2013-ENST-0063

EDITE - ED 130

Doctorat ParisTech

THÈSE

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité « Informatique et Réseaux »

présentée et soutenue publiquement par

Gabriel SERME

le 05 Novembre 2013

La modularisation de la sécurité informatique dans les systèmes distribués

Jury

M. Mario SÜDHOLT, Professeur, Département Informatique, École des Mines de Nantes	Président
M. Frédéric CUPPENS, Professeur, Télécom Bretagne	Rapporteur
M. Ludovic MÉ, Enseignant-chercheur - HDR, Supelec	Rapporteur
Mme Mireille BLAY-FORNARINO, Professeur, Laboratoire I3S, CNRS - UNS	Examinatrice
M. Refik MOLVA, Professeur, EURECOM	Examinateur
M. Yves ROUDIER, Maître de Conférences, Eurecom	Directeur de thèse
M. Anderson SANTANA de OLIVEIRA, Docteur, SAP Labs	Invité

THÈSE

TELECOM ParisTech
école de l'Institut Mines-Télécom - membre de ParisTech
46 rue Barrault 75013 Paris - (+33) 1 45 81 77 77 - www.telecom-paristech.fr

Acknowledgments

I would like to first acknowledge my academic supervisor, Yves Roudier, for his guidance, support as well as time and advice throughout the thesis. He is the one together with Jean-Christophe Pazzaglia from SAP Research who enable me to work on my thesis.

A thesis is the result of many years of work that could not come out without support from many people. I've been lucky to meet so many great people coming from different fields, and from different environments. Fulfill such a journey requires a balance in professional activities and spare time, although the limit between both tends to blur over time.

My first thought goes to Cedric Ulmer, Paul and Azzedine. I started a thesis after working and discussing with them, and they constantly provided me valuable advices. I owe my deepest gratitude to Anderson, as my thesis would also not have been possible without his daily support.

I am indebted to my many of my colleagues to support me. I would like to mention, although the list is not exhaustive, Wihem, Cedric, Laurent, Jakub, Gilles, and all my colleagues from SAP Research and Eurecom. I am grateful also to all PhDs': Sabir, Giancarlo, Theodoor, Joern, Corentin, Matteo, Samuel, Ahmad and Mehdi. We had great time with all the students and apprentices from the Nemea community. A special mention goes to Julien, Peng, Yann who worked as intern for our project, and a big thank you to Olivier and Aurelien.

A personal attention to Sylvine and Gwenaelle.

Finally, it is a pleasure to thank those I bothered with my thesis: my roommates Stephane and Francois. Another thank you for the numerous people I met at SAP (Sophia-Antipolis but also worldwide) and at Eurecom.

I'm thankful to my family (especially mother, father, brother, brother, brother) and to all my friends.

Last, but definitely not least, I would like to thank Julie for all these years at my side. She handled all situations, and supported me whether I was in a bad mood, anxious, or nervous. She has been comprehensive and I congratulate her for still supporting me.

Contents

Résumé **1**

1 Introduction **16**

1.1 Background . . . 16
1.2 Overview . . . 17
1.3 Problem Statement . . . 18
1.4 Contributions . . . 19
1.5 Organization of the thesis . . . 19

2 Modularization of cross-cutting concerns **20**

2.1 Software Engineering . . . 20
2.2 Aspect-Oriented Software Development . . . 23
2.3 Security Implication in the Software Development Life Cycle . . . 31
2.4 Security Requirements . . . 32
2.5 Security and aspects . . . 33
2.6 Service Oriented Architecture . . . 41

I Modularization of defensive security **45**

3 Vulnerability remediation with modular patches **48**

3.1 Introduction . . . 48
3.2 Agile management of vulnerabilities . . . 51
3.3 A flexible architecture . . . 52
3.4 Static analysis process . . . 54
3.5 Assisted remediation with a security aspect library . . . 58
3.6 Evaluation . . . 64
3.7 Related work . . . 70
3.8 Summary . . . 72

4 Automation of input validation verification in application **74**

4.1 Introduction . . . 74
4.2 Aspect-based modularization with enhanced data-types . . . 77
4.3 Use case . . . 83

4.4 Validation 86
4.5 Related work 90
4.6 Language approach for security modularization 93

II Modularization of constructive security 95

5 Service framework modularization for message-based security 97
5.1 Introduction 97
5.2 Motivation 98
5.3 REST Security 104
5.4 Evaluation of REST security protocol 111
5.5 Related Work 115
5.6 Summary 116

6 Modularization of privacy in cloud platform around persistance layer 118
6.1 Introduction 119
6.2 Privacy-Aware Applications in the Cloud 121
6.3 Privacy Enhanced Application Programming 123
6.4 Related Work 128
6.5 Summary 130

III Conclusion and perspectives 131

7 Conclusion 132

8 Perspectives: towards a cross-layer and parametric aspect system 134
8.1 Motivation 134
8.2 Requirements 139
8.3 Aspect System and Language 140
8.4 Related work 142
8.5 Conclusion 144

Bibliography 145

List of Figures

2.1 Programming paradigms categorization presented in [USB09] 22
2.2 cross-cutting concern visualization with the AJDT eclipse plugin for a deliberate vulnerable application 25
2.3 Application execution output result 28
2.4 Class diagram of the example. The aspect and the base application are in black. The weaving addition are in red. 29
2.5 Software Development Life-Cycle 32
2.6 WS-Security standards 43

3.1 Vulnerability remediation process. The two first blocks correspond to the static analysis component. The two last blocks correspond to the remediation component. The last one corresponds to assisted processing component 51
3.2 Architecture 53
3.3 AST view example of a class 55
3.4 Static Analysis Activity Diagram 56
3.5 Code Analysis result 58
3.6 Gathering context for vulnerability protection 60
3.7 Example of result with code pro analytix 67
3.8 Example of result with lapse 68
3.9 Example of result with bigbro 69

4.1 Solution components 78
4.2 Components and ideal roles 80
4.3 Internal activity diagram 82

5.1 OAuth generic flow 101
5.2 Domains of the scenario with hierarchical reference monitors per domain . . . 102
5.3 Overhead of SOAP messages compared to REST. For each scenario and security, the REST size represents the base 100 113
5.4 Average processing time comparison for the different scenarios 114

6.1 Privacy aware PaaS components 120
6.2 Excerpt of a PPL policy rule 123
6.3 Excerpt of a PPL policy condition 123

6.4 JPA entity class annotation indicating persistency of private information 124
6.5 Annotating private data usage class with PII meta-information 125
6.6 Enforcement components 126
6.7 SQL transformation example 128

8.1 Architecture of the solution 140
8.2 Application layers 141

List of Tables

3.1 List of detected vulnerabilities with potential origin and potential remediation. . 59
3.2 Comparison of static analysis tools on several projects. It presents the detected vulnerabilities for the different categories when applicable. SLOC means Source Line of Codes. 66

4.1 Number of vulnerabilities detected by the arachni (arach.) and w3af web application security scanners with a black-box approach on original and protected wavsep application. 90

5.1 REST security protocol headers 106
5.2 Benchmark environment 112

8.1 Extensive list of source origins with their potential actors 142
8.2 Constructs for the language 143

Résumé

Intégrer les problématiques de sécurité au cycle de développement logiciel représente encore un défi à l'heure actuelle, notamment dans les logiciels distribués. La sécurité informatique requiert des connaissances et un savoir-faire particulier, ce qui implique une collaboration étroite entre les experts en sécurité et les autres acteurs impliqués. La programmation à objets ou à base de composants est communément employée pour permettre de telles collaborations et améliorer la mise à l'échelle et la maintenance de briques logicielles. Malheureusement, ces styles de programmation s'appliquent mal à la sécurité, qui est un problème transverse brisant la modularité des objets ou des composants. Nous présentons dans cette thèse plusieurs techniques de modularisation pour résoudre ce problème. Nous proposons tout d'abord l'utilisation de la programmation par aspect pour appliquer de manière automatique et systématique des techniques de programmation sécurisée et ainsi réduire le nombre de vulnérabilités d'une application. Notre approche se focalise sur l'introduction de vérifications de sécurité dans le code pour se protéger d'attaques comme les manipulations de données en entrée. Nous nous intéressons ensuite à l'automatisation de la mise en application de politiques de sécurité par des techniques de programmation. Nous avons par exemple automatisé l'application de règles de contrôle d'accès fines et distribuées dans des web services par l'instrumentation des mécanismes d'orchestration de la plate-forme. Nous avons aussi proposé des mécanismes permettant l'introduction d'un filtrage des données à caractère privée par le tissage d'aspects assisté par un expert en sécurité. Pour terminer, nous proposons un nouveau type de point d'insertion d'aspects (pointcut) centré sur le flot d'information dans un logiciel distribué et permettant d'unifier l'implantation de nos techniques de modularisation de la sécurité.

Introduction

La thèse apporte un point de vue industriel à la modularisation des propriétés de sécurité au niveau de la programmation d'applications. La sécurité est omniprésente et implique des mesures rigoureuses tout au long du cycle de développement d'un logiciel. Il s'agit de spécifier correctement et mettre en œuvre des propriétés de sécurité de manière cohérente. La sécurité est généralement considérée comme une préoccupation non fonctionnelle et transversale, indépendante des préoccupations métiers de l'application. Mais la sécurité revêt différentes facettes et peut avoir des ramifications fortes avec l'application, ce qui rend difficile la modularisation.

L'évolution rapide des applications d'entreprise apporte une nouvelle dimension à la sécurité. Au cours de la dernière décennie, les applications d'entreprise ont évolué vers plus de connectivité. Par conséquent, elles exposent leurs données à travers plusieurs canaux de communications tels que des applications web, des applications orientées services, des applications mobiles, ainsi que des applications dites dans les nuages (cloud computing). Cette connectivité permet une plus grande flexibilité pour créer de nouveaux marchés, de nouveaux produits, et pour favoriser la collaboration inter-partenaire. Et ce, malgré la complexité toujours plus poussée des systèmes d'informations. Mais cette connectivité apporte aussi une exposition supplémentaire aux données d'entreprise, ce qui rend encore plus complexe la gestion de la sécurité.

Des évènements récents présentent des attaques à fort impact. Les attaques surviennent dans tout type d'environnement: grandes entreprises, organisations, particulier, mais aussi et de plus en plus fréquemment des institutions et des pays pour les attaques les plus sophistiquées. Les attaques ont tendance à provenir de groupes organisés. Les groupes ont à leur disposition de nombreux outils pour pénétrer un système et obtenir les informations ou les ressources recherchées. Ces techniques, comme de l'ingénierie sociale, de l'exploitation de failles informatiques, mais aussi le vol de matériel ou de ressource numérique forment des attaques contre lesquelles il est difficile de se prémunir. Les entreprises ont besoin d'audits et de suivis réguliers dans la gestion de leurs ressources pour détecter dès que possible des intrusions afin de réagir correctement.

Dans cette thèse, nous couvrons un type d'attaque parmi celles énoncées. Nous nous intéressons aux attaques qui utilisent les manquements au moment de la programmation. Ces attaques, bien que moins structurées, n'en demeurent pas moins complexes et omniprésente. Il existe des outils pour automatiser un certain nombre d'attaques qui peuvent ainsi être contrôlées par des individus ou des groupes qui sont à la recherche de gloire, de récompenses, ou simplement de connaissances. Les menaces peuvent généralement être atténuées par le respect de politiques de sécurité ainsi que par l'application de bonnes pratiques de programmation. Le problème dans ce cas est de proposer un ensemble d'outils et de méthodes pour assurer l'application correcte des politiques et d'appliquer les bonnes pratiques de programmation lors du développement des applications. Ces pratiques sont fortement dépendantes de l'environnement utilisé. Ainsi, en fonction du langage de programmation choisi, du système d'exploitation utilisé, du serveur d'application retenu, ainsi que des différents protocoles de communication adoptés, les bonnes pratiques de programmation auront des spécificités. Chacun de ces éléments doit être pris en considération pour garantir l'absence de vulnérabilités et la bonne mise en œuvre des propriétés de sécurité.

Nous avons commencé avec le constat que les applications sont de plus en plus omniprésente dans notre environnement, allant de périphériques embarqués à des applications distribuées. Les différents types de logiciels, quels que soient leur environnement, doivent respecter des contraintes de qualité. L'industrie se concentre d'abord sur la production d'application offrant de la valeur ajoutée dans un domaine fonctionnel spécifique. Il peut s'agir d'applications pour aider à la décision, faciliter la gestion de ressources, mais il peut aussi s'agir de produits pour faciliter le développement de nouvelles applications. Chacun de ces produits possède des spécifications pour décrire les besoins fonctionnels qui sont le cœur de métier de ces logiciels. En plus de ces besoins fonctionnels, les éditeurs de logiciels doivent respecter plusieurs contraintes lorsqu'ils architecturent et développent ces solutions: les politiques internes, le droit juridique, les besoins non fonctionnels, *etc.* Ces besoins non fonctionnels sont des spécifications qui ne portent pas sur les besoins directs de l'application. Ils sont présents pour assurer la bonne exécution de l'application dans son environnement. En général, les besoins non fonctionnels sont tels que l'utilisabilité, l'intégrité, la fiabilité, la performance, *etc.* Une autre préoccupation qui revient souvent dans la littérature comme un besoin non fonctionnel est la sécurité dont nous allons traiter dans cette dissertation.

Nous allons voir dans les chapitres suivants que la sécurité a plusieurs facettes. Elle est généralement considérée comme une préoccupation non fonctionnelle qui est entrecroisé dans l'application. Plusieurs outils peuvent aider à introduire de la sécurité dans l'application (tels que des bibliothèques tierces, le support par l'environnement d'exécution de primitives, une protection en provenance du système d'exploitation, des filtres applicatifs, *etc.*). En réalité, la sécurité a des ramifications plus profondes. Les propriétés de sécurités devant être introduite au sein de l'application couvrent des besoins fonctionnels, mais aussi des besoins non fonctionnels. Les entreprises doivent développer des produits avec toutes les exigences de sécurités correctement implémentées et orchestrées tout en respectant les différentes contraintes mentionnées ci-dessus.

Problème traité

Le problème du développement d'applications sécurisé a évolué au fil des années. Le problème est connu et un certain nombre de solutions ont émergé, mais on observe toujours des vulnérabilités introduites par les développeurs au sein des applications. Nous abordons le problème de l'intégration non invasive et systématique de la sécurité au sein des applications lors de la phase de développement, tout en respectant les contraintes de temps induit par les contraintes industrielles lors de l'élaboration des logiciels. Les exigences de sécurité proviennent des politiques de sécurité en place pour respecter les normes de qualité des logiciels, mais aussi de différentes réglementations. Les propriétés de sécurité de ces systèmes affectent différentes couches qui sont fortement interconnectés. La sécurité d'un système est affectée par plusieurs décisions au cours du développement des logiciels. Il commence à partir de la définition des besoins de sécurité. En supposant que les exigences sont correctement posées et décrites, les développeurs reçoivent une liste de spécification à respecter, tout en développant. Ils doivent maintenir une qualité lors de l'introduction des mécanismes de sécurité répondant aux exigences de sécurité. Malgré la littérature apportée par la discipline sur la manière d'intégrer au mieux les mécanismes de sécurité, les développeurs introduisent encore des vulnérabilités qui affectent les systèmes.

L'introduction de failles de sécurité par le développeur dépend de plusieurs facteurs: le manque de connaissances sur les mécanismes de sécurité, la mauvaise interprétation des spécifications ou de l'architecture du logiciel, la mauvaise configuration des frameworks et des bibliothèques, ou même quelques défauts de refactorisation non détecté avant la mise en production. Il existe la possibilité d'avoir un développeur qui introduit délibérément des portes dérobées ou des défauts dans l'application, même si cela reste rare. Il est important de noter que le coût pour corriger une vulnérabilité de sécurité, s'il est pris comme un défaut de logiciel, croît de manière exponentielle au fur et à mesure de l'avancement de l'application (développement, tests, production, *etc.*). L'exposition des vulnérabilités logicielles devient effective lorsqu'un défaut est mis en production, et que ce défaut peut entraîner une exploitation de l'application à l'insu de ce qui a été spécifié. En d'autre termes, il s'agit de faiblesses qui restent présentent alors que l'application est censée répondre à tous les besoins qui ont été spécifiés.

Afin de limiter le nombre de défaut introduit involontairement, nous proposons d'accompagner les développeurs en leur donnant des méthodes et des outils. Nous avons défini un ensemble de besoins de sécurité, provenant d'un sous-ensemble de politiques internes que l'on peut retrouver en entreprise, mais également provenant d'exigences que nous avons observés dans les bonnes pratiques de programmation. Les exigences visent principalement les vulnérabilités des applications web, bien que l'on puisse généraliser le problème à tous les défauts que l'on rencontre dans le cadre de systèmes distribués. Dans la première partie de la thèse, nous limitons la liste des exigences aux défauts qui sont introduits lors du développement du code applicatif. Nous sommes intéressés par la détection et la prévention de toutes les manipulations d'entrée: cross-site scripting, injection SQL, manipulation de chemin d'accès, injection de commandes, pollution de paramètre HTTP, et tout type d'injection qui en découle. Dans la deuxième partie, nous proposons de simplifier l'ajout de propriétés de sécurités que nous appelons "constructive". Ainsi, nous fournissions des outils pour que les développeurs utilisent de mécanismes de sécurités, au niveau de l'environnement d'exécution ou au niveau de la plateforme. Les contributions gèrent des propriétés telles que la responsabilité, ce qui implique l'ordonnancement de plusieurs mécanismes différents, ainsi que de la gestion de la protection de la vie privée et de la communication dans les systèmes distribués.

La modularisation permet aux entreprises de proposer des logiciels qui sont conformes aux différentes régulations en vigueur dans les pays où sont commercialisés ces logiciels, en appliquant les contraintes de sécurité de manière tardive. Par exemple, la gestion de la confidentialité des données personnelles est différente d'un pays à l'autre. Bien que la solution que nous développons dans ce qui suit puisse être appliquée dans plusieurs contextes, nous les avons mis en œuvre dans un contexte de systèmes distribués. Nous ciblons plus particulièrement les systèmes interconnectés dans un style architectural qui est souvent désigné comme une architecture orientée services (SOA). Nous apportons aussi des solutions pour des plateformes de cloud computing, ce qui implique que nous devons non seulement cibler les applications, mais aussi les infrastructures et la plateforme.

La thèse discute donc du développement de logiciels sécurisés dans les environnements distribués. Nous avons développé des techniques pour améliorer la flexibilité dans la gestion des propriétés de sécurité au moment du développement applicatif. Nous appliquons le concept de préoccupation transversale, principalement promue par le paradigme de développement à partir d'aspect (AOP). Ce paradigme permet d'améliorer la flexibilité à la fois dans la définition des propriétés de sécurité, mais aussi dans son exécution. Nous différons la mise en œuvre concrète des propriétés de sécurité dans l'application. Nous commençons par collecter les points impactant au niveau du code source de l'application, puis nous injectons les comportement de sécurité pour obtenir des logiciels qui respectent à la fois les bonnes pratiques de programmation sécurisés, mais aussi les différents besoins fonctionnels de sécurités qui peuvent survenir (gestion de l'authentification et des autorisations, gestion de la vie privée, gestion de la confidentialité et de l'intégrité des données, *etc.*).

Dans la suite du résumé, nous décrivons plus en détail les différentes contributions qui sont exposées dans la thèse, afin de répondre à la modularisation des propriétés de sécurité en milieu industriel: modularisation des techniques pour de la sécurité constructive, et modularisation des techniques pour de la sécurité défensive. Nous introduisons une séparation dans les catégories de sécurité afin de distinguer la sécurité qui répond aux besoins fonctionnels tels que la vie privée, la confidentialité, *etc.* de la sécurité qui limite l'exploitation de vulnérabilités logicielles. Les contributions proposent de faciliter la gestion des propriétés de sécurité tout au long du cycle de vie de sécurité avec un accent tout particulier sur la phase de développement. La prochaine section présente deux contributions relatives à la modularisation de la sécurité défensive. Ces deux contributions ont la même finalité, c'est à dire modulariser du code de sécurité qui limite la possibilité d'exploiter une application, mais dont la méthode de protection diffère. Ensuite, la section suivante présente deux autres contributions relatives à la modularisation de la sécurité constructive. Ce sont deux approches différentes, qui présentent deux types de propriétés dans des environnements différents que l'on peut modulariser pour en simplifier l'intégration par les développeurs. Enfin, une dernière section présente les conclusions que l'on retire de cette thèse, qui ouvre sur une perspective d'unification des approches présentées.

Modularisation de la sécurité défensive

La modularisation de la sécurité défensive peut être simplifiée à l'analyse et l'application des bonnes pratiques de sécurité au niveau du code applicatif. Les bonnes pratiques de sécurité sont un ensemble de méthodes, de règles, de processus, de concepts et théories qui doivent être bien partagés entre les différents acteurs évoluant autour du développement du logiciel. Les personnes impliquées dans la définition et la mise en œuvre d'un tel système doivent partager la même vision globale en termes de sécurité, et aussi se former afin de détecter et réagir aux nouvelles menaces. Il existe plusieurs sources pour partager ces connaissances, avec des portées différentes. Les gouvernements et institutions publient des recommandations pour maintenir une sécurité des systèmes d'information. Ils éditent aussi les lignes directrices en matière de sécurité informatique. Par exemple, le gouvernement français a une structure dédiée qui propose des documents, des analyses et des informations autour de la sécurité de l'information. Le gouvernement allemand pour sa part propose une structure similaire en qualité d'Office fédéral de la sécurité de l'information. L'Europe dispose aussi d'une agence afin d'améliorer la transmission d'information sur cette problématique entre les différentes nations. L'agence publie aussi des guides pour sensibiliser le grand public aux problématiques de sécurité, à destination des citoyens, des entreprises, ou encore du secteur public, *etc.*. Un institut renommé est le "National Institute of Standards and Technology" (NIST). Il fournit des conseils dans plusieurs domaines de l'ingénierie de la sécurité de l'information. Des institutions indépendantes existent aussi qui élaborent des guides de bonnes pratiques.

En plus de toutes ces institutions, il y a des consortiums qui se structurent avec un objectif commun pour partager des connaissances. Ils centralisent les meilleures pratiques dans des domaines spécifiques. Dans le domaine de la sécurité, il y a des groupes tels que le CERT qui décrit des approches pour une meilleure sécurité du système de l'information: des outils et des techniques pour évaluer et mesurer les vulnérabilités. Le CERT a d'abord été une équipe de l'Université Carnegie Mellon, et forme maintenant une équipe dédiée à la réponse des menaces sur internet et dans le monde de l'informatique. OWASP est un projet open-source pour la sécurité des applications web. C'est une communauté pour les sociétés, les chercheurs, et les particuliers ayant plusieurs événements régionaux pour partager les connaissances et comprendre les dernières avancées en termes de menaces. Son champ d'application se limite à des applications web et à service. Une autre organisation diffuse les bonnes pratiques dans l'environnement du cloud computing. Ainsi, ils publient des documents spécifiques à la sécurité de l'informatique dans les nuages par exemple.

Bien que de nombreuses organisations éditent des guides pour la gestion de la sécurité de l'information, les failles de sécurité sont encore répandues dans les applications, et en particulier les applications web et dans les nuages. Ces types d'applications sont spécialement exposés à un public plus large avec une forte médiatisation des attaques. Elles interagissent avec les utilisateurs et recueillent un vaste ensemble de données liés à la vie privée. Développer ce type d'applications nécessite une attention particulière lors de l'édition des spécifications. Ensuite, il faut implémenter de manière rigoureuse les besoins de l'application. Il est nécessaire d'éliminer au maximum les vulnérabilités qui pourraient être introduite lors du développement avant la mise en production de l'application. Il existe plusieurs stratégies pour s'assurer que des applications fonctionnent correctement et sont sécurisées. L'environnement informatique est hétérogène et donne aux développeurs de nombreuses possibilités: frameworks, bibliothèques, langages de programmation, serveurs d'applications, *etc.* Ces éléments permettent de fournir de la flexibilité, performance et sécurité, bien qu'il soit parfois nécessaire de faire un compromis. Une application web devra par exemple vérifier les données en entrée, qui aura un impact sur la performance de l'application. La modularisation de la sécurité permet de savoir à quels endroits les vérifications sont effectuées, et ainsi avoir une meilleure vision des points de contrôle. Dans ce qui suit, nous présentons un premier travail où nous utilisons l'environnement de développement de l'utilisateur afin de détecter et de protéger les vulnérabilités communément rencontrées dans les applications web. Ensuite, nous proposons une autre approche qui généralise et rend plus souple l'automatisation de la validation d'entrée dans des applications web.

