle de développement.

5.2 LIMITATION DE LA RECHERCHE

Il est vrai que toute étude est soumise à des limites qui peuvent affecter la portée et la validité de ses résultats. Ces limites peuvent être causées par une variété de facteurs, notamment la méthodologie utilisée, les ressources disponibles, les contraintes temporelles et les biais potentiels. Pour interpréter correctement les résultats de la recherche et éviter des conclusions excessives ou simplistes, il est essentiel de prendre connaissance de ces limites. Cette section examine les différentes facettes des limites de la recherche, en nous basant sur le concept de la *menace à la validité*.

5.2.1 MENACES À LA VALIDITÉ : INTERNE

1. La diversité des environnements de développement : Des variations dans les conditions environnementales peuvent potentiellement affecter les résultats. Les applications peuvent être déployées sur différentes plates-formes, avec des configurations de bases de données variées, pouvant également utiliser des versions différentes du langage. Les tests unitaires conçus pour une plate-forme spécifique peuvent ne pas être directement transférables à d'autres environnements ou d'autres langages, créant ainsi des lacunes dans la couverture de sécurité.
2. Les variations dans les systèmes de gestion de bases de données (SGBD) : La variété des SGBD compatibles, telle que MySQL, PostgreSQL, Oracle ou SQL Server introduit une complexité supplémentaire. Chacun de ces systèmes peut avoir des spécificités dans la manière dont il traite les requêtes SQL, nécessitant une adaptation des tests unitaires pour chaque SGBD spécifique. Cette diversité peut entraîner des incohérences dans les tests si ces différences ne sont pas prises en compte lors de l'implémentation de ceux-ci dans les environnements.
3. La qualité de la conception des tests : Puisque l'efficacité des tests unitaires dépend largement de la qualité de la conception des tests, des scénarios de tests insuffisamment élaborés peuvent donner une fausse impression de sécurité. Une conception de tests rigoureuse et axée sur des cas d'utilisation réels est essentielle pour maximiser l'efficacité de cette approche.
4. La difficulté de reproduire des conditions réelles : La fidélité des tests unitaires à reproduire fidèlement les conditions d'utilisation réelles constitue une limite méthodologique importante. Les tests peuvent ne pas toujours refléter la charge de travail réelle ou d'autres facteurs du monde réel. Cela peut entraîner des résultats biaisés et laisser des failles de sécurité non détectées.

5. La maintenance des tests unitaires : La maintenance continue des tests unitaires est un aspect critique souvent sous-estimé. À mesure que le code évolue, les tests unitaires doivent être constamment mis à jour pour refléter ces changements. La complexité accrue des tests peut rendre leur maintenance plus délicate, nécessitant un engagement continu de l'équipe de développement.
6. Charge de travail supplémentaire pour les développeurs : Tel qu'il a été mentionné précédemment, la conception et la mise en œuvre de tests unitaires pour prévenir les injections SQL exigent une connaissance approfondie dans le langage, les bases de données et la sécurité au niveau du développement logiciel. Cela peut représenter une charge de travail significative pour les développeurs, qui doivent équilibrer la sécurité avec des délais de développement souvent serrés.

5.2.2 MENACES À LA VALIDITÉ : EXTERNE

1. La complexité et l'évolution des injections SQL : La sophistication croissante des attaques par injection SQL pose un défi pour la conception des tests unitaires. Les scénarios d'attaque par injection SQL peuvent être complexes, exploitant des failles subtiles dans la logique du code. Les tests unitaires peuvent ne pas couvrir tous les scénarios, laissant des vulnérabilités non détectées, en particulier lorsque la variété des attaques possibles est considérée. Comme nous l'avons constaté dans le *Chapitre 4*, les outils de test tels que *SQLMap* peuvent être utilisés pour se familiariser avec les nouvelles subtilités relatives aux injections SQL (Baklizi *et al.*, 2022).

2. La consistance des outils : Bien que les outils que nous avons employés ne soient pas spécifiquement conçus pour détecter les injections SQL, nous sommes persuadés qu'ils produisent des résultats cohérents à chaque exécution, assurant ainsi une évaluation fiable de la résilience du système face aux attaques par injection SQL. Cependant, il est plausible de considérer que des outils plus spécialisés ou même adaptés aux divers styles d'injections SQL pourraient apporter des informations supplémentaires dans les résultats. Ces informations pourraient inclure des détails tels que le type d'attaque spécifique ou la validité de l'injection par rapport au langage SQL utilisé.