Contribution 1

Les failles de sécurité sont fréquemment rencontrées dans les applications en dépit de l'existence de méthodes pour les éviter depuis plusieurs années. Afin de détecter les failles de sécurité oubliées par les développeurs, des solutions complexes sont entreprises comme l'analyse statique de code source, souvent après la phase de développement. Le problème de ce type de solution une fois que le développement est terminé réside dans la décorrelation entre le contexte applicatif et le rapport d'état indiquant la localisation des problèmes dans le code: les personnes en charge de la correction ont toutes les difficultés pour comprendre l'architecture et les raisons qui ont mené à un code source en particulier. Ensuite, des vulnérabilités peuvent être trouvées, sans pour autant qu'une protection systématique soit appliquée.

Dans la discussion de la thèse, cette contribution introduit un plugin intégré à l'environnement de développement de l'utilisateur (dans ce cas, Eclipse) pour aider les développeurs à détecter et corriger des vulnérabilités au niveau applicatif à l'aide de la programmation orientée aspect, et ceci, le plus tôt possible dans le cycle de vie de développement du logiciel. La solution est une combinaison d'analyse statique au cours de la phase de développement suivi de la génération de code de protection. Nous utilisons l'interaction des développeurs avec l'outil intégré pour obtenir plus de connaissances sur l'état de l'application à un endroit donné. Ceci nous permet de modulariser le code de sécurité et ainsi avoir une meilleure vue d'ensemble sur les différents aspects de sécurité appliqués. Le résultat permet d'obtenir des logiciels plus robustes avant leur livraison en production, et surtout une flexibilité accrue par l'application d'aspects le plus tard possible au sein de l'application.

La manière de corriger plusieurs vulnérabilités de sécurité en utilisant une combinaison entre un analyseur statique qui aide les développeurs à détecter les vulnérabilités et une correction semi-automatique de ces résultats avec AOP présente plusieurs avantages. Il permet à plusieurs acteurs de respecter des contraintes complexes à mettre en œuvre en temps normal. Premièrement, les experts en sécurité sont en mesure de propager des mises à jour du code de protection aux équipes de développement. Les développeurs peuvent ainsi plus facilement détecter et corriger des bugs de sécurité. Les acteurs peuvent interagir étroitement pour décider des meilleures solutions à implémenter pour une situation donnée. Les développeurs bénéficient de cette approche en ayant un outil opérationnel déjà configuré pour leur environnement de développement. Ils peuvent se concentrer sur l'écriture de leur code fonctionnel et, de temps en temps, vérifier le respect des bonnes pratiques de programmation ainsi que l'absence de vulnérabilité dans leur code applicatif. Les codes de protection de sécurité sont souvent parsemés et surtout entremêlés au sein du code applicatif, ce qui tend à avoir des contrôles de sécurité répartis dans toute l'application. L'utilisation d'une solution unifiée dans la gestion de l'application des codes de protection permet d'avoir une vue d'ensemble. Ceci est plus efficace et permet une plus grande productivité des développeurs. L'automatisation de la détection et de l'application de code de protection permet une application plus large et cohérente de la sécurité dans toutes les applications. L'utilisation d'AOP facilite le déploiement et le changement de code de protection de sécurité. Enfin, le fait que le processus se fasse au cours de la phase de développement permet de détecter au plus tôt d'éventuels problèmes.

Le plugin Eclipse que nous avons développé lors de cette contribution permet, sous un angle de vue éducatif, une meilleure prise de conscience des problèmes de sécurité d'un point de vue du développeur. Il est important de noter que la correction apportée par le plugin ne couvre pas forcément toutes les problématiques liées aux besoins de sécurité. L'application peut se retrouver avec quelques vulnérabilités non corrigeable par notre approche. Notre approche permet de limiter les bugs de sécurité introduit au moment de l'écriture du code. Les éléments qui ne peuvent pas être vérifiés par notre solution se trouvent être des besoins de sécurité dont nous discutons plus loin dans la thèse: authentification, autorisations, *etc.*. En outre, nous encourageons les développeurs à creuser les rapports de vulnérabilités que nous fournissons: la correction automatique proposé est un premier pas, qu'il est possible d'améliorer. Nous voulons que les développeurs apprennent les bonnes pratiques pour avoir de nouveaux réflexes lors de l'écriture de futures applications.

Cette contribution propose une approche de la programmation pour modulariser de la sécurité. Nous commençons à partir du code source d'une application et essayons d'injecter du code de protection à des endroits appropriés. Cet outil intégré a plusieurs avantages mentionnés ci-dessus, mais il souffre de limitations dues à des décisions que nous avons prises. Par exemple, quand nous développons un outil comme le plug-in Eclipse, nous visons une plate-forme et un langage de programmation, ce qui limite volontairement le champ d'application. L'outil lui-même est un prototype que nous avons validé sur des projets en interne à SAP et que nous avons comparé par rapport aux logiciels commerciaux. Dans plusieurs cas, l'approche agile qui consiste à faire des vérifications régulières sur le code, puis d'appliquer les corrections des vulnérabilités conduit à une réduction des faux positifs et l'absence de faux négatifs. Ceci est permis car le moteur de détection de vulnérabilité est affiné au fur et à mesure des analyses. En outre, fournir une assistance intégrée pour corriger les vulnérabilités est novatrice et nous nous concentrons maintenant sur l'amélioration du code de protection ainsi que de l'interaction qu'il peut y avoir entre plusieurs aspects de sécurités.

Contribution 2

Dans cette seconde contribution, nous nous intéressons au problème de validation d'entrée systématique pour application web, afin d'apporter des contre-mesures efficaces à plusieurs types d'attaque par injection. La solution s'appuie sur des annotations java qui fournissent des méta-données concernant les paramètres d'entrée de l'application. Cette information est ensuite utilisée pour injecter automatiquement du code de validation dans l'application cible, en utilisant une approche orientée aspect. La solution permet de réduire les risques d'oubli et de maintenir la logique de sécurité indépendamment de la logique applicative. La différence avec le précédent chapitre repose principalement sur la méthode pour recueillir les points où injecter le code de sécurité. Alors que dans la solution précédente, nous avons utilisé un analyseur statique, nous utilisons maintenant des annotations dans le code source de l'application pour donner plus d'information sur le contexte.

Beaucoup d'applications web et de services web sont sujets à des vulnérabilités sur la validation des entrées. Des exemples bien connus de cette classe de vulnérabilités comprennent les XSS (Cross-Site Scripting), SQL injection, injection de commandes, *etc.*. Bien que les vulnérabilités sur la validation des entrées sont bien connues et ont été étudiées, les vulnérabilités sont parmi les plus répandues dans les classements. Plusieurs institutions listent ces problématiques, comme OWASP qui édite chaque année le classement des dix failles les plus critiques sur les applications web. Les vulnérabilités sur la validation des entrées ont une origine commune: une vérification incorrecte des données d'entrée fournis par les clients de l'application qui résulte en un mauvais état. Les attaques par injection, qui exploitent ces vulnérabilités, sont des attaques dans lesquelles un attaquant crée un jeu de donnée en entrée qui permet de contrôler à distance le comportement de l'application web pour ainsi exécuter du code non voulu, ou contrôler des pans de l'application. Ces attaques peuvent avoir des conséquences dévastatrices, allant de la fuite d'informations à une escalade de privilège dans laquelle l'attaquant peut prendre le contrôle complet du système attaqué.

Prévenir les attaques est une tâche complexe. Malgré la présence d'outils pour détecter les vulnérabilités, les failles sont encore très répandues dans les applications web et le nombre de vulnérabilités signalées continue de croître. En outre, la complexité de la majorité des attaques exploitant cette classe de vulnérabilités n'a pas réellement augmenté, ce qui indique que même les protections les plus basiques et donc largement documentées ne sont pas prises en compte.

Afin d'éviter ces vulnérabilités, chaque entrée lue par le programme doit subir un processus de validation et de nettoyage. Nous nous concentrons sur la validation d'entrée qui est essentiellement, le processus d'ajouter du sens à la donnée qui entre. Il faut s'assurer que les entrées respectent un ensemble de contraintes afin d'obtenir une entrée bien formé. Selon le type de données, des contrôles supplémentaires peuvent être nécessaires: ne contenir que des caractères autorisés, vérifier la longueur d'une chaîne de caractères afin qu'elle reste dans certaines limites, ou encore de valider qu'un nombre soit compris dans une fourchette prévue par les spécifications.

Une des raisons qui fait que la présence de nombreuses vulnérabilités de validation des entrées est toujours hautement critique est le fait que les techniques pour remédier à ces vulnérabilités reposent sur la bonne écriture du code de vérification par le développeur. Bien que de nombreux frameworks encadrent l'écriture du code pour limiter ces types de vulnérabilités et fournissent des bibliothèques contenant des fonctions de validation et désinfection, ceux-ci doivent encore être appelés explicitement à partir de la logique applicative afin de valider ou désinfecter les données en entrée. Cela a deux inconvénients: d'abord, les développeurs sont susceptible d'oublier d'écrire ces vérifications (ou ils ignorent le problème). Ensuite, il est difficile de maintenir, mettre à jour et faire évoluer la logique applicative indépendamment des codes de vérifications: l'appel aux fonctions de validation serait disséminé dans de nombreux modules de code, et complètement entremêlé avec le code métier. En outre, les fonctionnalités de validation intégrées dans les frameworks n'ont pas le degré de granularité nécessaire pour gérer la validation d'un grand nombre de types de données différents qu'une application est susceptible de gérer.

Nous présentons dans cette seconde contribution une méthode innovante et des outils pour limiter la principale cause de vulnérabilités. La première phase requiert que le développeur applicatif annote dans le code les différentes entrées du programme afin d'ajouter des informations plus précises sur son typage. Cette approche est légère et permet d'enrichir les informations de type concernant les entrées de l'application. Ensuite, des experts de sécurité peuvent écrire du code modulaire qui vérifie les données à partir du typage indiqué. Ainsi, les fonctions de validation sont modulaires et sont intégrés dans le code existant à l'aide de la programmation orientée aspect. Les principaux avantages de notre solution sont l'utilisation de ce type de programmation pour injecter le code sans que les développeurs applicatifs aient à apprendre un nouveau paradigme de programmation. Ensuite, nous obtenons un haut degré d'automatisation et permettront d'ajouter des vérifications de sécurité avec un effort mineur. Le développeur applicatif n'a pas non plus besoin d'écrire du code de sécurité et peut se concentrer sur son code métier. La solution est extensible: les développeurs applicatifs peuvent définir de nouveaux types de données spécifiques à leur environnement. Les aspects de validation peuvent être écrits par eux ou par des experts en sécurité. Enfin, l'application considère la sécurité et la validation des entrées dès la phase d'architecture, ce qui permet d'avoir plus de souplesse pour gérer le tout.

Modularisation de la sécurité constructive

La modularisation de la sécurité constructive peut être simplifiée à la modularisation des préoccupations de sécurité de l'entreprise. Ce sont des préoccupations qui sont spécifiés comme des besoins fonctionnels. Elles permettent à l'application de fonctionner correctement dans son environnement tout en respectant tous les besoins en sécurité. En général, les préoccupations à introduire au sein de l'application sont complexes, ce qui peut entraîner un développeur malinformé à introduire des bugs qui peuvent ensuite se transformer en vulnérabilité exploitable par des personnes extérieures. Les concepts autour de la sécurité informatique sont nombreux. Ainsi, il y a plusieurs types de sécurité qui peuvent être affectés. Les développeurs, afin de respecter les spécifications, doivent donc utiliser plusieurs mécanismes de sécurité qui doivent être correctement injectés dans l'application. Les propriétés sont de l'ordre de l'authentification et l'autorisation au sein de l'application. Cela va aussi plus loin comme la gestion de la vie privée, la confidentialité des données, l'intégrité des messages, *etc.* Ce sont des préoccupations qui sont souvent exposées à travers des politiques de sécurité.

Dans ce qui suit, nous présentons deux nouvelles contributions qui montrent une modularisation de ces préoccupations de sécurité en deux couches différentes de l'infrastructure utilisée dans les applications distribuées. La première contribution présente une modularisation au niveau d'une plateforme qui fournit des services informatiques. Elle permet de simplifier l'utilisation de politiques de sécurité en rendant les règles agnostiques à la technologie sous-jacente. Dans un environnement à service, enclin à l'utilisation de services basé sur SOAP ou basé sur le paradigme REST, il est ainsi possible de définir des règles de haut niveau qui sont ensuite vérifiées et exécutées lors de l'exécution. Nous introduisons un protocole de sécurité pour la sécurité des messages REST pour l'occasion, afin de fournir un équivalent à des briques de sécurité pour les messages qui existent déjà pour les services basé sur SOAP. Ainsi, nous fournissons au niveau de la plateforme de services de nouveaux mécanismes tels que le chiffrement et la signature afin de pouvoir transmettre des jetons en toute sécurité, rendre confidentiel et intègre les données qui transitent. La deuxième contribution présente une architecture pour simplifier la gestion des données liées à la vie privée sur une plateforme qui fournit des applications dans les nuages. Nous permettons aux clients qui développent des applications pour cette plateforme d'utiliser de nouveaux mécanismes qui permettent de modulariser des spécifications souvent soumises à régulations. Ces deux approches se basent dans le cadre de systèmes distribués, où les systèmes d'exécutions communiquent à travers plusieurs canaux, et où interviennent généralement plusieurs domaines administratifs qui nécessite une entente entre tous les acteurs.

Contribution 3

La modularisation de la sécurité informatique peut être obtenue en modifiant la façon dont la sécurité est injectée dans l'application. Au lieu de figer l'implémentation de la sécurité au cœur d'une application, nous pouvons obtenir une sécurité plus flexible et modulaire en laissant la plate-forme injecter de la sécurité à des endroits définis. Cette notion est désignée comme l'inversion de contrôle et permet de gérer l'orchestration des problématiques transversales au niveau du conteneur de l'application. Il est alors possible de définir la sécurité en tant que composant et de différer son application au sein du code au moment où l'on en a besoin, et compte tenu du contexte spécifique de l'application.

Dans cette contribution, nous proposons d'introduire un nouveau modèle de sécurité des messages de services de type REST, afin de transporter tout en protégeant les ressources. Cette contribution est issue d'une réflexion globale pour pouvoir appliquer de la sécurité de manière transparente dans un système où plusieurs domaines collaborent. Il manquait une couche de sécurité pour les services REST équivalente à la couche de sécurité définie pour les services basé sur SOAP. Ces types de services sont utilisés pour échanger les informations, et nous avions besoin de décrire des mécanismes de haut niveau qui peuvent avoir des implications sur les services. Les propriétés de sécurité apportées par cette approche peuvent être facilement introduites par des politiques de sécurité. Pour améliorer la flexibilité des transformations nécessaire à l'ajout des propriétés de sécurité, nous proposons un module accessible au niveau de la couche d'infrastructure de service web, ou au niveau d'un moniteur de référence.

La sécurité et la fiabilité des applications distribuées nécessitent une forte confiance dans le protocole de communication utilisé pour accéder aux ressources. Les plus grands fournisseurs de services et acteurs du web se tournent vers les services basés sur REST au détriment de ceux basés sur SOAP. REST propose une facilité de consommation des ressources sans encapsulation spécifique, mais manque d'une description des métadonnées comme pour une description de la sécurité associée au service. Actuellement, la sécurité des services REST repose sur une implémentation au cas par cas (dont la mise en œuvre est sujette à erreur) ou sur la sécurité de la couche de transport (offrant une faible flexibilité dans l'application de plusieurs propriétés à grain fin). Nous introduisons des mécanismes pour sécuriser la communication de service REST en permettant d'appliquer des propriétés de sécurité de manière flexible et à grain fin sur les ressources qui sont contenues dans les messages HTTP.

Nous présentons dans cette contribution une approche pour fournir une sécurité des services REST qui puisse être équivalente à celle décrite par les spécifications de WS-Security. Notre solution respecte la philosophie REST tout en minimisant la charge de traitement pour les consommateurs de ces services, et sans interférer dans l'orchestration des services déjà en place. Nous fournissons des mécanismes qui permettent de préserver la confidentialité des messages et de les signer avec une granularité fine. Le traitement effectué sur les messages est une alternative valide aux approches similaires, qui considèrent seulement l'encapsulation dans des canaux sécurisés (encapsulation SSL par exemple) et donc au niveau de la couche de transport pour les services REST. L'avantage de notre approche est de déporter la complexité pour les consommateurs de services à l'environnement d'exécution qui est alors capable de traiter et vérifier les propriétés de confidentialité et les signatures, sans pour autant changer le contenu des messages quand ce n'est pas nécessaire.

En outre, la solution que nous proposons permet de construire de nouvelles collaborations entre les différents systèmes. Nous présentons deux cas dans lesquels REST-security montre une meilleure flexibilité: fournir des moyens pour transmettre des jetons en toute sécurité, et une application facile des propriétés dans un flux de messages qui transitent par des moniteurs de référence pour valider et appliquer des politiques de sécurité. Ceci est rendu possible car nous modifions une sous-partie des messages, sans impacter le reste de la ressource. Les propriétés de sécurité sont propagées avec la ressource, ce qui nous permet d'obtenir une sécurité de bout à bout et non pas d'un point à un autre. Nous fournissons également une évaluation de la performance compte tenu de plusieurs cas d'utilisation afin d'analyser l'impact de la protection des messages sur la performance des services web. L'analyse comprend des scénarios hétérogènes et compare différents mécanismes de sécurité entre eux. Les résultats montrent que les performances avec et sans sécurité des services REST sont plus efficaces de n'importe quel point de vue, mais qui s'explique aussi par la nature de ce service: les services REST sont beaucoup moins coûteux en terme de ressource utilisée et consommé au détriment d'une description un peu moins poussée. Enfin, les services REST sont destinés à exposer des ressources, alors que les services basés sur SOAP exécutent des appels distants à des méthodes. Le protocole que nous proposons est auto-descriptif, donc toutes les informations de sécurité sur les messages sont exprimées, et le destinataire peut ainsi effectuer les vérifications et transformations sur le message.

La modularisation des propriétés de sécurité peut être introduit soit sur la couche d'application ou sur des moniteurs de références entre différents domaines distribués. Bien que nous n'utilisons pas les aspects pour injecter les préoccupations de sécurité, nous nous appuyons sur des points extérieurs à une application pour détecter l'état de la sécurité en place, et de réagir en conséquence.

Contribution 4

Dans cette contribution, nous proposons la modularisation d'une des propriétés de sécurité que nous avons vue dans d'autres chapitres: le respect des données liées à la vie privée. La modularisation est appliquée au niveau de la plateforme sur un serveur d'application dans le cloud. Les plateformes sur le cloud sont nombreuses, et elles proposent une flexibilité plus grande par rapport à des solutions déployées et gérées par des clients: les plateformes dans le cloud proposent des solutions à la demande et flexibles pour de nombreuses situations. Elles permettent à leurs utilisateurs de gérer la complexité de la configuration, l'installation, et surtout mise à l'échelle du trafic. En échange de la flexibilité, les clients acceptent de déléguer le contrôle de données avec la plateforme. La sécurité du fournisseur de la plateforme est un facteur de différenciation dans le choix d'une plateforme appropriée pour héberger leurs applications. Le rôle du fournisseur de plateforme est alors de fournir des moyens fiables et efficaces pour aider leurs clients à gérer la sécurité de leurs applications. Fournir une solution modulaire et complète pour à la fois la plateforme et les utilisateurs est difficile. Nous définissons au niveau de la plateforme de nouvelles API et des outils pour appliquer les besoins de l'application en termes de respect de la vie privée.

Le respect de la vie privée dans le cloud computing est une préoccupation importante pour les différents acteurs qui utilisent les services, mais aussi qui fournissent les services. Dans ce contexte, la conformité avec les politiques de sécurité en vigueur sur la protection des données personnelles est essentielle, mais difficile à réaliser. Par exemple, la mise en œuvre des contrôles de confidentialité est sujette à divers types d'erreurs.

Dans ce chapitre, nous présentons comment l'application de la politique de confidentialité peut être facilitée par une plateforme. Les développeurs d'applications qui doivent être déployés sur cette plateforme dans le cloud ont à leur disposition des annotations java qui enrichissent le modèle de données. Les annotations indiquent dans le code les données personnelles, ce qui permet à un système s'appuyant sur la programmation orientée aspect (AOP) de facilement détecter ces endroits. L'évaluation des préférences définies par l'utilisateur sur la gestion de ses données par l'application est réalisée par des composants de confiance fournis par la plateforme. Cela permet aux développeurs d'éviter la charge de la conception et de l'implémentation de mécanismes pour gérer les contraintes liées à la vie privée.

Dans cette contribution, nous présentons une solution pour simplifier la gestion des données liées à la vie privée dans les applications web déployées sur une plateforme dans le cloud. Nous donnons plus de contexte au niveau de l'application via des annotations et les traitements appliquées par la plateforme pour respecter la gestion des données liées à la vie privée est presque transparente d'un point de vue du développeur (la principale tâche est de placer les annotations dans le code applicatif). Ensuite, nous injectons du code au niveau de l'application pour intercepter les requêtes vers la base de données et ainsi filtrer les ressources contenant des données sensibles. Les applications qui sont déployées indiquent comment et où des informations personnelles sont manipulées. Les composants de la plateforme permettent de gérer correctement la manipulation des données sensibles.

Les avantages de notre approche sont que les détails d'implémentation sont cachés au développeur qui n'a qu'à se concentrer sur le développement de son code applicatif. Il est possible d'utiliser notre approche avec des applications existantes en ajoutant des annotations, mais sans plus de modifications. Les applications sont compatibles pour plusieurs plateformes du moment que les composants fournissant la gestion des données liées à la vie privée soient présent.

Conclusion

Dans le cadre de la thèse, j'ai développé plusieurs contributions liées au développement sécurisé d'applications. Initialement, les solutions ciblaient les applications orientées services, mais nous avons observé que les techniques et concepts s'appliquaient à une catégorie plus vaste d'applications. L'encapsulation correcte du comportement suivi de l'application des problématiques transversales dans les applications, composants, plates-formes, ainsi que tous les éléments d'un système d'information est le résultat de nombreuses années de recherche. Les solutions apportées par la recherche et l'industrie sont de plus en plus matures au fil du temps. Elles permettent de bien séparer les différentes problématiques des logiciels: code métier, code technique, code transversal; ainsi, le comportement du programme s'adapte plus facilement à plusieurs environnements. La thèse s'inscrit dans un contexte industriel, où la sécurité est une problématique majeure. Les développeurs doivent respecter plusieurs types de besoins: ils doivent mettre en œuvre des comportements métier, c'est à dire respectant les besoins fonctionnels, mais aussi s'occuper de besoins non fonctionnels qui sont tout aussi important pour la qualité du logiciel. Ces besoins se trouvent être généralement transverse à l'application, entremêlé et dispersé au sein du code. Nous utilisons des technologies existantes liées à l'ingénierie logicielle et à la modularisation de propriétés transversales afin de les adapter à des besoins de sécurité. La sécurité est un domaine complexe, et nous nous concentrons sur l'automatisation de bonnes pratiques de programmation, mais aussi à la modularisation de propriétés de sécurité nécessaire aux applications d'entreprise.

Les contributions peuvent être séparées en deux catégories: les contributions qui abordent la programmation sécurisée, et les contributions qui introduisent des propriétés de sécurité qui sont généralement indiquées dans les spécifications de l'application. Nous aidons les développeurs dans le développement de logiciels sécurisés avec le minimum d'effort dans la première approche. Nous fournissons des outils directement intégrés dans l'environnement des développeurs afin de minimiser les erreurs manuelles et éviter les pièges les plus courants. Les contributions sont une application directe des bonnes pratiques de programmation sécurisée dans lesquelles nous offrons des contrôles au niveau de l'application des comportements au sein de l'application. Dans le second type de contribution, nous proposons d'introduire des propriétés de sécurité. Ainsi, nous proposons des méthodes au niveau de la plateforme utilisée par l'application. L'environnement d'exécution fournit de nouveaux mécanismes de sécurité qui peuvent être utilisés de manière modulaire par les développeurs. La problématique diffère d'une analyse où il faut combiner plusieurs outils pour recueillir les exigences de sécurité, les faire respecter dans l'application, et vérifier leur correcte application lors de l'exécution. Nous avons développé un langage de politique de sécurité agnostique de la technologie utilisée sous-jacente, pour laquelle nous avons développé des extensions sécurisées pour des services REST. Cela signifie que les politiques de sécurité permettent de définir un niveau de sécurité dans un système. Il est possible d'orchestrer les différents composants du système et d'exécuter des mécanismes afin d'accroître la sécurité sans la nécessité de connaître les solutions utilisées. La spécialisation est déportée à l'exécution pour offrir une plus grande souplesse dans l'application de la sécurité. Enfin, dans une dernière contribution, nous avons également montré la nécessité de fournir des composants au niveau de la plateforme pour permettre de mieux gérer les contraintes de gestion de la vie privée. Ces composants sont souhaitables pour offrir une approche cohérente au lieu de laisser les développeurs implémenter une nouvelle solution, sujette à erreur.

Les différentes contributions montrent la diversité des approches de la modularisation des propriétés de sécurité. A partir de ces travaux, nous avons observé les pièges courants pour lesquels nous proposons de nouvelles directions. Nous distinguons deux types de sécurité, que nous exprimons en tant que "sécurité défensive" et "sécurité constructive". Les deux types de sécurité sont complètement différents. Généralement les deux se retrouvent entremêlés et disséminés au sein de l'application, mais les approches pour intégrer ces deux types de sécurité sont différentes. L'approche orientée aspect est traditionnellement un cas d'utilisation indiqué dans la littérature pour introduire de la sécurité au sein de l'application. En fait, il existe plusieurs degrés de représentation, en fonction des propriétés. La programmation orientée aspect est particulièrement indiquée quand on peut facilement détecter des points de tissage. Les applications que nous avons analysées ont des points fixes et faciles à détecter, tels les entrées d'applications web. Mais quand il s'agit d'analyse et de suivi de flot d'information, la détection de ces points devient plus difficile avec les langages d'aspects, et les décisions ne peuvent être prises que lors de l'exécution. Dans ce cas, nous devons fournir des solutions sur mesure, et généralement à un niveau d'abstraction plus haut pour respecter la sémantique du programme et fournir la meilleure solution pour le problème donné. La gestion des propriétés de sécurité au niveau du programme seul peut mener à des incohérences. Les programmes sont interconnectés et évoluent dans des environnements complexes qui diffèrent: configuration, politiques de sécurités, plateformes, *etc.* L'encapsulation de comportements de sécurité doit être adaptée à la nature des programmes. Afin de remédier à ces inconvénients, nous décrivons dans la suite un point de vue que nous souhaiterions adopter dans des travaux futurs.

Les langages d'aspect ont évolué au fil des avancées pour faciliter leur intégration au sein de l'application. De nouvelles primitives ont été définies dans ces langages pour permettre aux développeurs d'aspect d'écrire des cas d'utilisation complexes. Par exemple, les langages d'aspects qui couvrent le langage Java ont intégré les nouvelles spécifications du langage Java pour pouvoir décrire des comportements lorsque des annotations sont utilisées. Les langages d'aspects courant sont pour la plupart confinés à leur environnement d'exécution locale. Le problème est que les préoccupations transversales couvrent différentes couches administratives, mais aussi différentes couches techniques qui peuvent être distribués sur plusieurs systèmes. Il est difficile de proposer en une solution qui permette une interaction souple entre les différents acteurs et qui assurent la définition et l'application des propriétés de sécurité à travers toutes les couches. Nous présentons dans la suite les spécifications nécessaire à l'élaboration d'un système que nous souhaiterions réaliser: une application cohérente et systématique des propriétés transversales de sécurité à travers les différentes couches et différents domaines. Nous cherchons de nouvelles primitives pour les langages d'aspect afin de fournir de nouvelles extensions à ce qui existe. Cela signifie que nous devons fournir des outils pour le système d'aspect. Le but recherché est de pouvoir augmenter l'information liée aux ressources qui rentrent et sortent de l'application. Ce langage aurait pu nous aider s'il avait été disponible lors de l'élaboration de nos précédentes contributions.

L'approche se place dans un environnement distribué, qui est composé de composants distribués dans plusieurs domaines. Chacun de ces domaines a un environnement d'exécution qui lui est propre. L'application (en cours d'exécution sur un environnement d'exécution locale) échange des messages avec les autres composants externes, qui ont tous une configuration de sécurité particulière. Afin de renforcer la confiance dans les données qui transitent entre ces intermédiaires, et afin de faciliter la vérification et la transformation des propriétés de sécurité, nous définissons de nouveaux outils et moyens pour intercepter les données à l'aide d'aspects. Nous extrayons les informations de sécurité liées aux ressources afin de les proposer via les primitives que nous définissons aux développeurs d'aspect. Cela permet aux développeurs d'aspect de réagir sur des évènements survenant au sein de l'application (données chiffrées en local sur le point de sortir par exemple), mais cela permet aussi d'ajouter de manière transparente des méta-informations de sécurité aux ressources. Notre solution propose d'extraire les informations sur la couche de communication et de les rendre disponible à la couche applicative. A l'inverse la solution permet, lorsque des ressources sortent du champ applicatif à destination d'un domaine distant d'extraire le contexte de sécurité de la ressource sortante et de la propager sur la couche de communication. Cette approche ouvre de nouvelles possibilité de collaboration entre les différents acteurs, et ce de manière automatisée dans un bon nombre de cas qu'il n'était pas possible de réaliser précédemment.

Chapter 1

Introduction

1.1 Background

The thesis brings an industrial point of view to the modularization of security concerns in enterprise applications. Security is pervasive in our environment and implies rigorous steps in the development lifecycle to correctly specify and implement security properties in a consistent manner. Security is usually referred to as a non-functional and cross-cutting concern that is independent from application core concerns, bringing overhead all along the development. But security has different facets and might have deep relationship with the application, making it hard to properly modularize it in all situations.

The fast evolution of enterprise applications brings a new dimension to security. In the course of the past decade, enterprise applications have evolved towards more connectivity. Hence, they expose their data through several means such as web applications, service oriented applications, cloud applications, mobile applications, *etc.*. This interconnection provides greater flexibility to create new businesses, to collaborate with partners, and to integrate their solution in the ever-complex enterprise landscape. But it also brings additional exposure to their business data through heterogeneous technological stacks, making it even more complex to consistently cover security. Recent events present security breaches with high impact. Attacks have been directed towards major companies, large organizations, but also institutions and countries. The attacks tends to originate from organized groups. The attacks get successful although complexity differs. They can be extremely complex, in which attackers use several means to penetrate a system and obtain what they want, such as social engineering, zero-day exploit, stealing of cryptography keys, robbery, *etc.*. In such case, it is hardly possible to protect sensitive resources, and companies need to perform regular audits on their processes to detect as soon as possible some intrusion to properly react. There is also another kind of attack that we are covering in this thesis, that are some attacks directed towards technical stacks of heterogeneous IT systems. They are less structured, in the sense that they necessitate less organization in time and in means. It exists several tools and frameworks to perform this kind of attacks, that can be controlled by individuals or groups that are looking for fame, rewards, knowledge, *etc.*. These attacks can generally be mitigated by respecting security policies and applying best practices. The problem in this case, is to propose a set of tools and methods to ensure the correct enforcement of security policies and best practices at several layers of the computing stack. For instance, an application will be running on an execution platform, coded in a specific programming language, using some frameworks and using various communication protocols. Each of these elements has to be considered to ensure the absence of vulnerabilities and the proper implementation of security properties.

We have started with the idea that applications are increasingly pervasive in our environment, ranging from embedded devices to large-scale distributed applications supporting economical exchange across countries. The different types of software, no matter their environment, need to respect quality constraints. The industry first focuses on delivering high quality products, providing added-value in a specific functional area. It can be products to support business process, to sell solutions and services, to ease exchange between individuals, or even products to facilitate development of new applications. Each of these products have specifications to describe the functional concerns that are the core business of the software owners. In addition to the functional concerns, the industry has to respect several guidelines when architecting and developing solutions: internal policies, law, regulations, non-functional specifications, *etc.*. These are concerns that address specifications for undirect business need. They are present to ensure the correct execution of the application in its environment. We often referred to these concerns as non-functional concerns. In general, the non-functional concerns are such as usability, integrity, reliability, performance, *etc.* [CPL09]. Last, but not least, a concern that often comes in literature as a non-functional concern is security.

We are going to see in the next chapters that security has several faces. It is generally considered as a non-functional concern that crosscut the application, that one can address straightforwardly, and that has several tools or frameworks that perform security (such as third-party libraries, execution environment support, operating system protection, application gateway, *etc.*). In reality, security conceals a large range of requirements from functional concerns to non-functional concerns. Companies have to develop products with all these concerns correctly orchestrated.

1.2 Overview

The thesis discusses how to develop secure software in distributed environments. For this purpose, we have developed techniques to enhance flexibility in operational development of security. We apply the concept of cross-cutting concern, mainly promoted by the Aspect-Oriented software development [KLM+97] paradigm to enhance flexibility in the definition of security properties. We defer the concrete enforcement of security properties in the application to first collect the location impacts in the source code of a program, then to later react at the application level scope to adapt the security enforcement code to the final execution environment.

We propose with the support of several contributions to address the modularization of security properties in industrial environment, that address both techniques for constructive security, and techniques for defensive security. We introduce a separation in security categories as we differentiate security that addresses business needs such as privacy, confidentiality, authenticity, and sometimes safety, from security that addresses protection against the exploitation of softwares' vulnerabilities by malicious users. The contributions propose to ease the management of security properties along the security lifecycle, with a focus on development. With this respect, we propose contributions to protect application from vulnerabilities, and contributions to ease the integration of security properties in applications.

1.3 Problem Statement

The problem of secure development has evolved over the years. It is now at a mature stage, but we observe that vulnerabilities introduced by developers are still prevalent. We tackle the problem of achieving a non-invasive and systematic application of security, while respecting the time constraints of the industrial environment when developing software. The security requirements are coming from security policies in place to respect software quality standards, but also from external regulations. The security properties of such systems affect different layers that are heavily interconnected. The security of a system is impacted by several decisions along the software development. It starts from the definition of security requirements. Assuming that the requirements are correctly elicited and specified, the developers are given a list of rules to respect while developing. They have to maintain a certain quality of coding when introducing the security mechanisms fulfilling the security requirements. Despite the numerous resources brought by the secure development discipline to correctly implement security mechanisms, developers still introduce vulnerabilities that affect systems.

The introduction of security vulnerabilities by the developer depends on several factors: the lack of knowledge about security mechanisms, the misinterpretation of the specifications or of the software's architecture, the misconfiguration of the frameworks and libraries, or even some refactoring defects when going from testing and development to production [KYL09]. There also exists the possibility of having an internal developer who deliberately introduces backdoors or defects in the application. It is important to notice that the cost to correct a security vulnerability, if taken as a software defect [oST] will exponentially grow while software development progresses. The exposure of software vulnerabilities becomes effective when the application is released to the production stage, in other words, when the application is supposed to fulfill all the goals it has been developed for, and the application is exposed to the intended audience.

In order to mitigate the unintentional defects, we first propose to accompany the developer with methods and tools. To clarify the situation, we have defined an internal set of security requirements, coming from a subset of internal policies in place at SAP, but also general requirements we have observed in best practices [OWA, Roo07, CM07]. The requirements target primarily the web application vulnerabilities, but also all defects one encounters when dealing with distributed systems. In this first part, we have restricted the list to requirements impacting the development. We are interested in detecting and preventing all input manipulations: cross-site scripting, SQL injection, path manipulation, remote shell injection, HTTP parameter pollution, and any kind of protocol injection. In the second part, we propose to cover security properties related to constructive security, and to present several security mechanisms. The contributions cover properties such as accountability, which imply the correct enforcement of several mechanisms, as well as privacy and communication protection within distributed systems.

The correct modularization allows companies to propose software products that are compliant in the many regions the applications are commercialized in, by applying late binding to the properties. For instance, the privacy of personal data will differ from one country to another, with potentially different cryptographic restrictions to protect resources. Although the solution we develop in the following can be applied in several contexts, we have developed them in a context of distributed systems. We are targeting inter-connected systems in an architectural style that is often referred to as Service-Oriented Architecture (SOA). We are also bringing solutions to the growing deployment of cloud computing systems, that not only target applications but also infrastructure and platform.

1.4 Contributions

The contributions of this thesis are manifold. They can be summarized as follows:

- *Review and identification of security problems for distributed systems with Aspect-Oriented Programming*
- *Detection of security vulnerabilities with corrective patch applied in AOP, to help the development of secure web applications*. It lead to publications [GEKS11, SGEKSDO12, SSO13]
- *Modularization of security concerns like privacy for distributed platforms*. It lead to publications [YSSSdO12, DSI$^+$12b, SSdOMR12]
- *Proposition of a cross-layer aspect system*

1.5 Organization of the thesis

The thesis is organized to present the aforementioned contributions in the area of secure software engineering for distributed applications. The next chapter introduces the different concepts we are referring to throughout the thesis. We have divided the thesis into two parts. The first part presents *modularization of defensive security* which is the introduction of programming aspects for security. It covers contributions related to secure programming with the assistance of Aspect-Oriented paradigm. The second part presents the *modularization of constructive security* which can be simplified to the modularization of business security concerns to propose more flexibility in the application of security properties. In the conclusion part, we present the definition of a *cross-layer and parametric aspect system*, which would be our next step towards modularization of security properties across layers.

Chapter 2

Modularization of cross-cutting concerns

This dissertation describes a contribution to the secure engineering discipline, that promotes the separation of concerns in application development of security modules. It allows to better architect solutions, but also to bring flexibility in the application of security policies. In the following, we briefly introduce the origin and goals of separation of concerns, through the presentation of the Aspect-Oriented programming discipline. Then, we present the security processes that are generally defined within the application development to produce secure applications.

2.1 Software Engineering

The software engineering discipline is vast and promotes concepts to properly manage software development from its early existence. The field has developed over time several techniques and methodologies to produce software with a focus on delivery time and quality. It gives guidelines to manage specifications, to gather and elicit requirements, to define an architecture with design of components and interfaces, to provide procedure for testing, to maintain software, to manage configuration, to review processes around management, to define security and safety, *etc.*.

The primarily goal is to translate real-world concerns to software, letting customers expressing needs that are then translated to applications that runs on computers. The whole process is complex and involving heterogeneous actors. It is also still evolving, with new methods to better manage systems. The definition of software is made in terms of a set of requirements, that represent the business goal of the application. The translation of these business needs in software is realized using a programming language. A set of concepts also govern programming engineering.

At the beginning, the programs were defined as structured instructions executed sequentially. The Figure 2.1 shows an evolution of programming paradigms these past decades, that diverged in two main dimensions. The first dimension represents an advanced conceptualization of components through the programming language. For instance, programming languages started to conceptually define separation of concerns to virtually manage piece of software independently.

The major evolution that raised in the eighties was the evolution from structured programming to object-oriented programming. The object-oriented programming is a paradigm that has several objects which encapsulate common behaviors. It promotes flexibility and reuse of concerns. The data is accessed through interfaces defined by the objects. In short, good object-oriented design might be achieved by applying five main principles. These principles have been compiled by Robert Cecil Martin [Mar99] :

Single responsibility principle promotes the encapsulation of one concern ine one class. The reason is to clearly define responsibilities, to have independent classes and avoid the introduction of side-effects when modifying a class dealing with more than one concern.

Open-closed principle indicates that classes should be open for extension, but closed for modifications. It means that the design of a class should be complete. New features would not modify directly the source code, but rather pass through the creation of a new class. In such situation, code reuse could be achieved through inheritance.

Liskov substitution principle comes from Barbara Liskov [Lis88] and defines a notion of substitutability for types. It is also called behavioral subtyping and provides standard requirements in the definition of method signature for the sub-class hierarchy. For instance, it states that "objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program". It lays the foundation for design by contract.

Interface segregation principle promotes the smallest dependency possible by splitting large interface in smaller and more specific ones. It allows to decouple concerns and thus ease code refactoring.

Dependency inversion principles states that high and low level modules should depend upon abstractions. Abstractions should not depend upon details, but details should depend upon abstraction. It promotes the separation of layers by specifying abstract notions from which the code will depend at a certain level. It allows then to abstract notions and decouple any relation to concrete/technical low-level code.

Although the separation is clearly defined, the object-oriented paradigm has some limitations. First, the paradigm works well for local systems, but suffers from a certain scalability when addressing distributed systems. Distributed applications rely on additional layers to manage and moke object communications. Second, the encapsulation provides protection and control to the internal data of an object, but software development requires additional code to manage cross-cutting concerns. These concerns can not be encapsulated like in an object, as it represent a concern that is highly tied to the code. We say that it is scattered and tangled to the application. It is generally technical code, such as code to manage transactions, caching, and in some cases security. All of the tangled code breaks the principle of having classes responsible of one concern, dealing with abstractions rather than low levels.

In the separation of structure and behavior, Gregor Kiczales and the Xerox Team [KLM$^+$97] introduced the Aspect-Oriented programming paradigm in late nineties, with the goal to untangle the code into cross-cutting, loosely coupled aspects. We further discuss this paradigm in next section.

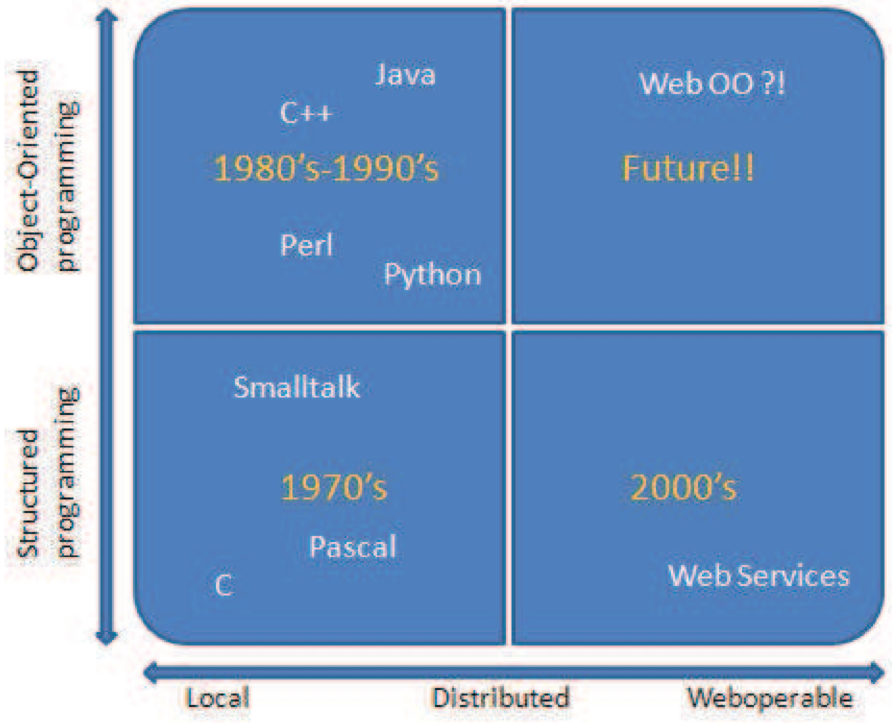


Figure 2.1: Programming paradigms categorization presented in [USB09]

2.2 Aspect-Oriented Software Development

Aspect-Oriented Software Development (AOSD) is a whole discipline around the concept of aspect. It describes approaches to software modularization and composition, and gather techniques and methodologies to incorporate the aspect concepts. The set of software development techniques include requirement engineering, analysis and design, architecture, testing, and programming. The last point is the core of AOSD, but requires support of the aforementioned techniques with either tools, or methodologies. The preliminary steps elicit requirements that represent cross-cutting concerns. In software, there are several concerns that can be either functional or non-functional. It means that some concerns are business related, while other are related to the correct execution of the program and are generally implied by some technical choices: environment, programming language, framework, platform, etc. These concerns are sometimes translated in programming pieces that are scattered and tangled over the code. Such concerns are good candidate to be implemented with aspects.

There are several use cases that are discussed. For instance, the benefit of adopting Aspect-Oriented Software Development in Business Application Engineering [PCG$^+$08] for large companies lies in overcoming several challenges in the context of specific products. AOSD is not bound to a particular language, and needs several tools and methods for the particular environment of the company: debugging tools, processes that respect existing business process, extension to languages, flexible mechanisms, etc. Also, AOSD does not necessarely fits all projects, and needs to be introduced slowly. For instance, Robinson and al. [LSM05] present the relation between AOSD and security in the area of application security management, in which the application handle security policy and needs to enforce specific behavior. they differentiate the business logic from the security management logic, the security enforcement logic and the communication logic. They show that AOSD fits the use case by using an interaction specification language describing behavioral dependencies between components, and runtime adaptation in the interaction scheme.

AOSD refers to the general concept of aspect that is implemented in a particular language. We are discussing the aspect paradigm through the presentation of Aspect-Oriented Programming. The term Aspect-Oriented-Programming [KLM$^+$97] (AOP) has been coined around 1995 by a group led by Gregor Kiczales, with the goal to bring proper separation of concerns for cross-cutting functionalities. Roots for foundations can be traced back to adaptive programming, or composition filters [Lop05]. O. Selfridge introduced a notion that can be related to AOP as "demons that record events as they occur, recognize patterns in those events, and can trigger subsequent events according to patterns that they care about" [Sel58]. But the approach has then derived to become a discipline apart.

It is a paradigm to ease programming concerns that crosscut and pervade applications. The aspect concept is composed of several advice/pointcut couple. Pointcuts allow to define where (points in the source code of an application) or when (events during the execution of an application) aspects should apply modifications. Pointcuts are expressed in pointcut languages and often contain a large number of aspect-specific constructs that match specific structures of the language in which base applications are expressed, such a pattern language based on language syntax. Advices are used to define modifications an aspect may perform on the base application.

Advices are often expressed in terms of some general-purpose language with a small number of aspect-specific extensions, such as the *proceed* construct that allows the execution of the behavior of the base application that triggered the aspect application in the first place. The main advantage using this technology is the ability to intervene in the execution without interfering with the base program code, thus facilitating maintainability.

Aspect Oriented Programming goal is to intervene in first class language execution. Most of current languages are related to programming languages such as the well known AspectJ. AspectJ is a java-based language that allows different weaving scheme and provides the most recent progress related to AOP features. Through the definition of a pointcut based on a pointcut language, one can represents an aspect that enhance base classes. It is possible either to enhance business objects with inter-type declarations, or adding a specific behavior when conditions are met. The language heavily relies on program syntax to wrap behavior respectively $Before$, $Around$, or $After$ a method execution. But AspectJ is not the only language. It exists nowadays a language for several common programming languages, but also other domain-specific languages.

For example, AO4BPEL represents the willingness to provide a clear definition of cross-cutting concerns for business processes described with the BPEL language. cross-cutting concerns are normally tangled and scattered across the code, such as Activities and Tasks within the business process that are composed with other services, with no distinction between the real business code and the concerns. In order to provide functionality that pervades and crosscutt several business process, Charfi [Cha07] focused on modularity of workflow process specifications. It allows, with a language extension to describe non-functional requirements separately. The approach is similar to Aspect concept for object language but translated for business process languages. The defined aspect language is able to represent concerns such as logging, transaction or security. Charfi proposes deployment mechanisms to ensure correct application of non functional requirements, based on an XPath pointcut language describing where and when to inject behavior in business process. The BPEL engine is responsible of interpreting deployment descriptor and weave cross-cutting concerns.