   À titre d'exemple, dans un rapport de tests, les résultats pourraient afficher non seulement les succès et les échecs, mais indiquer également les tests unitaires qui ont effectué des tentatives d'attaque sous *MySQL*, qui normalement ne devraient être applicables que sous *Microsoft SQL Server*.

3. La généralisation : La généralisation constitue nécessairement un aspect essentiel de notre approche. La démonstration que nous présentons est intrinsèquement liée au langage de programmation et à l'ensemble de données que nous avons employés dans le cadre de ce projet. La création des tests unitaires requiert une démarche manuelle de

réflexion et de rédaction, alignée sur le contexte spécifique de la mission du système à sécuriser. Il va de soi que, dans le cas d'un contexte différent, les mécanismes de validation, de neutralisation et de paramétrisation des requêtes resteraient sensiblement les mêmes, cependant les scénarios devraient être adaptés à ce dernier.

5.2.3 PERSPECTIVES D'AMÉLIORATION

En offrant une vue des enjeux et en suscitant la réflexion sur les pistes d'optimisation envisageables, nous croyons qu'il serait possible d'améliorer et normaliser notre approche.

Voici quelques propositions d'amélioration :

1. Généralisation : Les projets à venir pourraient illustrer la possibilité d'améliorer la neutralité de la méthode. Il serait envisageable de concevoir un standard ou un guide qui établit les bases de la pratique (*Baselines*) pour une équipe de développement dans l'implémentation d'une structure de tests unitaires axés sur la prévention des vulnérabilités aux injections SQL, et ce, peu importe le langage de programmation, le langage SQL, la mission du système et le contexte des scénarios de tests.
2. Automatisation des tests : L'automatisation des tests pourrait contribuer à la création d'un processus de développement plus robuste. Les tests automatisés pourraient également être intégrés aux *pipelines* de développement, assurant ainsi une évaluation continue de la sécurité du code à chaque modification. Cela favoriserait une approche proactive de la sécurité, en introduisant une forme de portail de validation (*Gating*) tout en réduisant la probabilité que des vulnérabilités ne soient pas introduites dans le temps.

3. Amélioration des outils : Il pourrait être intéressant de normaliser l'approche à l'aide d'outils tels que *JUnit*[36] et *Cucumber*[37] afin de créer un (*framework*) dédié aux tests unitaires contre les injections SQL, et ainsi faciliter la scénarisation des tests unitaires et d'intégration, tout en générant le code source nécessaire pour exécuter les tests. Ceci en fonction du langage de programmation et du système de gestion de base de données, mais également en fonction des scénarios de tests unitaires souhaités.

Bien que la recherche sur les tests unitaires dans un contexte de sécurité présente des limites, une compréhension approfondie de ces défis offre une opportunité d'amélioration. L'intégration de perspectives technologiques et méthodologiques, combinées à une vision plus globale de la sécurité peut mener à l'augmentation de l'efficacité des tests unitaires, en matière de prévention des vulnérabilités liées aux injections SQL.

5.3 CRITIQUE DE L'APPROCHE

La démarche visant à utiliser des tests unitaires pour éviter les portions de code source vulnérables aux injections SQL est certainement une approche viable dans la quête de la sécurité des systèmes d'information. Cependant, une analyse critique de la méthode révèle que les tests unitaires servant à prévenir les injections SQL ne peuvent être l'unique mesure de protection. Elle offre certainement une première ligne de défense, mais elle doit être accompagnée d'autres mesures.

La méthodologie repose principalement sur des tests unitaires rédigés par les développeurs eux-mêmes. Bien que l'approche puisse refléter la réalité des pratiques de développement,

36. JUnit 5. 2024. Récupéré de https://junit.org/junit5/

37. Cucumber. 2024. Récupéré de https://cucumber.io

elle soulève des préoccupations quant à l'objectivité des développeurs dans l'évaluation des vulnérabilités.