2.2.1 A pointcut-advice model

AspectJ is the foundation language for cross-cutting concern definition and implementation for java programs. It relies on a so-called pointcut-advice model for the definition of cross-cutting concerns, such as many of other aspect-languages. It allows to clearly separate the definition of the cross-cutting concern to the definition of their weaving points in the application.

The pointcut model in AspectJ defines boolean expressions that matches some points in a target application (the application in which one want to inject the cross-cutting concerns). The boolean expressions can be combined and mostly relies on the syntax of the application. For instance, the pointcut model defines *method call, construct call, method execution, construction execution, advice execution, within method code, field getter and setter, static class instantiation, within class, this, target, args* primitives. It makes it possible to select and filter several points from a target application, that are called joinpoints. The joinpoints are the concrete points that are selected when applying a pointcut boolean expression. The pointcut model is the first part of an aspect, and its expressivity can be extended through several works. For instance, Masuhara

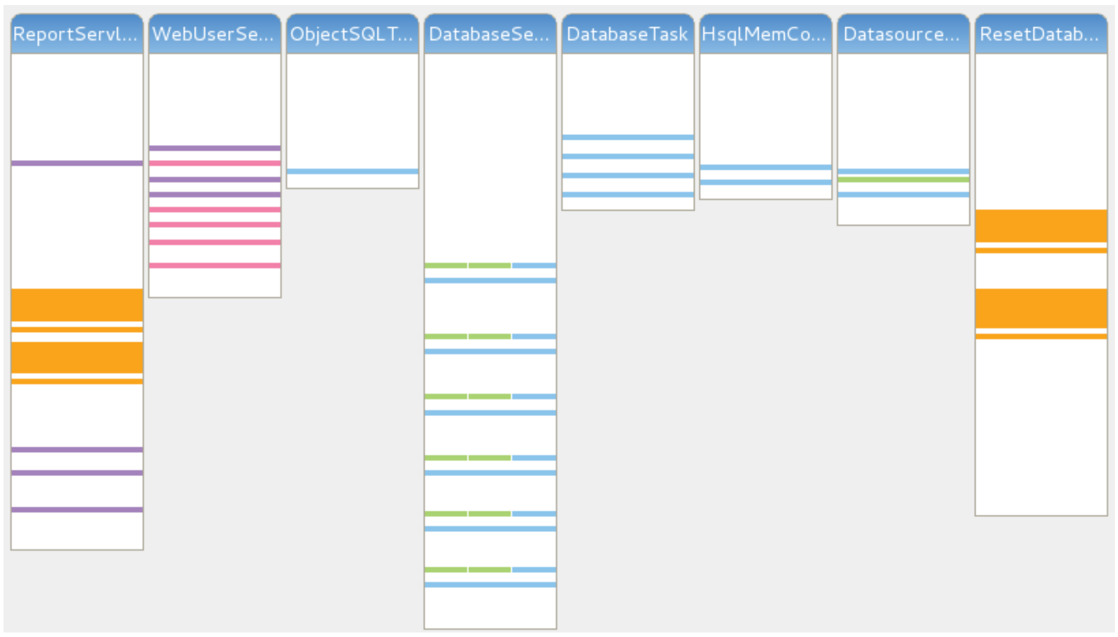


Figure 2.2: cross-cutting concern visualization with the AJDT eclipse plugin for a deliberate vulnerable application

and al. defined a new primitive called *dflow* [MK03] to quickly select points in the application to track a dataflow. Another example comes from Belblidia thesis in which she defines two new primitives *getlocal, setlocal* [Bel08] that allows to track local variables from within a method body, thus observe the propagation of information flow from the aspect language. The Figure 2.2 presents a view that is produces by the AJDT eclipse plugin for AspectJ. It shows the actual joinpoints that applies in the context of an application for several concerns. The different colors present different concerns - hence aspects. With a simple pointcut language, one can defined cross-cutting concerns that apply at several points in the application.

The second part is the advice model. The advice defines the actual concern, meaning the code that executes in order to fulfill functional concern. There is several means to inject the context of the application in an advice. For instance, it is possible to access variables, parameters, or instances of the base applications. The aspects might either be independent or completely modify the application flow. Typical aspect use cases that are completely independent from the application are logging aspects. They need to access context to extract information and log them. This technical code is best described as an aspect when you want to log common information, such as every time a web application executes a method. In some cases, you want from within an advice to modify the return of a method, an instances, or some variables. For example, you can want to provide mock up application with aspects, that always replace some functionality of your application on test platform. Also, you would need to modify variable content with safe-content after security check.

The pointcut-advice model provides an elegant approach to describe cross-cutting concerns that apply to components, such as objects, or any type of container. A problem has nevertheless been encoutered in the practical use of aspects, which is called the *fragile pointcut problem* [SK04]. The problem occurs on specific pointcut languages, upon evolution of an application. As the set of aspects is separated from the application, aspects use pointcut languages to determine the joinpoints. Most of the pointcut languages propose a syntax-based approach to write pointcut. It the pointcuts get desynchronized from the syntax of an application (i.e, a package is renamed, or an object type change its name), the aspects can no longer interact with the base application.

The binding between the application and the advice is complex, but is not the only weak points. De Fraine et al. [FSJ08] present StrongAspectJ which provides flexible and safe binding between the advice and its corresponding pointcut. The problem in this case is related to the type system used by the advice/pointcut. They enhance the existing pointcut model to recover type safety for both generic and non-generic pointcut/advice declarations.

In the next sections, we provide concrete examples of the pointcut-advice model with the AspectJ language.

2.2.2 AspectJ

We illustrate the usage of the Aspect-Oriented programming concept through AspectJ examples. AspectJ is one of the many implementation dedicated for the java language. It is also the primary language created by Gregor Kiczales and its team in Xerox parc. Since, it represents the most updated language, back-porting several of concepts developed in the scope of the research in the field.

We represent a base program example in the Listing 1. A base program traditionally provides the functional concerns of an application. These concerns are those defined by the business users during the design of the application, either through careful specification or requirements documents, or through other communication means. In this example, we are presenting a simple execution of methods with some latency. The program, for demonstration purpose only does nothing else than a loop with a delay.

AspectJ and aspect-oriented programming in general manipulates indirectly languages. They introduce several instructions that are not directly represented in a first-class language. For instance, developers and aspect writers write java lines of code independently. Then, through an additional step called weaving in aspect terminology, they unify the concerns together. In such case, the syntax of the base program does not change, but its semantic will when the aspects are added to the program. The modification are introduced in this case in java-bytecode. In order to compare the effects of aspects static compilation, we presents in Listing 1 the corresponding instructions obtained after compilation in Java 1.7(check `http://docs.oracle.com/javase/specs/jvms/se7/html/index.html`).

The code presented in Listing 2 represents a cross-cutting concern defined in AspectJ. The java annotations add information for AspectJ system to interpret correctly the different methods. In this case, it presents a concern called *aroundMethod* which is inserted in the program execution flow, every time a method is called from a place which is not the aspect module. The aspect itself monitors execution time of the base application, and displays a message for each method execution. An output result is given in Figure 2.3.

Listing 1 Simple class counting with delay and its equivalent bytecode

```java
public class Simple {

  public static void main(String[]argv) {
    countFast(1000);
    countSlow(1000);
  }

  public static void countSlow(int value) {
    count(value,5);
  }

  public static void countFast(int value) {
    count(value,0);
  }

  private static void count(int value, int delay) {
    for (int i=0;i<value;i++) {
      try {
        Thread.sleep(delay);
      } catch (Exception e) {}
    }
  }
}
```

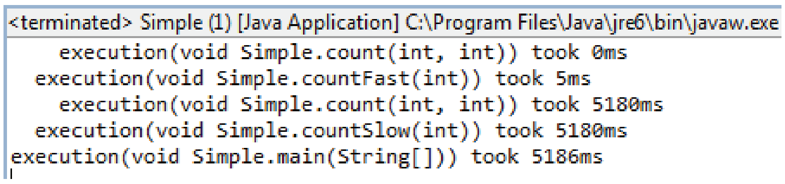
```
public class Simple {
  //...
  public static void countSlow(int);
        0: iload_0
        1: iconst_5
        2: invokestatic  #4    // Method count
        5: return

  public static void countFast(int);
        0: iload_0
        1: iconst_0
        2: invokestatic  #4    // Method count
        5: return

  private static void count(int, int);
        0: iconst_0
        1: istore_2
        2: iload_2
        3: iload_0
        4: if_icmpge     22
        7: iload_1
        8: i2l
        9: invokestatic  #5    // Method Thread.sleep
       12: goto          16
       15: astore_3
       16: iinc          2, 1
       19: goto          2
       22: return
}
```

In order to properly implement new concerns into the base application, AspectJ adopts different strategies. It is for instance possible to weave the concerns either statically or dynamically. In the following, we have compiled statically the two java classes (Simple.java and WhereDoesTheTimeGo.java) and apply the static weaving approach. We can achieve the same result by loading the aspects at the beginning of the application execution. It is realized thanks to java bootstraping, intercepting calls to methods and injecting when detecting a pattern match the loaded concerns. In our case, the static weaving renders bytecode application with the different concerns already inlined in the application. We present in Figure 2.4 an UML class diagram of the base application and the aspect module. We differentiate in this picture the different elements that are added by the weaving process in the application. The black elements are class diagram representing the base application and the cross-cutting concern. The red elements are the addition when compiling and weaving the aspects into the application. Several inner classes, and several wrappers around the concerned methods are added to execute at runtime the desired properties.

We provide a last listing, to observe the change at the bytecode level. The Listing 3 contains therefore several new constructs compared to the compilation with no aspects. To keep high modularity in the aspect concern, AspectJ transforms the bytecode to obtain a separation between the actual concern and the original source code. Therefore, for the *countSlow* function, the original method content remain unchanged in the bytecode, but the call to the method as additional wrappers to execute matching aspects at the joinpoint.

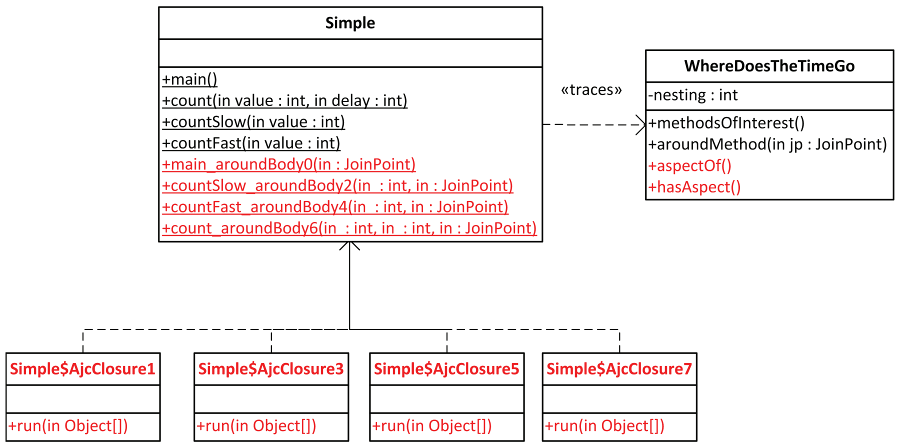
Listing 2 A cross-cutting concern to display the time spent in methods.

```
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class WhereDoesTheTimeGo {

    @Pointcut ("execution(* *(..)) && !within(WhereDoesTheTimeGo)")
    void methodsOfInterest() {}

    private int nesting = 0;

    @Around ("methodsOfInterest()")
    public Object aroundMethod(ProceedingJoinPoint thisJoinPoint)
        throws Throwable {
        nesting++;
          long stime=System.currentTimeMillis();
          Object o = thisJoinPoint.proceed();
          long etime=System.currentTimeMillis();
          nesting--;
          StringBuilder info = new StringBuilder();
          for (int i=0;i<nesting;i++) {
            info.append("  ");
          }
          info.append(thisJoinPoint+" took "+(etime-stime)+"ms");
          System.out.println(info.toString());
          return o;
    }
}
```



```
<terminated> Simple (1) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe
    execution(void Simple.count(int, int)) took 0ms
  execution(void Simple.countFast(int)) took 5ms
    execution(void Simple.count(int, int)) took 5180ms
  execution(void Simple.countSlow(int)) took 5180ms
execution(void Simple.main(String[])) took 5186ms
```

Figure 2.3: Application execution output result



Simple
+main()
+count(in value : int, in delay : int)
+countSlow(in value : int)
+countFast(in value : int)
+main_aroundBody0(in : JoinPoint)
+countSlow_aroundBody2(in : int, in : JoinPoint)
+countFast_aroundBody4(in : int, in : JoinPoint)
+count_aroundBody6(in : int, in : int, in : JoinPoint)
«traces»
WhereDoesTheTimeGo
-nesting : int
+methodsOfInterest()
+aroundMethod(in jp : JoinPoint)
+aspectOf()
+hasAspect()
Simple$AjcClosure1
+run(in Object[])
Simple$AjcClosure3
+run(in Object[])
Simple$AjcClosure5
+run(in Object[])
Simple$AjcClosure7
+run(in Object[])

Figure 2.4: Class diagram of the example. The aspect and the base application are in black. The weaving addition are in red.

Listing 3 Result of weaving on application bytecode for the *countSlow* function.

```
public class Simple {
  //...
  public static void countSlow(int);
    Code:
       0: iload_0
       1: istore_1
       // Field Lorg/aspectj/lang/JoinPoint$StaticPart;
       2: getstatic     #82
       5: aconst_null
       6: aconst_null
       7: iload_1
       // Method org/aspectj/.../Conversions.intObject
       8: invokestatic  #88
       // Method org/aspectj/.../Factory.makeJP
      11: invokestatic  #55
      14: astore_2
      // Method WhereDoesTheTimeGo.aspectOf
      15: invokestatic  #75
      18: iconst_2
      // class java/lang/Object
      19: anewarray     #3
      22: astore_3
      23: aload_3
      24: iconst_0
      25: iload_1
      // Method org/aspectj/.../Conversions.intObject
      26: invokestatic  #88
      29: aastore
      30: aload_3
      31: iconst_1
      32: aload_2
      33: aastore
      // class Simple$AjcClosure3
      34: new           #92
      37: dup
      38: aload_3
      // Method Simple$AjcClosure3."<init>"
      39: invokespecial #93
      // int 65536
      42: ldc           #63
      // Method .../AroundClosure.linkClosureAndJoinPoint
      44: invokevirtual #69
      // Method WhereDoesTheTimeGo.aroundMethod
      47: invokevirtual #79
      50: pop
      51: return

  static final void countSlow_aroundBody2(
    int, org.aspectj.lang.JoinPoint);
    Code:
       0: iload_0
       1: iconst_5
       // Method count
       2: invokestatic  #26
       5: return
  //...
}
```

```
public class Simple$AjcClosure3
  extends org.aspectj.runtime.internal.AroundClosure {
  public Simple$AjcClosure3(java.lang.Object[]);
    Code:
       0: aload_0
       1: aload_1
       // Method org/aspectj/.../AroundClosure."<init>"
       2: invokespecial #10
       5: return

  public java.lang.Object run(java.lang.Object[]);
    Code:
       0: aload_0
       // Field org/aspectj/.../AroundClosure.state
       1: getfield      #14
       4: astore_2
       5: aload_2
       6: iconst_0
       7: aaload
       // Method org/aspectj/.../Conversions.intValue
       8: invokestatic  #20
      11: aload_2
      12: iconst_1
      13: aaload
      // class org/aspectj/lang/JoinPoint
      14: checkcast     #22
      // Method Simple.countSlow_aroundBody2
      17: invokestatic  #28
      20: aconst_null
      21: areturn
}
```

2.3 Security Implication in the Software Development Life Cycle

The secure software development lifecycle gather techniques and methodology to follow in addition to the software development lifecycle to produce high quality solutions, respecting compliance and quality level to avoid flaws and vulnerability when developing applications. At the early age of software development, the complexity and time to write a single piece of software was too high to let someone abuse of a system. With the rapid and growing usage of application for any kind of problems, people started to exploit software logic to bypass some internal limitations. Then, when systems became ubiquitous and inter-connected, a new category of people have been given access to mass systems. Software are complex pieces designed and developed by individuals and automated tasks. Sometimes, without the possibility to control entirely the process, some errors are introduced in the application. It goes from architecture flaws, where specifications represent a subset of the real-environment, letting some unknown behavior at the mercy the luck. It can also be software weaknesses, that when exposed to some attackants can introduce security vulnerabilities. These vulnerabilities can be detected and exploited by a large range of attackers. The rapid growing of systems and complexity to manage several teams together to timely deliver softwares combined to the structuration of governments, groups of security experts, talented system hackers and a whole range of malicious users lead to the emergence of a new discipline in computer science. The aim is to not only produce softwares, but produce high quality softwares compliant with regulations in place, internal company policies, etc. Furthermore, the lifecycle involves several actors from the early gathering of specifications to the final stage of testing, deployment, and detection. In this section, we detail the different steps that are traditionally accepted as being part of the so-called SDLC (Security Development Lifecycle).

The global picture is presented in Figure 2.5. It presents several points around the traditional software development lifecycle in which actors interact to augment the system coverage with security properties. This Figure is the starting point for the numerous contributions of this thesis, as it brings an approach to ease integration of security at different points over the security development lifecycle. We try to makes more flexible and completely modular the security properties with clear separation of concerns from business code to technical code, and from security code to business code.

The overall vision of the thesis starts from the design and development of web applications to assist concerned stakeholders in their tasks to properly implement secure applications. It also touches the branch of application testing through careful review and assistance in correction in an integrated plugin. The thesis has some important implications at the runtime: the solutions separate concerns during the development of the application, and then delay the final binding of security code with business and technical code to the last moment: deployment or execution. In some cases, we have bundle security out of the direct application, and preferred to offer it as a service, wrapping around the application. It is the case for instance with service framework security and platform security.

The field is also known as security hardening, and pertains different levels: code level hardening, software process hardening, design level hardening, and operating environment hardening. The achievement of these levels render systems resilient to attacks.

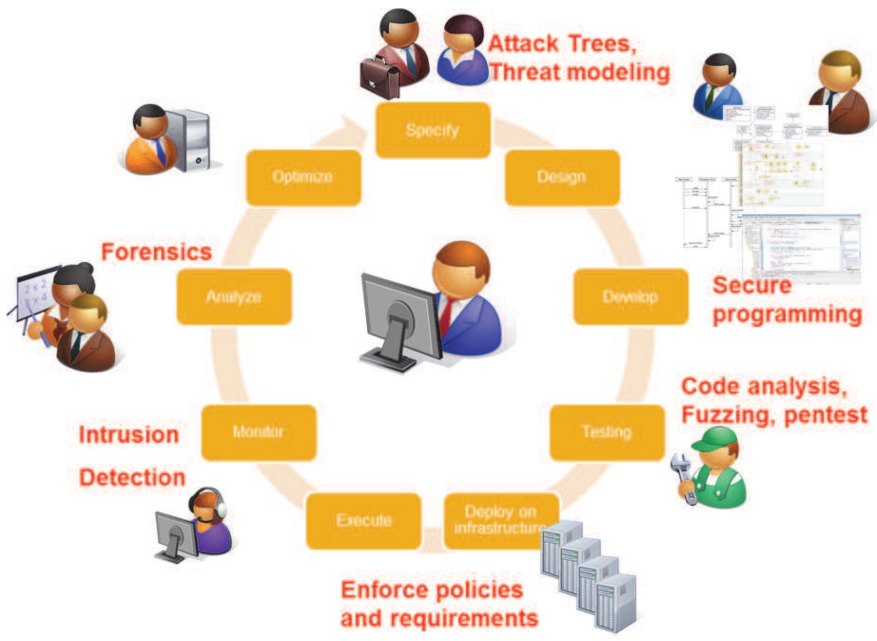


Figure 2.5: Software Development Life-Cycle

2.4 Security Requirements

The engineering of the requirements for software application, systems, or components cover several functional and non-functional definitions. Among them, we can identify requirements such as quality, interoperability, performance, portability, availability, reliability, usability, and security requirements. Security requirements are one of the most difficult to deal with, as requirement engineers are not used to complexity and diversity of security. They lack of security architectural overview, and often confuse security mechanisms and security properties. There are several kinds of security requirements, that we specify in the following with our specific needs throughout the thesis.

Identification Requirements specifies the extent to which the application shall identify the actors before interacting with them. This requirements often goes along with authentication in order to properly know the involved actors. In industry, the identification steps is generally mandatory to render services.

Authentication Requirements specifies the verification that apply to validate the identity of involved actors. In industry again, and in most of our use-cases, the application shall verify the identity of all of its users before granting the system access. In some cases, we restrict authentication to critical sections, where sensitive or personal data is manipulated.

Authorization Requirements specifies the access and usage control of authenticated users and actors. The authorization needs both a correct authentication and identification of the users, in order to grant the execution of critical part of system.

Immunity Requirements specifies the protection against infection. Although this requirements generally apply for virus infection, in our use-cases we define some requirements to protect against injection attacks. For instance, we indicate an application shall be immune against injection attacks that are frequent in web application and service-oriented applications.

Integrity Requirements specifies the extent to which application shall ensure the consistency of data. The requirements permits the detection of any alteration or deletion of data. In our industrial use cases, the integrity applies to communication, whom we prevent unauthorized users to manipulate data.

Non-repudiation Requirements specifies to which extent an application shall capture evidences of interaction with actors. The non-repudiation is fundamental to avoid deny of actors for having participated in a conversation with the application. It generally tracks information such as the identity of the involved actor, as well as actions or data in manipulation throughout the interaction.

Privacy Requirements specifies how an application or component deals with sensitive and personal data. The data are generally under regulations and have a specific lifetime. Also, the privacy requirements keep information private from unauthorized actors, and is a good way to respect the "need-to-know" principle.

There are a couple of additional security requirements categories that we are not addressing along the thesis. It is for example survivability requirements to survive the alteration or loss of system or part. Physical protection requirements that address physical locations and plans to avoid robbery and deterioration of hardware. Intrusion detection requirements provides specifications to detect and record attempt to access application by unauthorized parties. Security auditing requirements which specifies to which extent an application shall collect and report the state of its security. Finally, system maintenance security requirements propose plans to properly manage migration and upgrade with no conflict with existing security requirements.

2.5 Security and aspects

AOP has been proposed to represent functionalities of a software in a decorrelated fashion, so that they can be incorporated anytime, and at any location. Several languages exist to describe the behavior of the functionalities, and to bind the behavior to the actual code. In the following, we are calling them advice language and pointcut language. Over the past decades, these languages evolved to incorporate new primitives to ease definition of location. They have also exposed through an API context of an application to be reused inside aspects. These languages have been used to represent several cross-cutting concerns : logging, transactions, and some security concerns. In the past, these concerns where encapsulated in applications with introduction of explicit calls. AOP render the binding of these properties more transparent from an engineering point of view.

There is a long list of already identified cross-cutting concerns [Mic, Paw02]. Authentication & Authorization code, that tends to appear before critical sections. Database encryption to provide systematic encryption of data prior of its storage but also decryption after data access. Data integrity to ensure at multiple points the correctness and freshness of the data. Session management that pervades a whole application to expose a context. Digital signature that provides identification and integrity of data. Persistence and transaction are technical elements who are present prior access to a data-layer, and that manage lifecycle of transaction independently from the remaining of the application. Monitoring and custom logging is highly scattered and tangled to the application code. Caching of data objects might appear at different place in the code. The configuration management provides technical code to retrieve and store configuration of an application. Exception treatment are sometimes cross-cutting an application, when the exception are not related to the business code but rather manages technical errors. The management of state such as automata is also a cross-cutting element as the internal logic of the state is not directly related to the application.

There has been several work talking about Aspect-Oriented Programming for security. The premises of the decoupling of security mechanisms from the base programs have been introduced way before the emergence of aspect-oriented solutions. The "metaobject protocol [KDRB91] (MOP) defines specifications of a set of generic functions for accessing and manipulating core structure of an object. It defines the notion of meta classes that represent how a class is structured and behaves. The meta classes are responsible of the overall behavior of an object system. The concepts behind metaobject protocols are intensively used in introspection, in which it is possible to consult information about object's methods, or inheritence structure. Metaobject protocols also define the premises to intercession, which is the ability to modify the behavior of an existing object. The notion of introducing security with meta-object protocols is tackled by Braga *et al.* [BDR00]. They outline a metaobject protocol for secure composition of cryptography-aware meta objects. Their goal is to enable the definition of secure composition in a decoupled manner and orchestrate the cryptographic mechanisms in a control environment. By reducing the coupling in the application, they transparently introduce security in an order they control. For instance, they provide several mechanisms that can be mutually exclusive if introduce at once, and they show that a careful handling at the meta-level allows to control the order of secure composition, as well as the possibility to evolve the application on the fly. In a nutshell, they achieve inversion of control to centralize security requirements in a module through reflection mechanisms.

There are different strategies in which metaobject protocols can be used to define security enforcement. One needs to control how metaobject protocol interact with the base program and control somehow the permissions. The work from Caromel and al. [CV01] proposes to study the possible type of MOP strategies and their implication in security. They assume a component-based application, in which java security applies: protection domains are specified to limit the scope of permissions for principals. According to their classification, it exists four categories of

Meta-Object Protocol (MOP):

- Compile-time MOPs. It reflects on language constructs available at compile-time. The meta-level code is executed at compile-time in order to perform a translation on the source code of a program.
- Load-time MOPS. It reflects on the bytecode and make use of a modified class loader in order to modify the bytecode at the moment it is loaded into the JVM. They operate on bytecode rather than java source code.
- VM-based runtime MOPs. It accesses to runtime information, such as method invocation, read or write operations on fields, etc. It reacts on events to execute meta-level objects.
- Proxy-based runtime MOPS. It introduces hooks into the program to access specific runtime events, such as method invocation.

The type of MOPs depends on the time of reflection shift. The type of MOPs impacts the constraints on permission sets, meaning that the strategy of reflection will impact the propagation of security from the meta-level objects to the base level components.

Viega and al. [VBC01] focuses on the application of aspect-oriented programming to the security domain. They discuss the motivation and principle to achieve consistent security throughout an entire application. As aspect-oriented programming defines a new paradigm to separate cross-cutting concerns, they anticipate simplification in the separation of security module from the base application. The tasks are split with clear separation between stakeholders, security policies can be define once and the system takes care of the coherency and the enforcement within the application. They tackle the language expressiveness of such system: wildcarding that allows one system designer to express point of application of the security modules, context gathering that allows transmission of context from the base program to the security aspect, order of aspect composition that can change the semantic of the application. They list several possibility in applying AOP to security that they partially discusses through C-language example:

- Perform error checking on security-critical callas
- Implement buffer overflow protection, or inserting special code at function entry and exit
- Log data that may be relevant to security
- Replace generic socket code with SSL socket code
- Insert code at startup that goes through a set of lock-down procedures that most programmers would not add to their programs
- Specify privileged sections of a program and request privileges when needed

The transition between meta-object protocols and aspect-oriented programming to tackle security requirements is discussed to learn lessons from MOP and understand how aspects can benefits from such mature discussions [WS02]. The different experiments that have been made with meta-object protocols can be applied to aspects. They both tackle security as a cross-cutting concern, to increase the flexibility of enforcement mechanisms. MOPs correct enforcement depends upon all accesses to the object being controlled by a reference monitor. Such approach might not prevent against local use to bypass reference monitor checks on specific objects. One needs to guarantee the complete mediation of the system to ensure that all calls are correctly redefined through the meta-objects. The implementations should also be protected against tampering: it can be achieved by relying on operating system controls, or using code signature to verify the integrity of the implementation. A last lesson from MOP is the verification process that should allow verification, analysis and testing of modules. Meta-object protocol defines standard and very general interfaces that makes it possible to test them independently of the base level application. The advantages of aspects over MOPs, according to the paper, is that they offer domain specific language. It is easier to define high level application abstractions, that make direct expressive modeling to the security designers. A second advantage is the expressiveness of the weaving language. Pointcuts are expressive and sophisticated. Unlike MOPs which rely on a binding specification language, pointcut languages does not need to introspect the base program to gather the context.

In the following, we introduce several works that also address security with aspect-oriented programming. We distinguish two main categories in the different works: security with AOP, and AOP for security. The main distinction is that the first category introduces security mechanisms using aspect techniques, while the second extends AOP approaches to provide security construct included in the aspect system. The foundation of aspect-oriented programming for application-level security has been debated in a 2004 workshop attached to the main AOSD conference.

2.5.1 Security Engineering With AOP

Security is one of the several domains that aspect-oriented programming aim to address. Researchers started at the early stage of aspect-oriented programming design to think about security ease of management with modular cross-cutting concerns. In this section, we present various works that addresses security engineering, starting from the architecture of secure software systems to the secure programming enhancement for enterprise systems.

In the early workshop on aspect for application-level security, Ron Bodkin introduces a position paper to present enterprise security aspects [Bod04]. The main scenarios the author develop to highlight interesting use case of aspect for enterprise security are authentication and access control properties. The author lists the different scenario that one can encounter in enterprise environment, such as database authentication, role base access control, audit, encryption, filtering, etc. From these scenarios, the author focus on some use cases to outline problems and opportunities in applying aspect to security. He also states few development areas , such as the development of more expressive pointcuts (for predicting control flow and tracking data flow) or extension of the aspect for a better integration in third party systems (distributed systems, external librairies, etc.). Even with the tools available at that time, security with aspects already offers benefits: code is compact, separation of role is clear.

The problematic of security engineering is discussed in Bart de Win's thesis [Win04]. He explains why it is so difficult to address security during the software development process. Security is pervasive: it appears anytime anywhere. Each single piece of code that is written need to be secure, in the sense that a developer would think of all potential situations to avoid abuse of the program. Security also crosscut the application, even tough attempts to modularize security concerns have been going on. On top of this, the recommendation to build secure system require to build it from the very beginning. But, as the author mention, security comes from unanticipated risks and changes. Laws and policies can change, as well as threats within an environment. Changing a part of the application will often lead to a change in security requirements. The focus of the thesis is on the relation between business logic and security requirements, to achieve strict modularization of security. The author compares interception-based and weaving-based techniques, and shows that both approaches provide better security with each merits and deficiencies. More precisely, according to the author, interception is suited for situations that require coarse-grained but flexible composition whereas weaving provides rigorous, but more fine-grained support. The listed advantages of AOSD are numerous: support for advanced modularization leads to easier development with better separation of roles, the security binding is easier to modify and verify as it is centralized, explicit modularization leads to reusability in security mechanisms, and the explicit and late composition strategy of AOSD enables more flexible composition scenarios.

A couple of years before, Wohlstadter and al. [WTD02, WJD03] propose a framework for cross-cutting concerns in distributed and heterogeneous systems. The challenge in such diverse environment is to tackle complex implementation that crosscut the variety of systems, languages, platforms and OSs. From the different approaches already identified to deal with cross-cutting features (namely language-based, middleware-based and container-based approaches), the authors propose an approach that combine pieces of code in heterogeneous systems. They provide cross-cutting functions called *adaplets*, that are placed at the points where the application interacts with the middleware. In the course of their examples, they provide reliability, performance, and security use cases. The adaplets propose contract for client and server extension, and reuse partially notions of the AspectJ language. For instance, a service architect can declare a certain number of advices to be available on client side or on server side, declare pointcuts, etc. The authors also defined several features for the approach: modeling, type-checking through an enhanced-IDL and code generation. Their pointcut-based binding allows static transformation (weaving) or dynamic wrapping. Finally, to address the distributed and heterogeneous aspect, adaplets on client-side and server-side communicate through CORBA messages to exchange information. In the scope of security, they can transparently introduce security protocol to request for request authorization to perform some client actions.

Some efforts have been made to modularize security and provide maximal resuability. Hence, Huang and al. [HWZ04] propose security functions in a reusable and generic security aspect library. The goal is to provide the basic reusable components developers need in the phase of security implementation. The library is called JSAL and is implemented in AspectJ. It provides four independent categories of security aspects: encryption/decryption, authentication, authorization and security audit. This library encapsulate code that refers to the numerous security packages of java security, such as Java Cryptography Extension API (JCE) and Java Authentication and Authorization Service (JAAS). The security behavior is encapsulated in abstract classes, letting the developer determine the exact pointcut on application's integration. Whereas the contribution represent a first step towards reusable and generic security aspect library, the authors lack of a method to address evolution of the application, as well as evolution of the security aspects.

Aspect-Oriented security is debated in a work that compares advantages and disadvantages of the approach with container-based security [SZ03]. It augments the flexibility of security in environments that are container-based, such as application servers. Security in application servers is generally provided and managed by containers. More precisely, the different components representing the application do not have to execute specific security actions when the container provide the security for them. The security is centralized in container's configuration. The application server typically provides identification of a principal and its authentication, that does not need extra code. While addressing access control (role-based authorization for example), the container might provide some functionality, but generally needs modification in the components' code. The paper shows that container provide no support or standard solution for accountability in audit. Therefore, additional code in components is to be provided. The contribution shows that identification and authentication of a principal in a typical J2EE application can be modularized with few efforts in an aspect. The access control follows the same behavior. The accountability and audit of the code can also be put in cross-cutting modules with no components' code modification. The authors state that combination of container-based security and aspect-oriented security is complementary for maximal flexibility.

Industrial approaches of aspect leads to the development of new languages for specific needs. AspectJ2EE [Coh04, Coh07] is a language developed along a thesis that address the problematic of cross-cutting concerns in middleware framework such as application servers. The language is voluntary less general and complicated than AspectJ, and focus on the large scale distributed applications. It introduces parameterized aspects for flexibility and reusability of aspects. The aspect's code is weaved in a novel stage: deployment of the application. The authors claim several advantages to this approach: preserving the object model, better management of aspect applicability, and semantic model is more understandable hence maintainable. The approach proposes a standard library of core aspects:

- Lifecycle aspect
- Persistence aspect
- Security aspect
- Transaction aspect

Aspect for java security has been discussed in a master thesis [Far01]. Java security relates to the security model of the java language and java execution platform. The work address java application like applets, that run through several layers of protection to provide features to secure the environment against trusted or untrusted applets running on local machine. The author define a security aspect for java to define strong security models through security policies and specifying security restrictions.

The notion of decoupling security concerns to the application and architecting secure software systems using aspect-oriented approach is discussed in a survey [DS06]. The survey provides an exhaustive list of the different approaches. As the paper expresses, the properties to respect when designing, developing and implementing secure system in which developers and software engineers aim to produce secure systems are:

- The security-related properties in a system should be abstracted out of the main system to improve clarity, maintainability, manageability and reuse.
- Legacy source code with known or potential security vulnerabilities should be able to be patched with a minimal amount of new code. It should also be possible to avoid modifying the original code.
- When applicable, security-related properties should be reusable across different applications.

The survey discusses the different approaches, including design solutions, architecting frameworks, approaches with secure coding, etc. In the discussion part, the author expresses a statement that security cross the different layers of the computing stack, and AOP can help address the concern across layers. Also, security is not a localized concern. It is a distributed concern present across the network. Aspects can help in addressing distributed concerns, like mentioned in [NSV$^+$06].

2.5.2 Secure AOP

In this section, we present several approaches that extend the general aspect-oriented approach to propose security as feature built-in in the aspect parts. The solutions develop specific languages or techniques.

Under the supervision of Mourad Debbabi, two thesis propose an aspect-oriented framework for security hardening. Although one is oriented on an applied approach [Mou08], the other poses semantic foundation [Bel08]. In the course of the thesis, they develop a complete framework to overcome the difficulty of AOP usage in software development for systematic security hardening. They propose a pattern-based approach that limits the need of high expertise in security to secure software. They also propose a programming-independent language called SHL to propose security hardening plan and security hardening patterns, that combined together creates security hardening aspects.

They have defined new pointcut primitives to augment expressivity of pointcuts. For instance, they define pointcut to match data-flow, but also points relative to control-flow:

- GAFlow In the scope of control flow graph, which are potentially cyclic directed graph that represent the calling structure of a program, GAFlow computes from a list of joinpoints a unique joinpoint. The output joinpoint is "the closest common ancestor that is (1) the closest common parent node of all the nodes specified in the input set and (2) through which all the possible paths that reach them pass."
- GDFlow. It also operates on control flow graph and computes a unique joinpoint from a list of joinpoints. The output joinpoint "(1) is the common descendant of the selected nodes and (2) constitutes the first common node reached by all the possible paths emanating from the selected nodes."

They present several primitives that are useful for security. The primitives have been suggested in different works

- Predicted Control Flow [Kic01] pointcuts identify join points based on the predicted behavior at the current join point.
- Dataflow Pointcut [MK03]. The pointcut identifies join points based on the origins of values.
- Loop Pointcut [HG06] is a loop join point model that demonstrates the need for a more complex join point in AspectJ. Their approach to recognize loops is based on a control-flow analysis at the bytecode level.
- Pattern Matching Wildcard to perform pattern matching.
- Type Pattern Modifiers. Patterns are used inside primitive pointcut designators to match signatures and consequently to determine the required join points
- Local Variables. It would gives access to local variable that are created within a method, and not only parameter or return object of a function.
- Synchronized Block Joinpoint.

Belblidia proposes a semantic for the java virtual machine language bytecode [Bel08], and a semantic for the AspectJ weaving. Then, she provides at the practical level an implementation for two new AspectJ pointcut: getlocal, setlocal and dflow that matches local variable in function and dataflow in the information flow.

A recurrent use case for AOP is the access control strategy. Mariscal and al. [PMMD05] formalize a compilation mechanisms for security specifications. It translate representation of role-based access control in aspect-oriented code for security enforcement. The formal model gives some basis to evolve the model in future. The model translates role slices, which is a record of the permissions for methods in a system. The translation outputs a policy database, and an access control aspect for the enforcement part. The control usage is also part of multilevel security strategy. Aspect-oriented has been applied to usage control [PE07] to facilitate the introduction of the security logic with a non-intrusive approach. The claim is that this technique allows to abstract the different features one need to put in the application, which ease the management and evolution of the features.

Sewe and all. [SBM08] worked the applicability of several aspect-oriented languages to effective java security, regarding the security model in java language. They indicates that security with AOP is not trivial, and that the languages does not fully take the security model in account. They also argue security implications in case of inability to address some issues related to the class loading. A protection domain may be erroneously assigned when advice is inlined in the application, and the separation of namespace can not be guaranteed.

We have seen that many extensions are proposed to better control application of aspects within a base program. In addition to verify that pointcut languages are expressive enough to describe cross-cutting concerns, one need to provide proof of correct mapping of aspects in the program. The work of Balzarotti and al. [BM04] propose an analysis of aspect-oriented composition using program slicing.

All these works demonstrate an high interest in securing systems using the aspect-oriented paradigm. The works tackle the different aspects of software engineering, to either enhance design of application with the numerous cross-cutting concerns, either to enhance the applicability of aspects solutions in system with a fine-grained control of side effects.

2.6 Service Oriented Architecture

The thesis deals with distributed systems, although we are mainly referring to the service oriented architectures. In few paragraphs, we introduce the terminologies we are using in the thesis. Service Oriented Architectures (SOA) enable a world of loosely-coupled and interoperable software components towards reusability. Nowadays, the main entity used to represent a software service is a Web Service. Web-Services represent a paradigm defined by W3C as "a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, conveyed using HTTP with an XML serialization in conjunction with other Web-related standards" [BHM$^+$04]. Web Services can also be addressed through other transport mechanisms such as JMS or ESBs. The Web Service standards stack goes beyond the atomic service, and proposes different approaches depending on the level of abstraction. Service behavior can be defined when linking different services together, *e.g.,* with BPEL4WS or BPMN 2.0 [HS04]. It allows definition of service composition to realize a so-called business process.

In the following, we describe standards that are commonly used with Web Services, namely WSDL and SOAP for WS-* related standards. We also present RESTful services that are getting widely used for lightweight resource consumption and other means.

2.6.1 WSDL

WSDL 2.0 is a language based on the XML format, which provides a model for describing Web services [CMRW07] . Means for expressing service interfaces are at the core of all service models, and WSDL provides very flexible, highly-extensible, and well designed methods for doing this. This description is done in two fundamental stages: an abstract and a concrete one.

At an abstract level, WSDL 2.0 provides the structure description of the messages sent to and received by a Web service, such as data types, messages patterns and method description. "An operation associates a message exchange pattern with one or more messages. A message exchange pattern identifies the sequence and cardinality of messages sent and/or received as well as who they are logically sent to and/or received from. An interface groups together operations without any commitment to transport or wire format".

At a concrete level, "a binding specifies transport and wire format details for one or more interfaces. An endpoint associates a network address with a binding. Finally, a service groups together endpoints that implement a common interface." It means WSDL contains information of how messages are mapped to a concrete network protocol - a so-called binding - so that these messages can be exchanged in an interoperable fashion.

2.6.2 SOAP

SOAP is "a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. It uses XML technologies to define an extensible messaging framework providing a message construct that can be exchanged over a variety of underlying protocols" [GHM+07].

The protocol specifies the exchange of structured information in the implementation of Web services. SOAP uses the HTTP for RPCs, hiding the HTTP semantics from SOAP applications. In fact, ”SOAP treats HTTP as a lower-level communication protocol” and uses its own semantics [Man05]. The major goal while designing SOAP have been simplicity and extensibility. The focus has been put on specifying a model for message exchange and operation execution with no specific treatment of reliability, security or other concerns that are normally directly addressed in distributed protocols. Nevertheless, these concerns are covered thanks to the extensibility.

2.6.3 REST

The term REpresentational State Transfer (REST) was coined by Roy Fielding in his PhD dissertation [Fie00]. "REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems". R. Fielding describes the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, contrasting them to the constraints of other architectural styles.

In REST, everything is a resource. A resource can be thought of as a distant object one can interact with, but not manipulate directly. This is similar in spirit to object oriented programming where everything is an object, but the approach is fundamentally different. Every resource, identified by a unique identifier, is interacted with using a universally predefined set of verbs. These verbs are defined for every resource globally. On the web, the unique name is the Uniform Resource Identifier (URI) and the verbs are the standard HTTP methods as POST, GET, PUT and DELETE. The research community is having several opinions on how these methods can be mapped to the CRUD operations. For instance, these methods can respectively associated with the CREATE, READ, UPDATE and DELETE operations. Each method has clear defined semantics that can be relied upon.

2.6.4 WS-* Security

The most common nowadays is the primitive stack associated to security model named WS-Security. This standard provides quality of protection through integrity, confidentiality and authentication on messages with SOAP messaging enhancements. It allows one to sign and encrypt part of messages, that are at the XML Format to have a fine-grained control on a end-to-end communication. Credentials are then transmitted securely in the form of security tokens.

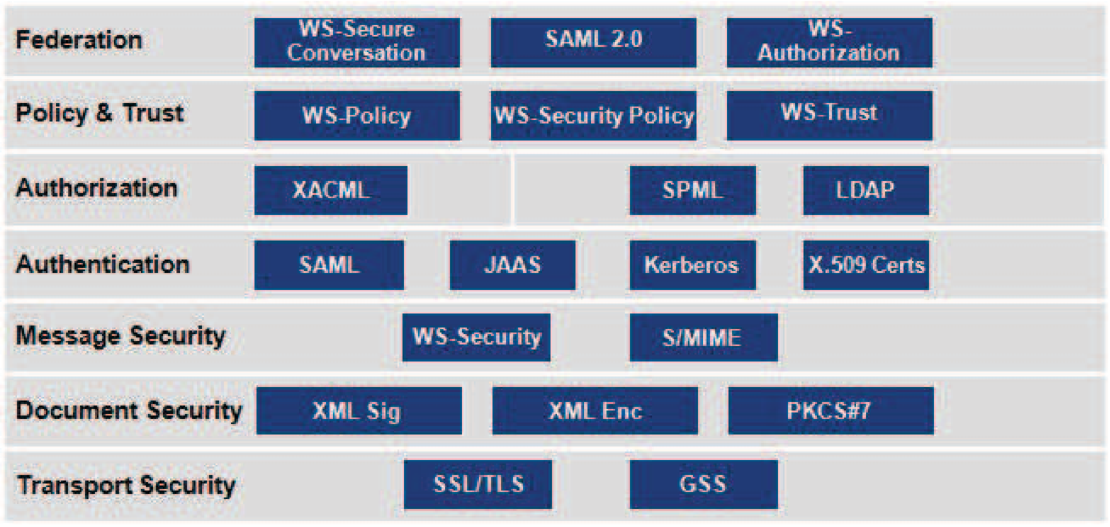


Figure 2.6: WS-Security standards

The initial roadmap has been submitted in 2002 to the OASIS consortium and the first version was ratified in 2004, with a version 1.1 in 2006. The core standards use several token profiles, such as UserName Token Profile [LKa06c], X.509 Token Profile [LKa06d], Kerberos Token Profile [LKa06a], and SAML Token Profile [LKa06b]. These tokens are the different means to serialize and transmit credentials across platform in a consistent manner.

Using WS-Security allows one to cover different scenarios with a same standard, such as providing an end-to-end security instead of a point-to-point. The latest release improves performance as all security mechanisms are ported to the Transport Layer (mostly TLS) or even the communication protocol (IPSec for example). The drawbacks are a less precise and fine-grained control on what is being transmitted. With a point-to-point protection, one can intercept a message in plain text before final delivery of the application and modify it without further detection. Also the transport layer protection provides security at the transport level rather than at a message level and allows to encrypt and sign only necessary elements in a large XML-document set. WS-Security can then be viewed as run-time declaration of how content is formatted, and what steps are required to process it, during messages exchange.

The web-service stack also proposes a declarative approach during the modeling phase. In a typical SOA, where the client and the service may not be in the same security domain, policies enforce security rules on the outgoing (client side) and incoming (service) messages. WS-SecurityPolicy [LKa09] is an OASIS standard. It describes how senders and receivers can specify their security requirements and capabilities. For example, a service can specify that it requires a SAML token and signed message in the incoming SOAP request. WS-Security Policy is based on WS-Policy (a W3C Recommendation). WS-Policy is fully extensible and does not place limits on the types of requirements and capabilities that may be described. It also defines a mechanism for attaching or associating service policies with SOAP messages.

Part I
Modularization of defensive security

The modularization of defensive security can be simplified to the evaluation and enforcement of secure programming best practices. In reality, best practices are a set of methodologies, processes, rules, concepts and theories that have to be properly defined and exchanged. All people involved in the definition and implementation of a system have to share the same overall vision in term of security, and also to up-scale in order to detect and react to new threats. There are several sources to exchange such knowledge, with different scopes. Government and cross-national institutions are publishing guide for information security, or IT security guidelines. For example, the french government has a dedicated structure [Age13] to propose documents in french with guides, news around information security. The German government proposes a similar structure in quality of the "Federal Office for Information Security" [Bun08]. Europa has an agency to improve network and information security across europe nations. It publishes guides such as security awareness [Eur09] to the destination of citizens, businesses, public sector, *etc.* One of the most significant worldwide institute is the National Institute of Standards and Technology [oST] (NIST). It provides guidance for several field of engineering with information security among the covered fields. Independent institutions also exists, such as the Information Security Forum since 1989 dedicated to investing, clarifying and resolving key issues in information security and risk management, by developing best practice methodologies, processes and solutions. They have published guides such as *Standard good practices for information security* [Inf12] to the benefits of they members, world-leading organizations and businesses.