De plus, l'étude ne prend pas suffisamment en compte la diversité des environnements de développement et des langages de programmation. Les tests unitaires peuvent varier considérablement en fonction du langage et du *framework* utilisés. En conséquence, l'efficacité dans la prévention des injections SQL pourrait différer. Une approche plus globale aurait dû inclure une variété de langages et de contextes de développement pour rehausser la pertinence des conclusions.

Une autre lacune concerne l'absence d'une comparaison directe avec d'autres méthodes de prévention des injections SQL. L'étude se concentre exclusivement sur l'efficacité des tests unitaires, sans établir de comparaisons avec des techniques complémentaires telles que l'utilisation de procédures stockées ou d'outils d'analyse statique du code (*SAST*). De plus, l'évaluation de l'efficacité des tests unitaires est souvent réalisée dans des environnements en hors production ou en laboratoire plutôt que dans des conditions opérationnelles réelles, ce qui souligne la nécessité d'avoir une lecture prudente des résultats.

Enfin, dans un angle moins technique, le projet de recherche ne prend pas en compte les coûts associés à la mise en œuvre des tests unitaires à plus grande échelle. Il aurait été intéressant de connaître, d'un point de vue financier, les impacts d'une telle démarche. Bien que nous croyons dans l'efficacité et la pertinence de notre approche, les ressources nécessaires pour élaborer et maintenir des tests unitaires pourraient être significatives et une évaluation des coûts aurait pu nous fournir des informations intéressantes sur la viabilité de cette approche dans des contextes réels de développement de logiciel.

5.4 CONCLUSION

Il parait évident que l'apport des développeurs, dans une perspective d'amélioration de la sécurité des logiciels, est nécessaire. Leur expertise et leur engagement jouent un rôle indispensable pour assurer la sécurité des projets de développement. L'approche de ce projet remet entre les mains ces professionnels une portion importante de cet enjeu de sécurité.

À priori, les résultats obtenus suggèrent que les tests unitaires peuvent jouer un rôle important dans la défense contre les injections SQL, en permettant de prévenir efficacement les vulnérabilités dès le début du processus de développement. Néanmoins, il est impératif d'intégrer ceux-ci dans un processus plus global de sécurité, en tenant compte des spécificités et du contexte de développement. Le projet de recherche aurait également bénéficié d'une comparaison directe avec d'autres stratégies de prévention contre les injections SQL et d'une évaluation des coûts associés à sa mise en œuvre. Une perspective plus large aurait également permis d'apporter des recommandations pratiques plus nuancées pour les développeurs et les professionnels de la sécurité.

La diversité de langages de programmation, d'environnements de développement et de plates-formes rend la démarche complexe en apparence, cependant nous sommes d'avis qu'elle offre une base de connaissance intéressante et que la généralisation des concepts de tests unitaires dont la mission est la prévention des injections SQL serait un atout majeur. Nous croyons avoir établi les fondations d'une nouvelle approche préventive pour contrer les vulnérabilités liées aux injections SQL. Cette approche s'appuie sur une pratique de test bien établie dans l'industrie du développement, que nous avons adaptée à des fins de sécurité. Qui plus est, nous avons également démontré qu'il est possible de vérifier et de garantir que les paramètres de sécurité des systèmes de gestion de base de données sont conformes aux attentes et que ces derniers n'ont pas été altérés par inadvertance dans le temps.

Finalement, nous souhaitons souligner l'importance de la rigueur méthodologique et de la présentation des résultats dans la mise en œuvre de notre démarche. Nous croyons que les tests unitaires utilisés dans le cadre d'une approche préventive contre les injections SQL sont une méthode valable. Nous reconnaissons que cela peut représenter une charge de travail supplémentaire pour les équipes de développement, néanmoins, pour croyons que les prochains travaux de recherche devraient s'orienter à résoudre les lacunes identifiées en offrant des perspectives complémentaires à la sécurisation des systèmes d'information. Dans un monde technologique qui est sans cesse menacé par des criminels et des organisations malveillantes, il nous parait indispensable de chercher des pistes d'amélioration de nos pratiques de développement logiciel.

CHAPITRE VI
CONCLUSION

Ce mémoire de maîtrise a examiné l'utilisation des tests unitaires comme moyen de prévention contre les vulnérabilités liées aux injections SQL. Nous avons pu démontrer qu'en effectuant des tests unitaires, il est possible d'interrompre des tentatives d'attaques dès que le code source contient des mesures de contrôles qui lui permettent de vérifier et de traiter des intrants afin de s'assurer que ces derniers sont exempts de menaces. Également, nous avons pu démontrer qu'il est possible de valider la paramétrisation des systèmes de gestion de base de données et ainsi s'assurer que celle-ci est conforme aux attentes de sécurité.

L'étude a mis en lumière l'importance de l'intégration des tests unitaires dans les processus de développement de logiciel, en mettant l'accent sur une nouvelle méthodologie de protection. En adoptant une approche proactive, les tests unitaires peuvent non seulement identifier les faiblesses dès les premières phases du développement, mais aussi contribuer à les corriger de manière adéquate tout en sensibilisant les équipes de développement de logiciel aux concepts de développement sécuritaire.

D'autant plus qu'il serait envisageable d'élargir notre approche pour inclure d'autres types d'attaques impliquant des injections de code, susceptibles d'exploiter des vulnérabilités liées au traitement de données invalides.

Les résultats obtenus ont démontré que l'adoption de tests unitaires appropriés peut considérablement renforcer la sécurité, en réduisant les risques d'exploitation des failles liées aux injections SQL.

Cette conclusion offre également un retour sur quelques avantages tangibles de notre approche, offrant aux équipes de développement un moyen efficace de renforcer la résilience des logiciels contre les attaques qui exploitent les injections SQL :

1. La validité de l'approche : La recherche a permis de valider nos hypothèses quant à la validité des tests unitaires dans la prévention des attaques aux injections SQL. Nous sommes d'avis qu'il est trop tôt pour affirmer que notre approche peut être mise en œuvre. Néanmoins, nous croyons que les scénarios de tests que nous avons créés, en plus des tests de vérification de la conformité du système de gestion des bases de données, démontrent qu'il est possible de couvrir plusieurs aspects dans cette première itération de la démarche.
2. La détection précoce : La recherche a permis de développer des techniques de tests unitaires permettant d'identifier tôt dans le processus, des vulnérabilités relatives aux injections SQL. Grâce aux tests, des scénarios d'attaques peuvent être simulés dans des conditions contrôlées permettant ainsi aux développeurs d'identifier et de corriger les faiblesses avant même que le code ne soit déployé.
3. La sensibilisation : Ce projet de recherche pourrait contribuer à améliorer les connaissances des développeurs de logiciel, en les sensibilisant et en les soutenant dans leur cheminement.
4. Réduction des coûts : Nous savons que les tests unitaires permettent de réduire les coûts associés aux corrections. La détection précoce des vulnérabilités contribue à cette réduction. Elle permet d'éviter également les coûts d'une compromission pouvant être engendrée, si l'une ou l'autre des vulnérabilités non identifiées était exploitée. Il convient de noter que les corrections apportées dans les premières phases du développement sont moins coûteuses que celles effectuées une fois que le logiciel est en production.

5. Amélioration de la productivité des développeurs : En fournissant un retour d'information rapide sur la sécurité du code, les tests unitaires permettent aux développeurs de travailler plus efficacement et de consacrer moins de temps au débogage.
6. Réduction des risques de régression : Les tests unitaires s'assurent que les modifications ultérieures du code n'introduisent pas de nouvelles failles. Ce qui réduit considérablement les risques de régression.

Nous sommes conscients qu'il reste encore beaucoup de travail avant de prétendre que les tests unitaires, dans un contexte de prévention contre les injections SQL, apportent un nouveau paradigme de protection valable au niveau de la pratique de sécurité. Pour atteindre un modèle de protection efficace et applicable dans le quotidien des équipes de développement, nous devrons généraliser la méthode et faire en sorte que celle-ci soit applicable dans des contextes réels de développement logiciel.