In addition to all these institutions, there are groups that structured themselves to pursue a common goal of knowledge sharing and centralization of best practices for specific areas. In the area of internet security, there are groups such as the CERT Coordination Center that provides approach for better system security, such as tools and techniques for threat and vulnerability evaluation [CER]. The CERT was initially a team at Carnegie Mellon University, but now designate a team dedicated to respond to internet and computing incident. OWASP [OWA, Fou] is an open-source project for web-application security. It is a community for corporations, academia, and individuals with several regional events to share knowledge and understand latest threats. Its scope limits to web-applications and related. Another organization promotes the use of best practices for providing security assurance in cloud-computing: the Cloud Security Alliance organization. [Clo09]. They publish documents specific to cloud security, such as security guidance.

Although numerous organizations edit guides for proper information security management, security vulnerabilities are still prevalent in applications, and specifically web and cloud applications. These type of applications are specially exposed to a larger audience with high press coverage. They interact with users and collect large set of data subject to privacy, thus regulations. Developing such type of applications require carefulness in specification of the application, then rigorous enforcement of security properties. Finally, application needs built-in quality to remove from the development stage any vulnerability that can be exploited later on. In term of development, there are several strategies to ensure working and safe applications, that highly depends on IT environment: frameworks, libraries, programming language, application servers, *etc.*. These pieces provide advantages and drawback. It is generally a trade-off between flexibility, performance and security. Whereas flexibility is given at some points, rigorous verification needs to propagate. For instance, a web application taking input from its users will have to properly verify the data, and format it regarding the actual usage. Modularization of security helps to understand when security behavior is applied within the components of the application. These are additional points of control injected in the application to properly ensure the non-exploitation of the application by non-authorized users for example. In the following, we present a first work where we leverage user's development environment to detect and protect against common vulnerabilities in web applications. Then, we propose another approach that generalizes and render flexible the automation of input validation in applications.

Chapter 3

Vulnerability remediation with modular patches

Security vulnerabilities are commonly encountered in systems despite the existence of best practices for several decades. In order to detect the security vulnerabilities missed by developers, complex solutions are undertaken like static analysis, often after the development phase. The problem of solutions after the development resides in the loss of context: people in charge of correction have all the difficulties to understand the architecture and the reasons behind particular pieces of code. Although vulnerabilities are found, there is also an absence of systematic protection against them. In this chapter, we introduce an integrated Eclipse plug-in to assist developers in the detection and mitigation of security vulnerabilities using Aspect-Oriented Programming early in the development life-cycle. The work is a combination of static analysis and protection code generation during the development phase. We leverage the developer interaction with the integrated tool to obtain more knowledge about the system, and to report back a better overview of the different security aspects already applied. We discuss the challenges for such a code correction approach. The result is a solution to assist developers in order to obtain software with higher security standards. The whole solution, combining static analysis and remediation, proposes a better approach in terms of integrated security with clear modularization of security code. It contributes to the secure programming best practice enforcement, with enhanced flexibility.

3.1 Introduction

After more than decade of existence, cross-site scripting (XSS), SQL Injection, and other of types of security vulnerabilities associated with input validation can cause severe damage once exploited. Scholte *et al.* [SBK11] conducted an empirical study that shows that the number of reported vulnerabilities is not decreasing. Listing 4 provides an example of how a developer can introduce a vulnerability within a few lines of code.

Listing 4 Vulnerable java code in a servlet class.

```
package com.sap.research.nce;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.json.JSONException;
import org.json.JSONObject;

/**
 * Servlet implementation class RestSevices
 */
public class RestServices extends HttpServlet {

    protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
        System.out.println("doGet on RestServices");
        PrintWriter writer = response.getWriter();

        String title = request.getParameter("title");
        long seed;
        try {
            seed = Long.parseLong(request.getParameter("seed"));
        } catch (NumberFormatException e) {
            seed = 1000;
        }

        //potential SQL Injection
        Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/bigbro");
        Statement statement = con.createStatement();
        String parameter = request.getParameter("parameter");
        String query = "SELECT * from user where ID='"+parameter+"'";
        statement.executeQuery(query);

        try {

            JSONObject root = new JSONObject();

            JSONObject obj = new JSONObject();
            obj.accumulate("serviceName", "BarChartService");

            Random rand = new Random(seed);
            JSONObject objarray;
            for (int i = 0; i < 3; ++i) {
                objarray = new JSONObject();
                objarray.accumulate("year", 2000 + i);
                objarray.accumulate("problem", rand.nextInt(101));
                objarray.accumulate("return", rand.nextInt(101));
                objarray.accumulate("buy", rand.nextInt(101));
                obj.accumulate("sales", objarray);
            }

            response.setContentType("application/json");
            root.accumulate("Result", obj);
            writer.print(root.toString(2) + "{ \"title\" : ");
            //XSS vulnerability at this point
            writer.print(title);
            writer.print("}");

        } catch (JSONException e) {
            e.printStackTrace();
        }finally{
            writer.flush();
            writer.close();
        }
    }

}
```

The Listing presents two distinct vulnerabilities. From line 32 to 35, the application get a parameter from a HTTP request, and uses it to construct manually a query. The problem occurs as there is no verification of the input, and the query is executed as is. A malicious user can forge the parameter to control the result of the query. These lines have no consequences on the direct output, but one can still inject data or control the underlying database. The second vulnerability is present at line 58. A JSON object is populated with values coming from an HTTP parameter, and the object is written to the response with no particular treatment. One can control the parameter and exploit this XSS vulnerability.

While computer security is primarily a matter of secure design and architecture, it is also known that even with the best designed architectures, security bugs will still show up due to poor implementation. Thus, fixing security vulnerabilities before shipment can't be considered optional anymore. Most of the reported security vulnerabilities are simply forgotten by developers, thought to be some benign code. Such mistakes can remain unaudited for years until they end up being exploited by hackers.

The software development lifecycle introduces several steps to audit and test the code produced by developers in order to detect security bugs,ranging from code review tools for early detection of security bugs to penetration testing. The tools are used to automate some tasks normally handled manually or requiring complex processing and data manipulation. They are able to detect several errors and software defects, but developers have to face heterogeneous tools, each one with a different process to make it run correctly, and they have to analyze their results, merge them, and fix the source code accordingly. For instance, code scanner tools are usually designed to be independent from the developers' environment. Therefore, they gain in flexibility but lose comprehensiveness and the possibility to interact with people experienced with the application code. Thus, tools produce results that are not directly linked to application defects. For example, code scanner tools trigger false positives, which are not actual vulnerabilities.

Our contributions are twofold. First, we focus on static code analysis, an automated approach to perform code review integrated in developer's environment. This technique analyzes the source code and/or binary code without executing it and identifies anti-patterns that lead to security bugs. We focus on security vulnerabilities caused by missing input validation, the process of validating all the inputs to an application before using it. Although our tool handles other kinds of vulnerabilities, here we discuss on three main vulnerabilities caused by missing input validation, or an incorrect validation of the input: cross-site scripting (also called XSS), Directory Path Traversal, and SQL Injection. Second, we provide an integrated assisted remediation process that employs Aspect-Oriented Programming for semi-automatic vulnerability correction. The combination of these mechanisms improves the quality of the software with respect to security requirements.

Figure 3.1 presents the interaction between the two phases: the static analysis phase allows scanning the code in order to identify and classify the different vulnerabilities found. It is described in details in Section 3.4. The measurement is performed directly by developers who decide what to remediate directly within the development environment. The full remediation process is given in Section 3.5 .

We present the contribution in following sections: Section 3.2 presents the overall agile approach to conduct code scanning and correct vulnerability during the development phase.

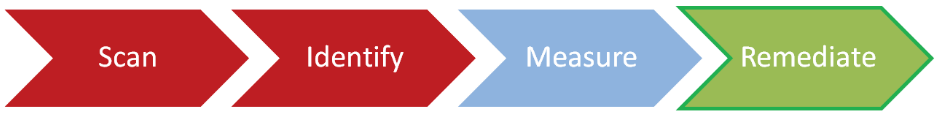


Figure 3.1: Vulnerability remediation process. The two first blocks correspond to the static analysis component. The two last blocks correspond to the remediation component. The last one corresponds to assisted processing component

Then, Section 3.3 presents the architecture we adopt to combine the static analysis with the code correction component. Section 3.4 describes the static analysis process with its integration in the developers' environment. Then, we explain techniques for assisted remediation along with pros and cons in Section 3.5. Finally, we discuss an evaluation of the methodology compared to other solutions in Section 3.6, and we present a summary in Section 3.8. This work has been awarded in [SGEKSDO12].

3.2 Agile management of vulnerabilities

Agile approaches to software development require the code to be refactored, reviewed and tested at each iteration of the development lifecycle. While unit testing can be used to check the fulfillment of functional requirements during iterations, checking emerging properties of software such as security or safety is more difficult. We aim to provide each developer with a simple way to do daily security static analysis on his code. That would be properly achieved by providing a security code scanner integrated in the development environment (we selected Eclipse IDE in our case), and a decentralized architecture that allows security experts to assist the developers. Typically, that would include verifying false positives and correspondingly adjusting the code scanner test cases, or assisting in reviewing the solutions for the fixes. It brings several advantages over the approach in which the static analysis phase takes place only at the end. The expertise of the context in which the code was developed lies in development groups. Therefore, the interaction between development team and security experts makes it faster and easier to find and to apply corrections to the security functionalities. The experts provide support on a case-by-case basis for a better tuning of false positive detection across teams and reducing final costs of maintenance: solving security issues into the development phase can reduce the number of issues that the security experts should analyze at the end.

Maintaining the separation of roles between the security experts performing the code scanning and the team members developing the application raises a critical complication, typically, from a time perspective, due to the human interaction between security experts and developers. If such an approach would have to scale to what most of the agile approaches describe, the amount of iterations between developers and experts would need to be reduced. That could be reduced by up-skilling the developers and reducing their interactions with the security experts for the analysis of the security scans of the project, which is simplified by the introduction of our tool.

Our incentive is to harvest the advantages acquired by using our approach in an agile and decentralized static analysis process early in the software development lifecycle. It raises security awareness for the developers at the development time and reduces maintenance costs. A tool covering the previous needs should fulfill several requirements:

- *easy-to use* for users non-expert in security
- *domain specific* with integration into the developers' daily environment, to maximize its adoption and to avoid additional steps to run the tool
- *adjustable* to maximize project knowledge and reduce false positives and negatives
- *collaborative feedbacks* to adjust accuracy of the scan over time.
- *supportive* to assist developers in correcting and understanding issues.
- *educative* to help developers understanding errors, steps to correct existing error, and techniques to prevent future vulnerability

We have developed an Eclipse plugin, presented in [GEKS11], made of components leveraging an agile and decentralized organization for static analysis. It gives direct access to detected flaws and global overview on system vulnerabilities. The developer analyzes its code and reviews vulnerabilities when necessary.

3.3 A flexible architecture

Figure 3.2 represents the architecture of our prototype. We consider two main stakeholders involved in the configuration and usage of the prototype. Security experts and developers have to communicate and collaborate. Their role is to configure altogether the knowledge database in order to avoid false positives and negatives, and to provide better accuracy during the analysis phase. The security experts have two main tasks. First, they update the knowledge base, adding to its classes or methods that can be considered as trusted for one or more vulnerabilities. Second,they analyze queries from developers to enhanced the knowledge database . The analysis is on possible trusted objects for one or more security vulnerabilities; they must analyze them more in detail and, if these objects are really trusted they tag them as trusted into the knowledge base. We better explain the different concepts and tasks in Section 3.4.

The second role is the developer, interacting directly with the static analysis engine to verify vulnerabilities in the application code and libraries under its responsibility. The remaining libraries have to be covered by the security experts group. The developer at this stage doesn't need to understand the complexity of security properties. The knowledge base is shared among developers. It contains all the security knowledge about trust: objects that do not introduce security issues into the code. Security experts and developers with understanding of security patterns maintain and keep under control the definitions used by all developers in an easy way using one admin web application or some web-services. In this way, the code scanner testing

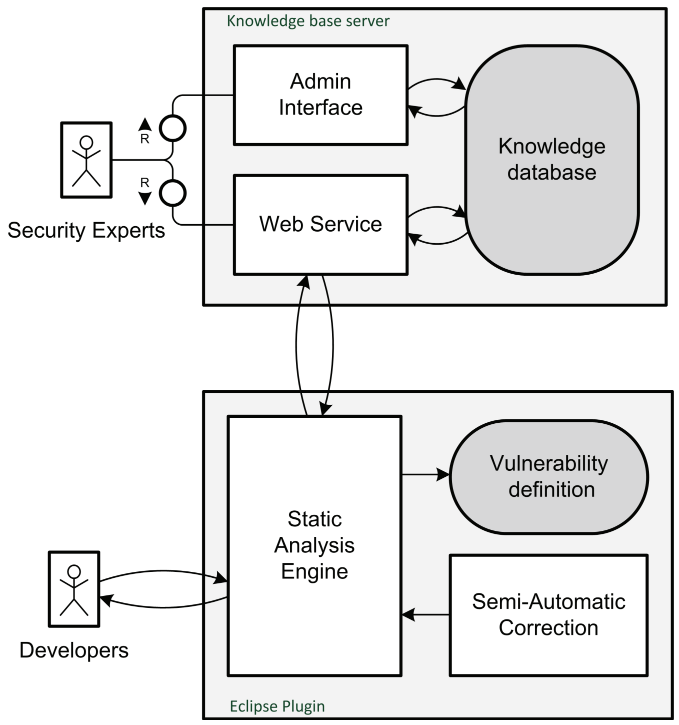


Figure 3.2: Architecture

rules are harmonized for the whole application or even on a project-basis. The knowledge base allows developers to run a static analysis that is perfectly adapted to the context of their project.

In industrial scale projects, daily scans are recommended. In order to facilitate this task, we wrote a plugin for Eclipse that uses an abstract syntax tree (AST) generated by the JDT compiler - the compiler that Eclipse provides as part of the Java Development Tools platform, to simplify the static analysis process. The plugin accesses the knowledge database via web-services making it possible for each developer to run the code scanner independently. We detail the scanner component in the next section.

The overall process is to find in applications where security checks are not performed, or not properly applied, leading to potential vulnerabilities. We use a specific taxonomy to describe security related code concepts *Entry points* are points from an application where injection during an attack can start. These are points that are not directly created by the application, but rather points opened to external components, thus susceptible to bring untrusted data and objects within the application. These entry points are present in external libraries, web applications dealing with forms and parameters, databases accesses, file system interactions, application arguments, property files, objects from web services, *etc.* In contrast, *exit points* are points of the application where data reach an outside domain, not under the control of the application anymore. These exit points are the result of application execution, like web pages, database writes, filesystem writes, *etc.* As we are going to see in the next sections, the purpose of our approach is to find in the control flow connecting entry points to exit points, which ones are introducing or propagating objects with potential leaks. A vulnerability is declared when an these objects are connected to an exit point. These objects will later be described as untrusted objects.

3.4 Static analysis process

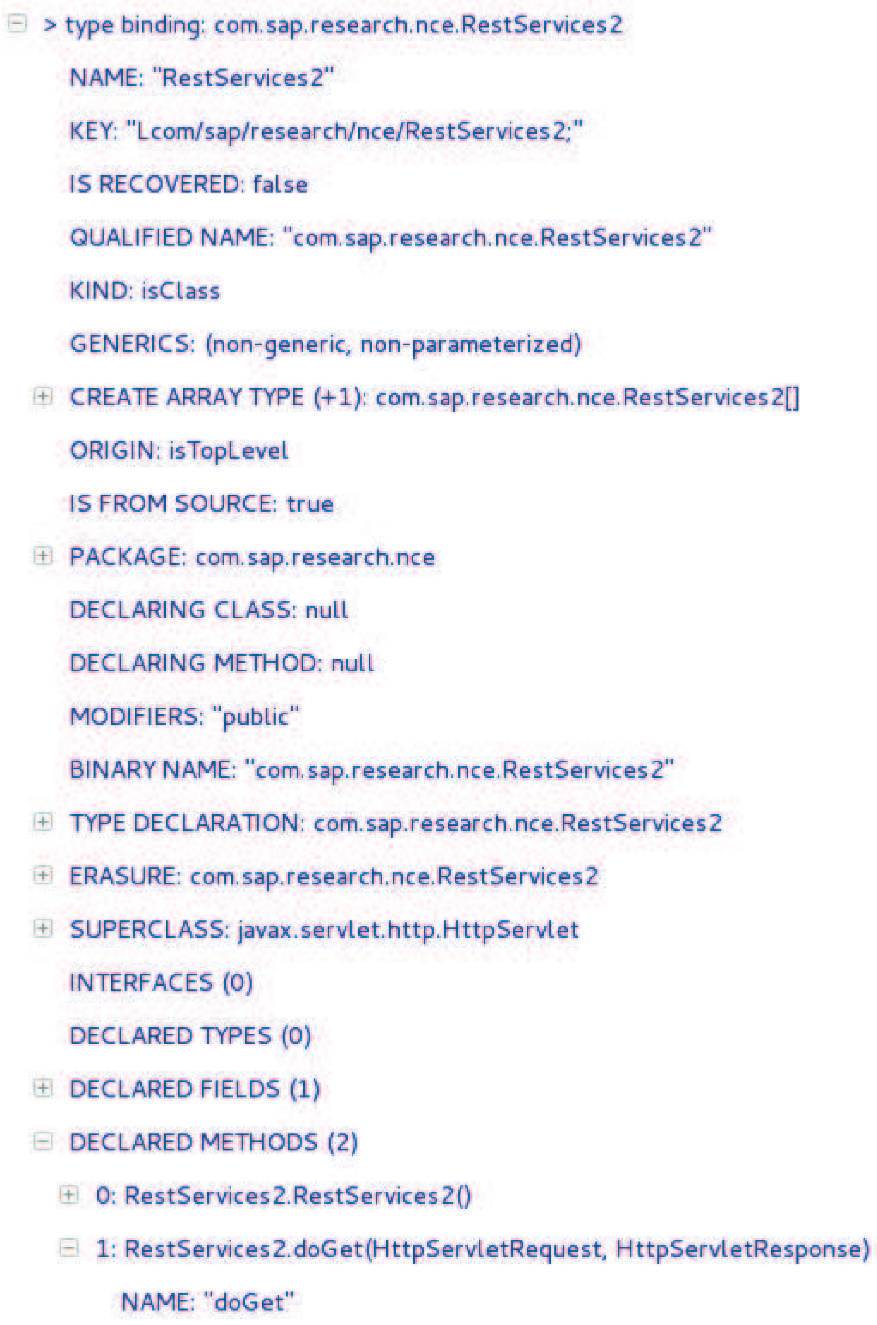
Static analyze can report security bugs even when scanning small pieces of code. Another family of code scanners is based on dynamic analysis techniques that acquire information at runtime. Unlike static analysis, dynamic analysis requires a running executable code. Static analysis scans all the source code while dynamic analysis can verify certain use cases being executed. The major drawback of static analysis is that it can report both false positives and false negatives. The former detects a security vulnerability that is not exploitable, while the latter means that it misses to report certain security vulnerabilities. Having false negatives is highly dangerous as it gives one a sense of protection while vulnerability is present and can be exploited, whereas having false positives primarily slows down the static analysis process. Modern static analysis tools, similarly to compilers, build an abstract syntax tree that represents the abstract syntactic structure of the code from the source code and analyze it.

3.4.1 Static analysis process

In a nutshell, our process allows developers to run a check on their code to uncover potential vulnerabilities by checking for inputs that have not been validated. It finds information flows connecting an entry point to an exit point that does not use a trusted object for the considered vulnerabilities. The process uses an abstract syntax tree of the software in conjunction with the knowledge base to identify the vulnerable points. The Figure 3.3 shows an excerpt of an AST tree for a class type. It gives a structured tree that one can handle with all required information regarding the syntax of a program. Each object has an object type. This type has its own information, such as parameters for a method, or interface information for a type, etc.

The static analysis works on the Document Object Model (DOM) generated by the Eclipse JDT component able, which can handle all constructs described in the Java Language Specification [GJSB05]. Figure 3.4 presents the different analysis steps performed from the moment developer presses the analysis button to the display of results. The static analysis process is described as follows:

- The engine contacts the knowledge database in order to retrieve the up-to-date and most accurate configuration from the shared platform. If the developer cannot retrieve the configuration, it can still work independently with the latest local configuration.
- The process identifies all *entry points* of interest in the accessible source code and libraries. The analysis is based on the previously mentioned AST. We are gathering the different variables and fields used as well as the different methods. We apply a first filter with pattern-matching on the potential *entry points*: a method call or a new object instantiation might be tagged as returning trusted inputs.
- For each *entry point*, the control flow is followed to create the connections between methods, variables and fields to discover all the *exit points*. For instance, the engine visits



```
> type binding: com.sap.research.nce.RestServices2
   NAME: "RestServices2"
   KEY: "Lcom/sap/research/nce/RestServices2;"
   IS RECOVERED: false
   QUALIFIED NAME: "com.sap.research.nce.RestServices2"
   KIND: isClass
   GENERICS: (non-generic, non-parameterized)
   CREATE ARRAY TYPE (+1): com.sap.research.nce.RestServices2[]
   ORIGIN: isTopLevel
   IS FROM SOURCE: true
   PACKAGE: com.sap.research.nce
   DECLARING CLASS: null
   DECLARING METHOD: null
   MODIFIERS: "public"
   BINARY NAME: "com.sap.research.nce.RestServices2"
   TYPE DECLARATION: com.sap.research.nce.RestServices2
   ERASURE: com.sap.research.nce.RestServices2
   SUPERCLASS: javax.servlet.http.HttpServlet
   INTERFACES (0)
   DECLARED TYPES (0)
   DECLARED FIELDS (1)
   DECLARED METHODS (2)
     0: RestServices2.RestServices2()
     1: RestServices2.doGet(HttpServletRequest, HttpServletResponse)
        NAME: "doGet"
```

Figure 3.3: AST view example of a class

assignments, method invocations, and construction of new objects with the variables and fields detected during the entry point gathering.

- Once the different *exit points* have been collected, we evaluate the risk of having security

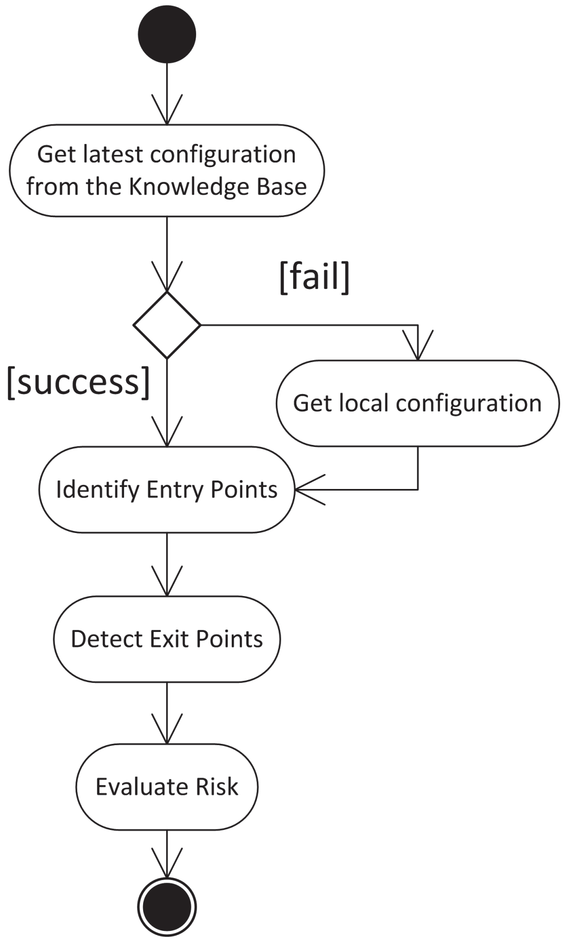


Figure 3.4: Static Analysis Activity Diagram

vulnerabilities in the code. We check for an absence of validation in the flow for the different kinds of vulnerabilities. For instance, if the flow from an entry point to an exit point passes through a method or a class, which is known to validate SQL input, the flow is tagged as trusted for this specific vulnerability. Of course, the tag runs from the moment where the method validates for the vulnerability to a novel composition with potential vulnerable code, or until it reaches an *exit point*.