Finalement, nous croyons que ce projet de recherche a apporté une contribution significative en termes de pratiques sur la sécurité du développement logiciel, offrant une avancée substantielle au niveau de la détection et la prévention des vulnérabilités. Cette contribution s'inscrit dans une démarche proactive visant à renforcer la robustesse des applications et à réduire les risques de failles de sécurité relatives aux injections SQL. Par ailleurs, quelques semaines avant la remise de ce mémoire, le *Federal Bureau of Investigation* et le *Cybersecurity and Infrastructure Security Agency* des États-Unis publiaient une alerte soulignant à la communauté technologique qu'il est inacceptable que des vulnérabilités de type injection SQL, connue depuis plusieurs décennies, soient encore à l'origine de compromission des données. Le *FBI* et le *CISA* ont rappelé les attaques de type *chaîne d'approvisionnement* (*Supply Chain Attacks*) de 2023 qui ont exploité une vulnérabilité *Zero Day* (Bilge & Dumitraş, 2012) à partir d'une injection SQL. Selon ces derniers, les fournisseurs doivent désormais prioriser l'approche *Secure-by-Design*, signifiant que les éditeurs doivent concevoir leurs logiciels de manière à intégrer la sécurité dans leurs produits (Jones, 2024).

BIBLIOGRAPHIE

(1986). IEEE Standard for Software Unit Testing. *ANSI/IEEE Std 1008-1987*, pp. 1–28. doi: 10.1109/IEEESTD.1986.81001

Aljabri, M., Aldossary, M., Al-Homeed, N., Alhetelah, B., Althubiany, M., Alotaibi, O. & Alsaqer, S. (2022). Testing and Exploiting Tools to Improve OWASP Top Ten Security Vulnerabilities Detection. Dans *2022 14th International Conference on Computational Intelligence and Communication Networks (CICN)*, pp. 797–803. doi: 10.1109/CICN56167.2022.10008360

Alwashali, A. A. M. A., Abd Rahman, N. A. & Ismail, N. (2021). A survey of ransomware as a service (RaaS) and methods to mitigate the attack. Dans *2021 14th International Conference on Developments in eSystems Engineering (DeSE)*, pp. 92–96. IEEE.

Baklizi, M., Atoum, I., Abdullah, N., Al-Wesabi, O. A., Otoom, A. A. & Hasan, M. A.-S. (2022). A technical review of SQL injection tools and methods : a case study of SQLMap. *International Journal of Intelligent Systems and Applications in Engineering*, *10*(3), 75–85.

Bilge, L. & Dumitraş, T. (2012). Before we knew it : an empirical study of zero-day attacks in the real world. Dans *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 833–844.

Bland, M. (2014). Goto Fail, Heartbleed, and Unit Testing Culture. Repéré à https://martinfowler.com/articles/testing-culture.html

Canada, S. (2022, octobre). *L'incidence du cybercrime sur les entreprises canadiennes, 2021*. Repéré à https://www150.statcan.gc.ca/n1/daily-quotidien/221018/dq221018b-fra.htm

Conklin, L. & Robinson, G. (2017). *OWASP Code Review Guide 2.0*. Repéré à https://owasp.org/www-project-code-review-guide/

Daka, E. & Fraser, G. (2014). A Survey on Unit Testing Practices and Problems. p. 201–211. doi: 10.1109/ISSRE.2014.11

Deckard, J. (2005). *Buffer overflow attacks : detect, exploit, prevent*. Elsevier.

Dencheva, L. (2022). *Comparative analysis of Static application security testing (SAST) and Dynamic application security testing (DAST) by using open-source web application penetration testing tools*.

Foggon, D. (2006). Stored Procedures. *Beginning ASP. NET 2.0 Databases : From Novice to Professional*, pp. 415–457.

Fowler, M. (2024, janvier). *Continuous Integration*. Repéré à https://martinfowler.com/articles/continuousIntegration.html

Galluccio, E., Caselli, E. & Lombari, G. (2020). *SQL injection strategies : practical techniques to secure old vulnerabilities against modern attacks* (1st edition). United Kingdom: Packt Publishing.