3.4.2 Multiple vulnerability analysis

In the previous section, we have presented the global analysis process. In this section, we discuss more in-depth the notion of trusted object and vulnerability propagation for the different vulnerabilities we address. The Listing 3.1 presents some source code vulnerable to cross site scripting. The vulnerability propagates from the request parameter to the object *query*, which is then written in the response. The problem of identifying security vulnerabilities caused by errors in the input validation can be translated into finding an information flow connecting an *entry point* and an *exit point* that does not use a *trusted object* for the considered vulnerabilities.

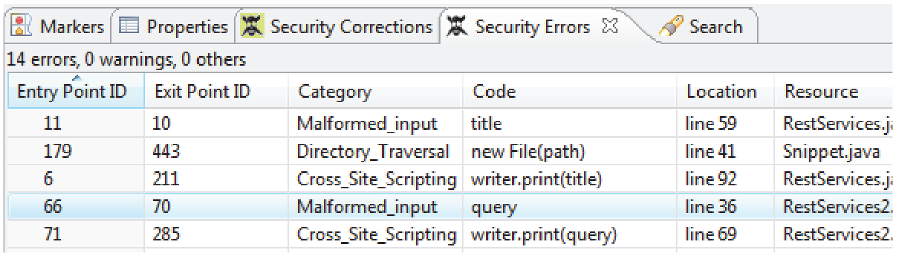
```
/** This servlet proposes XSS example. */
public class EchoServlet extends HttpServlet {
  protected void doGet(HttpServletRequest req, HttpServletResponse resp) {
    PrintWriter writer = resp.getWriter();
    String query = req.getParameter("query");

    resp.setContentType("text/html");
    writer.print("<html><h1>Results for ");
    writer.print (query);
    writer.print("</h1></html>");
    writer.flush();
    writer.close();
    }
}
```

Listing 3.1: Vulnerability propagation of a cross site scripting

We define an *input* as a data flow from any class, method or parameter into the code being programmed that is external from the application. We also define as *entry point* any object into the source code where an *untrusted input* enters to the program being scanned, like the *query* input from Listing 3.1. In an analogous way we define as *output* any data flow that goes from the code being programmed into external objects or method invocations. Our approach relies on our *trusted object* definition, which impacts the detection accuracy. A *trusted object* is a class or a method that can sanitize all the information flows from an *entry point* to an *exit point* for one or more security vulnerabilities. We implemented the trust definitions into the centralized knowledge base presented in the previous section. The knowledge database represents definitions using a trusting hierarchy that follows the package hierarchy.

Security experts can tag classes, packages or methods as trusted for one or more security vulnerabilities, according to their analysis, feedbacks from developers, or static analysis results. Obviously, defining a trusted element in the trust hierarchy also adds all the elements below it: trusting a package trusts all the classes and methods into it and trusting a class trusts all the fields and methods in it. A trusted object can sanitize one or more security vulnerabilities (e.g., the sanitization method can be valid for both SQL Injection and cross site scripting). This approach enables developers and security experts to define strong trust policies with regards to the system they are securing.

Defining a *trusted object* is a strong assertion as it taints a given flow as valid and free from a given vulnerability. The definition process to trust a class, a package, or a method must be supervised: it influences the risk evaluation accuracy. The object must not introduce a specific vulnerability into the code. This is the reason why developers report feedback and security experts take the decision. The experts can also analyze, manage, and update the base if the class, package or method is considered trusted. This phase allows system tuning that is related to a given organization and leads to fewer false positives. There should have no false negatives, as our approach will first detect all possible problems, that are eliminated with fine-grained control with the knowledge base.



14 errors, 0 warnings, 0 others

Entry Point ID	Exit Point ID	Category	Code	Location	Resource
11	10	Malformed_input	title	line 59	RestServices.j
179	443	Directory_Traversal	new File(path)	line 41	Snippet.java
6	211	Cross_Site_Scripting	writer.print(title)	line 92	RestServices.j
66	70	Malformed_input	query	line 36	RestServices2.
71	285	Cross_Site_Scripting	writer.print(query)	line 69	RestServices2.

Figure 3.5: Code Analysis result

The detected vulnerabilities (Figure 3.5 gives an example of the result of an analysis in the tool) are mainly caused by lack of input validation: SQL Injection, Directory Path Traversal, and Cross Site Scripting. The engine also detects a more general Malformed Input vulnerability that represents any input that is not validated using a standard implementation. The engine can be easily extended to support new kinds of vulnerabilities caused by missing input validation. The administrator needs to add the definition of the new vulnerability to the centralized knowledge base (and, if exists, adding trusted objects that mitigate the vulnerability), and to create a new class extending an interface, that implements the checks to be done on the result of the static analysis to detect the vulnerability.

3.5 Assisted remediation with a security aspect library

Performing a static analysis is already integrated in quality processes in several companies. Yet, the actual identification of vulnerabilities does not mean they are correctly mitigated. Given this problem, we can have several approaches: (i) refactoring the code, (ii) applying a proxy to inbound and outbound connections, and finally the solution we adopted, (iii) generate protection code linked to the application being analyzed.

Software refactoring requires the developer to understand the design of the application and its potential threats, in order to manually rewrite part of the code to implement the refactoring. The refactoring improves the design, performance, and manageability of the code, but is difficult to achieve. It costs time and is error prone. Up to six distinct activities have been observed in [MT04]: (i) identify where the software should be refactored (ii) determine which refactoring should be applied to the identified places (iii) guarantee that the applied refactoring preserves behaviour (iv) apply the refactoring (v) assess the effect of the refactoring (vi) maintain the consistency between the refactored program code and other software artifacts. The impacted code is generally scattered across the application, and some part can be left unchecked easily. This can lead to an inconsistent state where the application does not reflect the intended goal. Software refactoring is one of the most powerful vulnerability remediation approache due to its flexibility in terms of code rewriting and architecture evolution.

The proxy solution is equivalent to a gray-box approach, with no in-depth visibility of internal processes. It can be burdensome to put in place, especially when the environment is under control of a different entity than the development team. For instance, on cloud platforms, developers can deploy their application but have limited management capabilities, leading to the

Vulnerability	**Origin**	**Potential Remediation**
Cross-Site Scripting	Server does not validate input coming from external source	Validate input and filter or encode properly the output depending on the usage: the encoding differs from HTML content to Javascript content for example
SQL Injection	Server does not validate input and use it directly in a construct of a SQL Query	Use a parameterized query or a safe API. Escape special characters. Validate the input used in the construction of query
Directory Path Traversal	Application server is misconfigured, or the file-system policy contains weaknesses	Enclose the application with strict policies, that restrict access to the filesystem by default. Filter and validate the input prior to direct file access
Other malformed input	Misvalidation	Validate input, determine the origin and possible manipulation from externals

Table 3.1: List of detected vulnerabilities with potential origin and potential remediation.

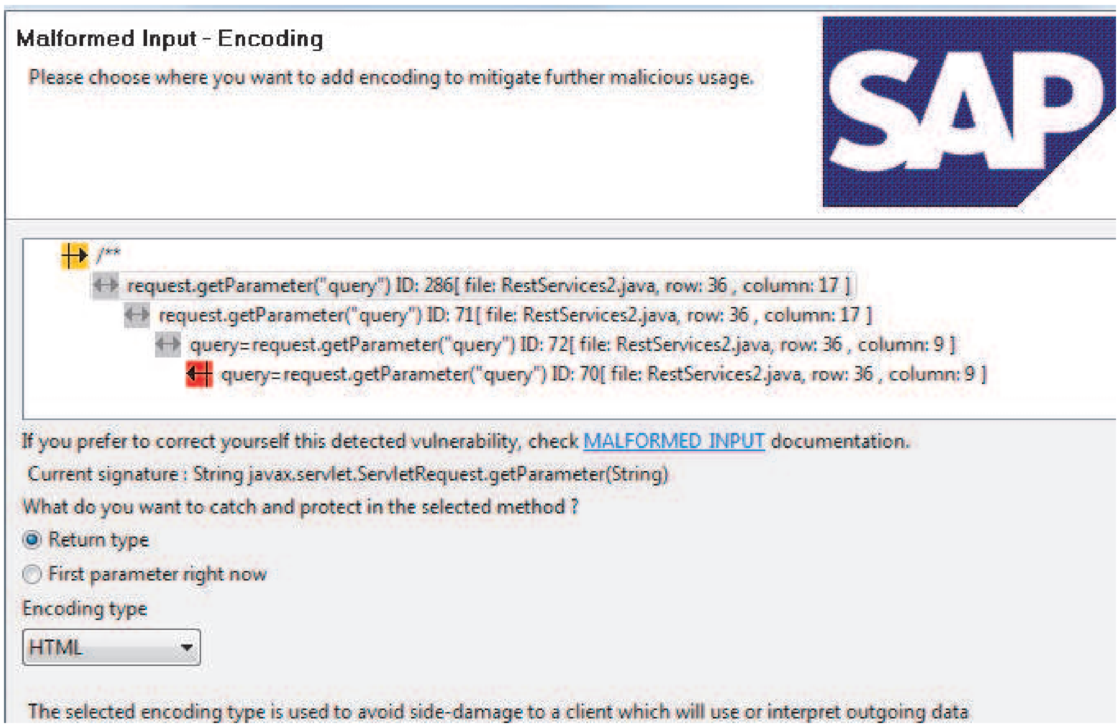
impossibility to apply a filter on the application. The lack of flexibility and the absence of small adjustments make it complicated to adopt at the development phase.

In this work we provide a protection inlined with the application. It means that the protection code is directly applied to the application to slightly modify the execution flow on vulnerable points. This solution has several advantages due to the underlying technology we use: Aspect-Oriented Programming paradigm (AOP) [KLM^{+}97], which is a paradigm to ease programming concerns that crosscut and pervade applications. In the next section, we describe our methodology and provide a comprehensive list of its advantages and drawbacks.

3.5.1 Methodology

Our methodology comprises the automatic discovery of weaknesses in the code. In addition, we integrate a protection phase tied to the analysis process which guides developers through the correct and assisted correction of the vulnerabilities previously detected. The protection phase uses information from the static analysis engine to know what vulnerabilities have to be corrected. Then the phase uses inputs from the developer to extract knowledge about the context, Figure 3.6 gives an example of the feedback asked from the developer and of the tooling supporting this feedback. These steps are necessary for our methodology to gather application context, therefore places in the application where to inject security correction. The security correction uses AOP. The goal is to bring a proper separation of concerns for cross-cutting functionalities such as security. Code related to a concern is maintained separately from the base application. The main advantage of using this technology is the ability to intervene in the control flow of a program without interfering with the base program code.

The list of vulnerabilities we cover are in Table 3.1. The table highlights the potential origin vulnerabilities and some of known remediation techniques. These vulnerabilities are known and subject to high attention. For instance, they have been in in the OWASP Top Ten [OWA10] for several years now, but also in the MITRE Top 25 Most Dangerous Software Errors [MIT11]. Albeit several approaches exist to remediate the vulnerabilities, we have chosen to apply escaping and validation techniques with aspect-orientation to consistently remediate the problems.



Malformed Input - Encoding

Please choose where you want to add encoding to mitigate further malicious usage.

```
/**
    request.getParameter("query") ID: 286[ file: RestServices2.java, row: 36 , column: 17 ]
        request.getParameter("query") ID: 71[ file: RestServices2.java, row: 36 , column: 17 ]
            query=request.getParameter("query") ID: 72[ file: RestServices2.java, row: 36 , column: 9 ]
                query=request.getParameter("query") ID: 70[ file: RestServices2.java, row: 36 , column: 9 ]
```

If you prefer to correct yourself this detected vulnerability, check MALFORMED INPUT documentation.

Current signature : String javax.servlet.ServletRequest.getParameter(String)

What do you want to catch and protect in the selected method ?

[✓ Return type]

First parameter right now

Encoding type

HTML

The selected encoding type is used to avoid side-damage to a client which will use or interpret outgoing data

Figure 3.6: Gathering context for vulnerability protection

By adopting this approach, we reduce the time to correct vulnerabilities by applying semi-automatic and pre-defined mechanisms to mitigate them. We use the aspect component to apply the protection code which is mostly tangled and scattered over an application.

Correcting a security vulnerability is not trivial. Different refactorings are possible depending on the issue. For instance, the guidelines for secure programming recommand SQL prepared statement to prevent SQL Injection. Yet, developers might be constrained by their frameworks to forge SQL queries by themselves. Therefore, developers would have to try another approach such as input validation and escaping of special characters.

Listing 5 Example of correction snippet generated for a malformed input

```
package sap.nce.research.security.aspects;

import org.aspectj.lang.ProceedingJoinPoint;
import org.owasp.esapi.ESAPI;
import org.owasp.esapi.Logger;
import org.owasp.esapi.errors.IntrusionException;

public aspect Validation {

    private final Logger logger = ESAPI.log();

    pointcut mainMethod(String s) :
        (cflow(execution(void com.sap.research.nce.RestServices.doGet
            (*..HttpServletRequest, *..HttpServletResponse)))
        && (execution(String javax.servlet.http.HttpServletRequest
            .getParameter(java.lang.String)) && args (s)))
        && !within(Validation)
        ;

    java.lang.String around(String s) : mainMethod(s) {
        s = proceed(s);
        String sanitized = "";
       try{
           sanitized = ESAPI.encoder().canonicalize(s);
       }catch (IntrusionException e){
           //in case of wrong encoding, log the error but still accept at this time
           logger.error(logger.SECURITY_FAILURE,
        "Canonicalization failed. Try without strict mode on : " + s);
           sanitized = ESAPI.encoder().canonicalize(s, false);
       }
       //add ESAPI.validator if needed
       logger.info(logger.SECURITY_SUCCESS,
        "In " + thisJoinPointStaticPart.getSignature() + "Sanitized to " + sanitized);

       return sanitized;
    }


    /***
     * encoding
     * @param s : the string object which needs encoding
     * @param target : the encoding scheme we have to apply
     */
    pointcut encodingPointcut(java.lang.String s) :
        (cflow(execution(void com.sap.research.nce.RestServices.doGet
        (HttpServletRequest, HttpServletResponse)))
        && execution(void java.io.PrintWriter.print(java.lang.String)) && args (s)
        && !within(Validation)
        ;

    java.lang.String around(java.lang.String s) : encodingPointcut(s) {
        String encoded;
        encoded = proceed (s);
        encoded = ESAPI.encoder().encodeForHTML(encoded);

        logger.info(logger.SECURITY_SUCCESS,
        "In " + thisJoinPointStaticPart.getSignature()
        + "Encoded " + s + " to (HTML) : " + encoded);
        return encoded;
     }

}
```

We assist developers by providing them an automated solution. For the previously mentioned correction, our integrated solution would propose to mitigate the vulnerability with an automatic detection of incoming, unsafe and unchecked variables. The developer does not need to be a security expert to correct vulnerabilities as our approach provides interactive steps to generate AOP protection code, like in Listing 5. The listing is an example that was generated for one of our test application. It consists of an aspect that contains two parts: sanitization of data, or encoding for a specific target format of the data. From line 12 to 18, the aspect defines one pointcut indicating which method needs data sanitization. From line 20 to 35, the actual protection code is written in an advice. The protection code uses a third-party library, although we can envision any code. From line 44 to 50, another pointcut is defined to match encoding joinpoints: points in the application that need special encoding. The advice from line 51 to 60 encode the content for an HTML target. The advice is yet statically generated by the tool, and the HTML information has been indicated through the assisted step of the plugin.

Although semi-automation simplifies the process to introduce protection code, the technique can introduce several side-effects if the developers are not following closely what is generated. The plugin gives the developer an overview of all corrected vulnerabilities, allowing him to visually manage and re-arrange them in case of need. Currently, the prototype does not analyze the interactions between the different protection code generated. By adopting this approach, we allow better a understanding of the different vulnerabilities affecting the system from a user point of view, and we guide the developer towards a better compliance of its application with best practices and corporate policies. The protection code can be deployed by team of security expert and modified without refactoring.

3.5.2 Security-aspect library limitation

The use of AOP in the remediation of vulnerabilities bring us more flexibility. One can evolve the protection library, making the security solution independent from the application. But this approach also brings us some limitations we discuss in this section.

Firstly, the language is designed to modify the application control flow. One of the limitations we have is related to the deep modification we need to perform in order to replace a behavior. For example, let us suppose that we would like to validate a SQL query written manually in the application. We are able to weave validation and escaping code, but we can hardly modify the application to construct a parameterized query. For instance, the modification would pertain several line of codes, and would concern functional code rather than crosscutting concern. The pointcut language could not handle such case, and aspect-oriented programming does not cover such case.

Secondly, the aspects cover the application in whole. When more than one aspect is involved, the cross-cutting concerns can intersect. Therefore, we need to analyze aspect interaction and prevent an annihilation of the behavior we intended to address. We further discuss this limitation in next section.

Thirdly, the evolution of the program leads to a different distribution of vulnerabilities. The vulnerabilities are detected after the static analysis phase. We are not yet addressing this problem of evolution to maintain the relation between the aspects and the application. This differs from the *fragile pointcut* problem inherent of aspect using pointcut languages referring to the syntax of the base language: the evolution affects the application as a whole, by introducing new entry points and exit points that need to be considered, or introducing methods that validate a flow for a given vulnerability.

The fourth constraint is that aspect weaving has no specific certification. The actual protection library is defined globally, but applied locally, with a late binding to the application. The protection code is the same everywhere, but we put strong trust in the protection library by assuming that aspects are behaving properly with the actual modification of the flow to mitigate the vulnerabilities.

Finally, the fifth constraint is user acceptance. Since the developers rely on a cross-cutting solution, the code itself does not reflect the exact state of the application. The point where the aspect interferes with the base application is not displayed in the code. We address this limitation with the strong interaction with the developer's environment. The Eclipse plugin provides a mean to display remediation code in place at a given time.

3.5.3 Solution pertinence

The choice of using aspect-oriented injection of security protection code relies on a correct detection of vulnerabilities and specification from a developer. The developer is an involved and implicated stakeholder that uses our solution to improve the overall solution quality.

Compared with other approaches, we provide a built-in solution with an efficient static analysis phase to collect vulnerability points in the application, in which to immediately propose corrections or guidance. We have intentionally reduced the separation between the two phases, as we are in a still flexible phase: the application is still under development and thus subject to frequent changes, refactoring, etc. Developers are left with some decisions, but the decisions are limited to the minimal interaction in order to let him correct most of the problems with minimal efforts.

Minimal efforts for the developer doesn't mean absence of reflection. We trust and leverage developer capacity to understand the problem, and our efforts are to highlight the problem and quickly enable a solution. The developer's knowledge is required to decide on these situations in which our analysis might fail, although we have general guidelines. The interference between several security code is problematic in our approach. We intensively rely on aspect-oriented programming, and thus refer to the notions of pointcut-advice model. It means we are using tools of a language - AspectJ, for instance, to syntactically detect points in the application in which we are weaving the security code that mitigate the vulnerability. One single joinpoint might inject several security snippets to correct different vulnerabilities. The order of injection is crucial to the correct anticipation of the application's behavior. For instance, we are proposing security code to validate and sanitize an input, or security code to encode content in order to normalize its data for a given usage. If one combines these different security codes, one ends up with different results:

- $sanitize + encode$: generally, the process one wans to achieve which validates the data and modifies it according to their future usage.
- $encode + sanitize$: this situation might lead to undo the encoding phase. For example, an html element is encoded once, and then decoded to be used by the application.

- $sanitize + sanitize$: you over control the data, with no real side effects.
- $encode + encode$: you loose the control over the data, having situations in which the client application is not able to properly interpret the data. For instance, imagine you get the following string from a database to display it as an html content : "*html <- value*". A first encoding will replace html entities to get "*html &lt;- value*" which renders properly on browser. A second encoding on this text will produce "*html &amp;lt;- value*" which renders incorrectly.

These situations already require user decisions, and are simple when located at a single joinpoint. The problem grows with a complete application in which we have to inject security code in all of a call-tree hierarchy as depicted in Figure 3.6. The decision of weaving security code at a certain point might introduce side effects, such as a desirable behavior is no longer possible. For instance, the situation in which we get and store a data in a backend and later use it in both an HTML element and in JSON output triggers two security alerts: cross site scripting and malformed input. If we inject our protection code before the retrieval of data from the backend, we achieve an incomplete protection and introduce a side-effect. The encoding code that we introduce will modify the data in HTML, or JSON valid data (depending on the developer's decision), and will mutually exclude the possibility to retrieve original data. The problem in such a situation is to control the point of application in the AST and decide of the best emplacement for aspect weaving. For this purpose, there are several alternatives. We have decided to represent graphically the interaction in the plugin, but we envision in future works to introduce work like in Hannousse et al. [HDA11]. They propose to detect potential interferences among aspects by formally modeling interactions between aspectualized components.

3.6 Evaluation

In order to test the accuracy of the methodology as well as the pertinence of the solution, we have defined a protocol to test several security tools and understand their capabilities at vulnerability detection of these different frameworks. We evaluated different java web applications, that are for some deliberately insecure, or that simply contains some known flaws.

Webgoat Webgoat [MO12] is one famous test application for web application security. The application has been developed by the OWASP consortium to create deliberate vulnerabilities in the application. For the sake of performance, we have decided to use a partial version of this project in our tests. The platform is specialized in web application vulnerabilities such as malformed inputs, cross site scripting, cross site request forgery, SQL injection, session fixation, etc.

Insecure InsecureWebApp [Ist05] is another OWASP project. It is a web application that includes common web application vulnerabilities. It is a target for automated and manual penetration testing, source code analysis, vulnerability assessments and threat modeling. The project proposes to guide user through user story to understand the vulnerabilities and how they can be fixed in the code.

Roller Roller [Apa04] is an Apache Project. It was featured in onjava.com. Roller is the open source blog server that was driving Sun Microsystem's blogs.sun.com employee blogging site, the Javalobby's JRoller Java community site, and hundreds of other sites. We have tested an old version that was in the test bench of Ben Livshits [Liv].

Testbench Testbench is a web application project we have crafted in order to test several test cases. It contains servlets and web applications that deliberetely contain SQL injection, cross site scripting, malformed input, directory path traversal, etc.

Personal Blog This java web application project also comes from the Ben Livshits test bench. PersonalBlog [PCE05] is a light-weight personal blogging application that is suitable for installing on your own host provider. It's written in Java and uses a variety of J2EE technologies, including: Servlets, Jsp, Jdbc, Hibernate, Struts, Tiles and Log4j.

Several tools are capable of static analysis on Java programs. We have decided to test tools that were correctly integrated in the Eclipse IDE. The following list describes the two solutions we have tested in comparison of our approach.

Bigbro Bigbro is our tool, from which we test the static analysis part. The tool was not specifically tuned for the projects, thus running with the default settings. One needs to pay attention to the fact that the results might contain several false positive. With the correct definition of trusted packages and untrusted packages in the knowledge database, that are specific to an application, the static analysis would reduce the number of false positive drastically.

LAPSE Lapse [LL05] (Lightweight Analysis for Program Security in Eclipse) is a tool designed and developed by Ben Livshits during its PhD. thesis at Standford. He was focusing on web application vulnerabilities, with also an enhanced integration into the Eclipse plugin. LAPSE is designed to help with the task of auditing Java J2EE applications for common types of security vulnerabilities found in Web applications. It helps to quickly gather taint sources, taint sinks and find the path between sources and sinks. It is similar to the type of detection we propose, but we are proposing some additional categories.

Codepro analytix The project [Goo12] is a Java software testing tool for Eclipse developers who are concerned about improving software quality and reducing developments costs and schedules. The Java software audit features assist the developer in reducing errors as the code is being developed and keeping coding practices in line with organizational guidelines". Among the different modules for code quality, there is a specific module for security audit. It provides warnings as well as critical reports for several vulnerability sources.