Gartner (2021, May). *Forecasts Worldwide Security and Risk Management Spending to Exceed $150 Billion in 2021*. Repéré à https://www.gartner.com/en/newsroom/press-releases/2021-05-17-gartner-forecasts-worldwide-security-and-risk-managem

Gatlan, S. (2024, janvier). *Mandiant's X account hacked by crypto Drainer-as-a-Service gang*. Repéré à https://www.bleepingcomputer.com/news/security/mandiants-x-account-hacked-by-crypto-drainer-as-a-service-gang/

Gomaa, Y., El Aziz Ahmed, A., Mahmood, M. A. & Hefny, H. (2015, Nov). Survey on securing a querying process by blocking SQL injection. Dans *2015 Third World Conference on Complex Systems (WCCS)*, p. 1–7. doi: 10.1109/ICoCS.2015.7483271

Gonzalez, D. (2021). *The State of Practice for Security Unit Testing : Towards Data Driven Strategies to Shift Security into Developer's Automated Testing Workflflows*. (Phd thesis). Rochester Institute of Technology, New York. Repéré à https://scholarworks.rit.edu/cgi/viewcontent.cgi?article=12059&context=theses

Gonzalez, D., Perez, P. P. & Mirakhorli, M. (2021). Barriers to Shift-Left Security : The Unique Pain Points of Writing Automated Tests Involving Security Controls. doi: 10.1145/3475716.3475786

Gren, L. & Antinyan, V. (2017). On the Relation Between Unit Testing and Code Quality. p. 52–56. arXiv :1904.04748 [cs], doi: 10.1109/SEAA.2017.36

Guidance, S. (2017, juillet). *Security Guidance for Critical Areas of Focus in Cloud Computing v4.0*. Repéré à https://cloudsecurityalliance.org/artifacts/security-guidance-v4

Halfond, W. G., Viegas, J., Orso, A. *et al.* (2006). A classification of SQL-injection attacks and countermeasures. Dans *Proceedings of the IEEE international symposium on secure software engineering*, Vol. 1, pp. 13–15. IEEE Piscataway, NJ.

Hydara, I., Sultan, A. B. M., Zulzalil, H. & Admodisastro, N. (2015). Current state of research on cross-site scripting (XSS)–A systematic literature review. *Information and Software Technology*, *58*, 170–186.

Hyslip, T. (2017). SQL Injection : The Longest Running Sequel in Programming History. *The Journal of Digital Forensics, Security and Law*. doi: 10.15394/jdfsl.2017.1475

IBM (2023). *Cost of a data breach 2022*. Repéré à https://www.ibm.com/reports/data-breach

Jones, C. (2024). Uncle Sam's had it up to here with "unforgivable" SQL injection flaws. *The Register*. Repéré à https://www.theregister.com/AMP/2024/03/26/fbi_cisa_sql_injection/

Khorikov, V. (2020). *Unit testing : principles, practices, and patterns*. Shelter Island, NY: Manning.

Kolawa, A. (2005). Reducing software security vulnerabilities through unit testing. *Military and Aerospace Electronics*, p. 41–42.

Leloudas, P. (2023). *Introduction to Software Testing : A Practical Guide to Testing, Design, Automation, and Execution*. Berkeley, CA: Apress. doi: 10.1007/978-1-4842-9514-4

Manky, D. (2013). Cybercrime as a service : a very modern business. *Computer Fraud & Security*, *2013*(6), 9–13.

Mohammadi, M., Chu, B. & Lipford, H. R. (2018). Detecting Cross-Site Scripting Vulnerabilities through Automated Unit Testing. *arXiv :1804.00755 [cs]*. arXiv : 1804.00755, Repéré à http://arxiv.org/abs/1804.00755

Mohammadi, M., Chu, B., Lipford, H. R. & Murphy-Hill, E. (2016). Automatic web security unit testing : XSS vulnerability detection. p. 78–84. doi: 10.1145/2896921.2896929

Özkan, S. (2017). Cve details. *Retrieved*, *16*, 2017.

O'Regan, G. (2019). *Concise Guide to Software Testing*. Undergraduate Topics in Computer Science. Cham: Springer International Publishing. doi: 10.1007/978-3-030-28494-7

Pitchford, M. (2021). The 'Shift Left' Principle. p. 18–21. doi: 10.12968/S0047-9624(22)60234-7

Poulsen, K. (2002). Guesswork Plagues Web Hole Reporting. Repéré à https://web.archive.org/web/20120709141229/http://www.securityfocus.com/news/346

Ray, D. & Ligatti, J. (2014). Defining Injection Attacks. p. 425–441. doi: 10.1007/978-3-319-13257-0_26

Saad, E. & Mitchell, R. (2020). *OWASP - The Web Security Testing Guide*. Repéré à https://owasp.org/www-project-web-security-testing-guide/