Table 3.2 presents the raw results that we obtain when we perform the static analysis on the different projects. The evaluation is currently limited to human-sized projects with a number of code source lines going up to 30k lines. The number of classes from the application are obtained through code pro analytix, or bigbro that both provide the information. The number of source lines is also gathered by the two tools, but we have preferred to use the sloccount project [1] to normalize the numbers.

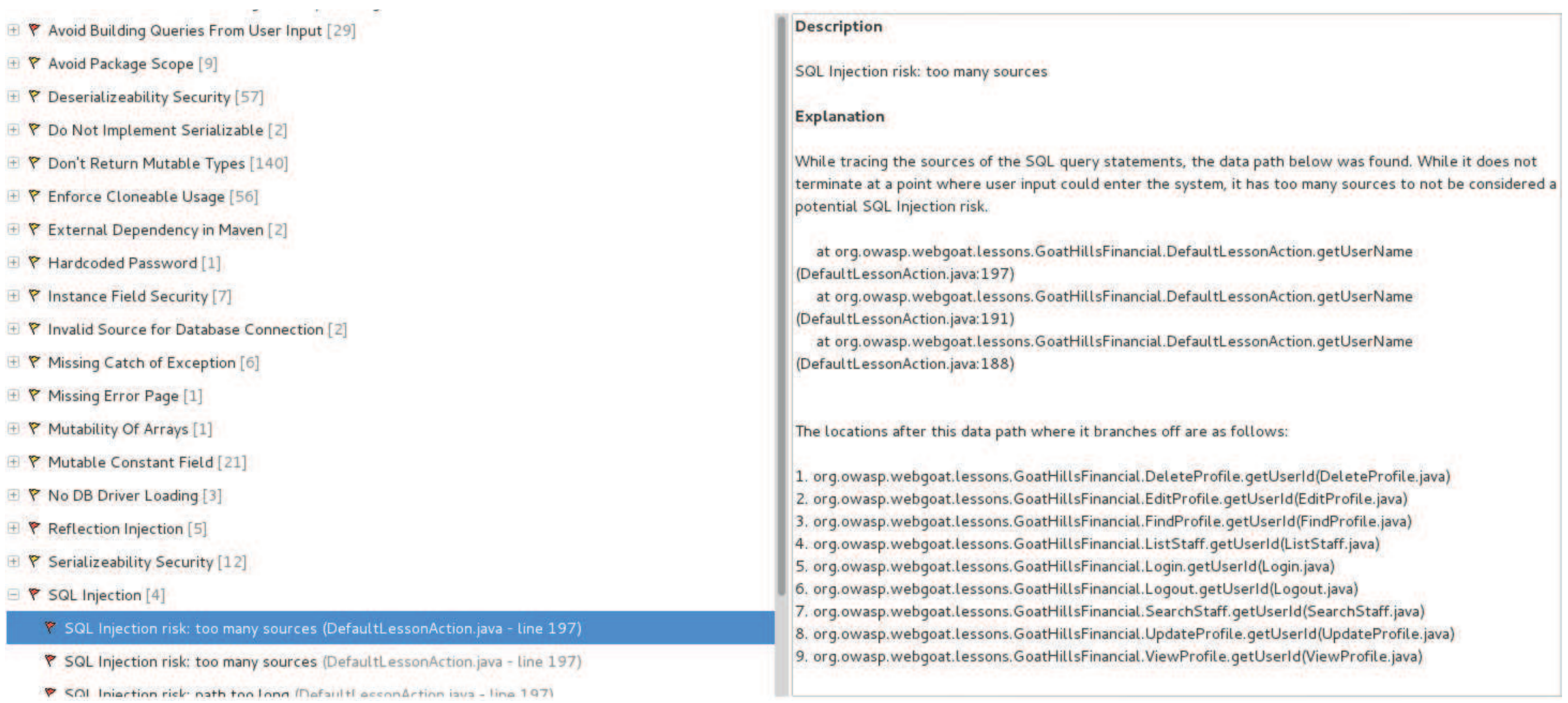
The categories XSS, SQL, and Malformed input are common to the different tools, with some distinctions for directory path traversal. There is some disparity on the malformed input category as different notions are falling into this category.

Project	Nb classes	SLOC	Tool	XSS	SQL	Malformed Input	Path Traversal	Total
Webgoat (partial project)	58	java=7525; jsp=3448	bigbro	8	162	282	29	481
			lapse	3	13	61	N.A	77
			codepro	4	29	3	2	38
Roller	275	java=30874; jsp=2806	bigbro	41	18	135	2	196
			lapse	47	6	112	N.A	165
			codepro	14	9	10	72	105
Testbench	6	java=181	bigbro	6	2	4	1	13
			lapse	5	2	10	N.A	17
			codepro	2	2	0	1	5
Personal blog	38	java=3049; jsp=1469	bigbro	0	38	81	3	122
			lapse	0	2	41	N.A	43
			codepro	0	2	0	0	2
Insecure	15	java=678; jsp=395	bigbro	7	8	20	0	35
			lapse	3	13	69	N.A	84
			codepro	2	7	1	0	10

Table 3.2: Comparison of static analysis tools on several projects. It presents the detected vulnerabilities for the different categories when applicable. SLOC means Source Line of Codes.

An example of the static analysis display in terms of result is presented in Figure 3.7 for Code pro analytix. It presents the result of the security audit, with the different warning regarding programming advices, along with critical reports such as SQL Injection, or cross site scripting. The tool provides a complete explanation of the vulnerability, as well as different locations in the analyzed source code. The analysis result for a LAPSE scan is provided in Figure 3.8. It presents the different suspicious calls detected by the analysis. For instance, the Webgoat analysis has several command injection, cross-site scripting, as well as other vulnerabilities. In Table 3.2 for the Insecure webapp project, the reported numbers are not complete as the analysis failed with some exceptions. The exceptions minor, therefore we provide the analysis result in the table. Bigbro results are shown in Figure 3.9. The summary view try to quickly indicates the problems, their origin, and it gives helper to correct the detected vulnerabilities. The helpers are directly available in the source code editor provided by eclipse, and give hints.

[1]Source Line of Codes (SLOC) is obtained by using sloccount program from `http://www.dwheeler.com/sloccount/`



```
⊞ Avoid Building Queries From User Input [29]
⊞ Avoid Package Scope [9]
⊞ Deserializeability Security [57]
⊞ Do Not Implement Serializable [2]
⊞ Don't Return Mutable Types [140]
⊞ Enforce Cloneable Usage [56]
⊞ External Dependency in Maven [2]
⊞ Hardcoded Password [1]
⊞ Instance Field Security [7]
⊞ Invalid Source for Database Connection [2]
⊞ Missing Catch of Exception [6]
⊞ Missing Error Page [1]
⊞ Mutability Of Arrays [1]
⊞ Mutable Constant Field [21]
⊞ No DB Driver Loading [3]
⊞ Reflection Injection [5]
⊞ Serializeability Security [12]
⊟ SQL Injection [4]
    SQL Injection risk: too many sources (DefaultLessonAction.java - line 197)
    SQL Injection risk: too many sources (DefaultLessonAction.java - line 197)
    SQL Injection risk: path too long (DefaultLessonAction.java - line 197)
```

Description

SQL Injection risk: too many sources

Explanation

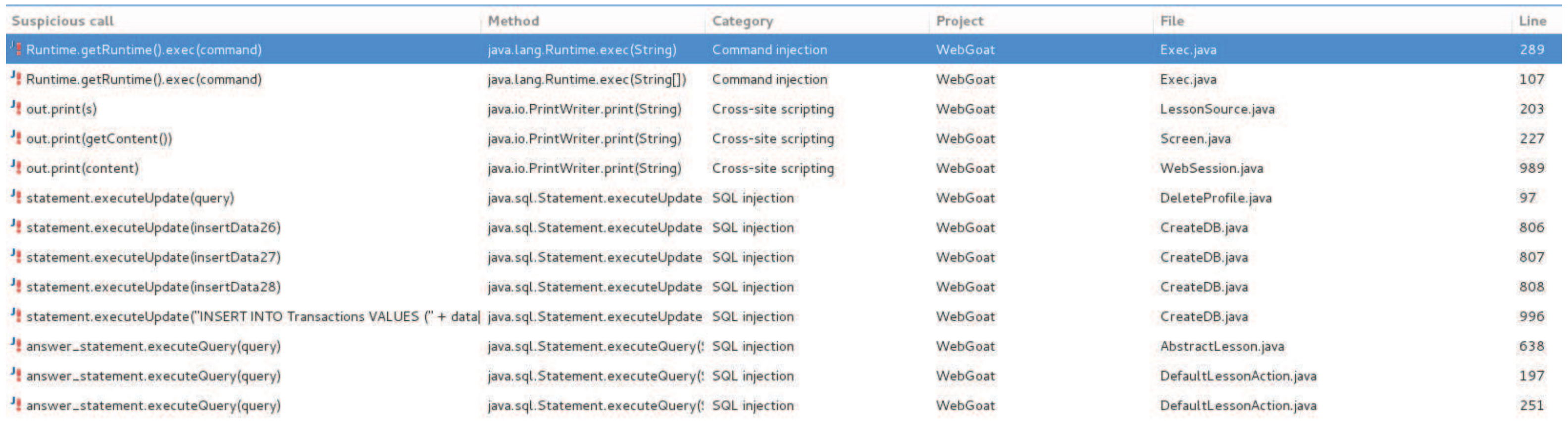
While tracing the sources of the SQL query statements, the data path below was found. While it does not terminate at a point where user input could enter the system, it has too many sources to not be considered a potential SQL Injection risk.

```
    at org.owasp.webgoat.lessons.GoatHillsFinancial.DefaultLessonAction.getUserName
(DefaultLessonAction.java:197)
    at org.owasp.webgoat.lessons.GoatHillsFinancial.DefaultLessonAction.getUserName
(DefaultLessonAction.java:191)
    at org.owasp.webgoat.lessons.GoatHillsFinancial.DefaultLessonAction.getUserName
(DefaultLessonAction.java:188)
```

The locations after this data path where it branches off are as follows:

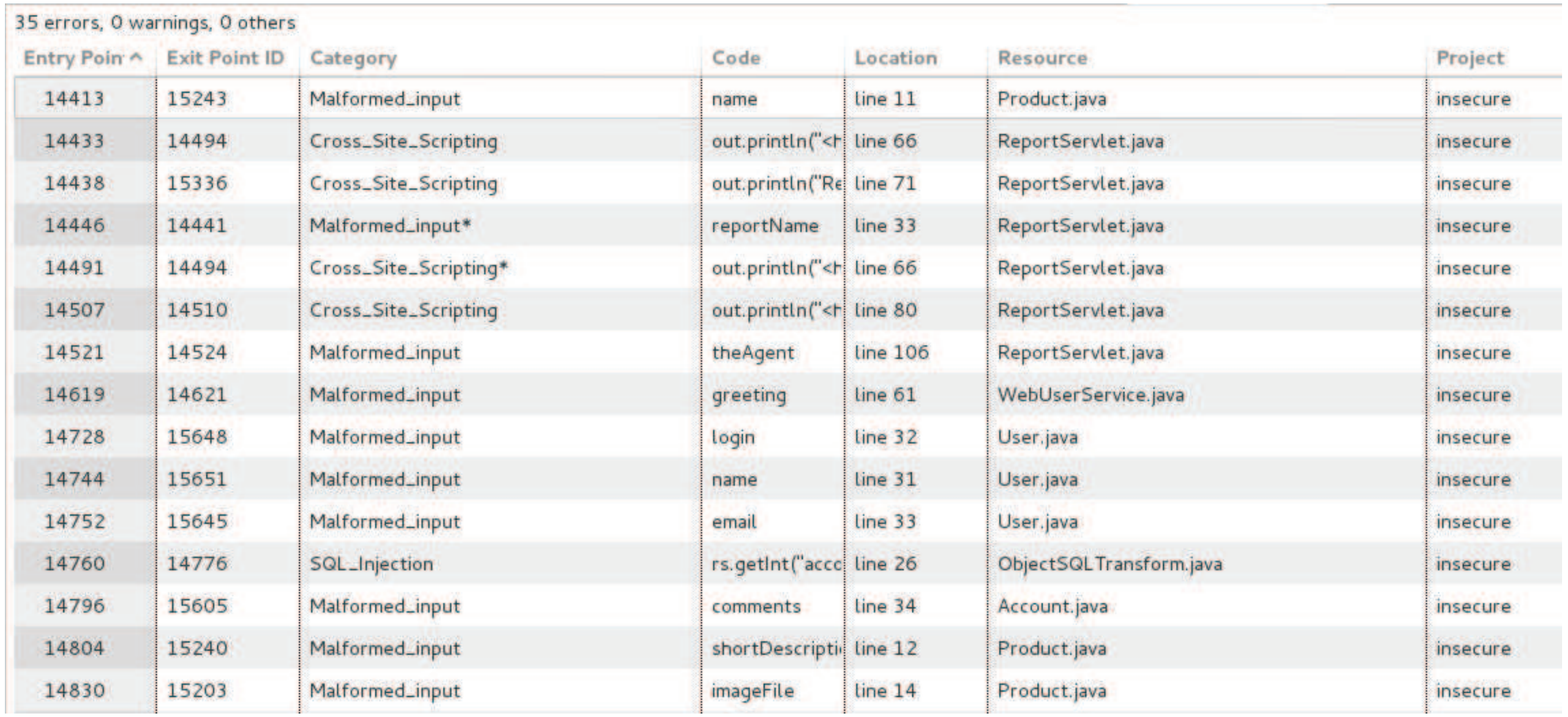
1. org.owasp.webgoat.lessons.GoatHillsFinancial.DeleteProfile.getUserId(DeleteProfile.java)
2. org.owasp.webgoat.lessons.GoatHillsFinancial.EditProfile.getUserId(EditProfile.java)
3. org.owasp.webgoat.lessons.GoatHillsFinancial.FindProfile.getUserId(FindProfile.java)
4. org.owasp.webgoat.lessons.GoatHillsFinancial.ListStaff.getUserId(ListStaff.java)
5. org.owasp.webgoat.lessons.GoatHillsFinancial.Login.getUserId(Login.java)
6. org.owasp.webgoat.lessons.GoatHillsFinancial.Logout.getUserId(Logout.java)
7. org.owasp.webgoat.lessons.GoatHillsFinancial.SearchStaff.getUserId(SearchStaff.java)
8. org.owasp.webgoat.lessons.GoatHillsFinancial.UpdateProfile.getUserId(UpdateProfile.java)
9. org.owasp.webgoat.lessons.GoatHillsFinancial.ViewProfile.getUserId(ViewProfile.java)

Figure 3.7: Example of result with code pro analytix



Suspicious call	Method	Category	Project	File	Line
Runtime.getRuntime().exec(command)	java.lang.Runtime.exec(String)	Command injection	WebGoat	Exec.java	289
Runtime.getRuntime().exec(command)	java.lang.Runtime.exec(String[])	Command injection	WebGoat	Exec.java	107
out.print(s)	java.io.PrintWriter.print(String)	Cross-site scripting	WebGoat	LessonSource.java	203
out.print(getContent())	java.io.PrintWriter.print(String)	Cross-site scripting	WebGoat	Screen.java	227
out.print(content)	java.io.PrintWriter.print(String)	Cross-site scripting	WebGoat	WebSession.java	989
statement.executeUpdate(query)	java.sql.Statement.executeUpdate	SQL injection	WebGoat	DeleteProfile.java	97
statement.executeUpdate(insertData26)	java.sql.Statement.executeUpdate	SQL injection	WebGoat	CreateDB.java	806
statement.executeUpdate(insertData27)	java.sql.Statement.executeUpdate	SQL injection	WebGoat	CreateDB.java	807
statement.executeUpdate(insertData28)	java.sql.Statement.executeUpdate	SQL injection	WebGoat	CreateDB.java	808
statement.executeUpdate("INSERT INTO Transactions VALUES (" + data|	java.sql.Statement.executeUpdate	SQL injection	WebGoat	CreateDB.java	996
answer_statement.executeQuery(query)	java.sql.Statement.executeQuery(!	SQL injection	WebGoat	AbstractLesson.java	638
answer_statement.executeQuery(query)	java.sql.Statement.executeQuery(!	SQL injection	WebGoat	DefaultLessonAction.java	197
answer_statement.executeQuery(query)	java.sql.Statement.executeQuery(!	SQL injection	WebGoat	DefaultLessonAction.java	251

Figure 3.8: Example of result with lapse



35 errors, 0 warnings, 0 others

Entry Poin ^	Exit Point ID	Category	Code	Location	Resource	Project
14413	15243	Malformed_input	name	line 11	Product.java	insecure
14433	14494	Cross_Site_Scripting	out.println("<h	line 66	ReportServlet.java	insecure
14438	15336	Cross_Site_Scripting	out.println("Re	line 71	ReportServlet.java	insecure
14446	14441	Malformed_input*	reportName	line 33	ReportServlet.java	insecure
14491	14494	Cross_Site_Scripting*	out.println("<h	line 66	ReportServlet.java	insecure
14507	14510	Cross_Site_Scripting	out.println("<h	line 80	ReportServlet.java	insecure
14521	14524	Malformed_input	theAgent	line 106	ReportServlet.java	insecure
14619	14621	Malformed_input	greeting	line 61	WebUserService.java	insecure
14728	15648	Malformed_input	login	line 32	User.java	insecure
14744	15651	Malformed_input	name	line 31	User.java	insecure
14752	15645	Malformed_input	email	line 33	User.java	insecure
14760	14776	SQL_Injection	rs.getInt("acco	line 26	ObjectSQLTransform.java	insecure
14796	15605	Malformed_input	comments	line 34	Account.java	insecure
14804	15240	Malformed_input	shortDescripti	line 12	Product.java	insecure
14830	15203	Malformed_input	imageFile	line 14	Product.java	insecure

Figure 3.9: Example of result with bigbro

The different projects have the goal to provide a quick overview of potential vulnerabilities in applications or web applications. They all encourage code quality and leverage the developer's ability to understand and correct the mistakes. The pertinence of the results (detection accuracy in the different categories) has not been completely tested, but a simple and quick overview allows to direct the developer towards vulnerable code. The developer is also redirected to pertinent documentation to understand and correct the vulnerability. Among tools, the higher detection rate does not mean that the tool performs better as several false positives might appear. We prefer to guarantee a solution free of false negative, then we can tune the results to limit the number of false positives.

3.7 Related work

The goal of static analysis is to determine whether tainted data, that is data that originate from possibly malicious users, reaches sensitive sinks (e.g. vulnerable points in the program) without being properly sanitized. For this purpose, data flow analysis techniques that operate on the control flow graph are used. Static analysis can be applied in cases where the source code or the bytecode is available. The advantage is that it is not necessary to execute the program to detect injection vulnerabilities. They analyze the application based on a model they abstract from the application. Unfortunately, approaches based on static analysis suffer from false positives and false negatives. This is due to imprecise approximations of the control and data flow available at runtime. In addition, false positives might result from some runtime validation at which the security label of the data (tainted/untainted) is not changed after the validation.

3.7.1 Static analysis

The literature related to static analysis is abundant. We provide in this section a quick review on approaches and position our work with regards to the developed techniques. A first approach consists in detecting common mistakes through pattern matching and string analysis in source code, like in [VBKM00, GSD04, WS04, CMS03]. The approach has been developed to provide offline and quick techniques to support code review. The downside of this approach is the limited analysis: they mostly read the code, looking for patterns. For instance, some tools are looking for *strcat* function in C language to report them as unsafe function. More complex techniques arisen to provide in-depth, contextual, and flow-sensitive analysis [LWLa05, LM10, BCF+08] There are also several commercial tools, such as Fortify [HP12] or CodeProfiler [Vir12]. They propose a large range of techniques to easily embed code review process, especially static analysis directly in developers environment. The target of these solutions is mainly large industries.

The WebSSARI project [HYH+04] pioneered vulnerability detection in web application with runtime protection. WebSSARI uses a combination of static and dynamic analysis to detect vulnerabilities in PHP code. Jovanovic et al. designed Pixy [JKK06b, JKK06a], a static analyzer tool that features a high-precision data flow analysis engine. This engine is flow-sensitive, interprocedural, context-sensitive, and performs alias analysis or literal analysis. Another approach that aims at overcome some of the limitations of WebSSARI is the work by Xie and Aiken [XA06]. Their approach performs an interprocedural analysis, which is able to model conditional branches and supports dynamic typing. The work by Wassermann and Su [WS07] employs a string-analysis based approach to detect SQL Injection vulnerabilities. It tracks the source of string values and ensures that user-supplied input is isolated within a SQL query. [WS08] presents a static analysis approach to detect Cross-Site Scripting vulnerabilities. It also employs string analysis techniques.

Several tools are based on the Eclipse's platform and detect vulnerabilities in web applications. Livshit et al. [LL05] developed a technique to review major web application flaws through a query language, to follow tainted object propagation in source code. We can also mention the SSVChecker tool [DFH06] that combines several external security detection tools to aggregate results and provide easy to review interface from the IDE. A bunch of Eclipse plugins have also been developed to enhance code quality [Uni12] or to verify the adherence to organizational guidelines [Goo12]. While the focus is on quality, these tools can detect some security vulnerabilities. Recent work from Xie et al. [XCLM11] studies programmers' behaviors to understand their needs, and proposes an IDE support for web application security. Their integration of security protection code is made through the refactoring of some part of the code. It consists of input validations and other checks.

3.7.2 Detection and protection with AOP

Compared to the aforementioned techniques, we aim at a better integration into the daily development lifecycle with our tool, and propose an integrated correction with good accuracy as we leverage the developer's knowledge about the development context. More specifically, we leverage aspect-oriented techniques to clearly separate business code from security code.

Hermosillo et al. [HGSD07] use AOP to protect against web vulnerabilities (XSS and SQL Injection). They use AspectJ - the mainstream AOP language, to intercept method calls in an application server then perform validation on parameters. Viega et al. [VBC01] present a simple use case for the use of AOP for software security. Masuhara et al. [MK03] introduce an aspect primitive for dataflow, allowing to detect vulnerabilities like XSS. More recently, Masuhara et al. introduced a design and implementation of AspectShield [SW13] in order to mitigate the most common web applications. They leverage AOP to clearly separate concerns. They identify vulnerabilities from an external tool (Fortify in this case), to then produce aspects with control points at the detected places where vulnerability might be exploited. Their solution comprises what we offer in our approach, but they are a bit less flexible than ours, due to the binding from Fortify to AspectShield. In our case, the static analysis process is part of the whole process. It means that we provide the process is integrated and maintained by us, which makes it easier to correctly bind security aspects to the application. Furthermore, AspectShield covers SQL-injection and Cross site scripting vulnerabilities, whereas we provide an extension system to map new vulnerabilities.

Our approach reduces the overhead brought by the detection of vulnerability patterns at runtime and allows a wider range of vulnerability detection. Our integrated approach differs also from the state of the art, as the aforementioned approaches use either external tools or manual processing to understand the architecture and to decide where to apply aspects/security validation. Our approach also brings more awareness to the developer as he obtains a visual indication of what is applied at which place in his application.

A combination of detection and protection is found in Deeprasertkul et al. [DBO05]'s approach. It detects faults through pre-compiled patterns. Faults are corrected using a correction module. The difference with our approach lies in the detection of faults rather than security vulnerabilities. Faults are different than security vulnerabilities as they are not introducing security breaches. They rather introduce defects in the application. Also, the correction module fixes the faults statically and prevents further modifications of the introduced code. A recent work conducted by Yang et al. [YAM+11] uses static analysis to determine the security points where to deploy protection code with aspects, for distributed tuple space systems. These two approaches suffer from the same limitations as the ones presented in the previous paragraph, which is a lack of visual support from the tool, and a loss of context: people responsible in correcting the problems detected are not familiar with the architecture and technical choices. It is worth mentioning the work from Hafiz et al. [HAJ09], in which the authors propose several techniques to correct data injection through program transformations. They have listed several cases along with transformations to realize security policies. Their work can benefit from our overall methodology to propose multiple corrections once vulnerability has been identified.

3.8 Summary

We presented how to overcome several security vulnerabilities using a combination between a static analyzer that assists developers to report security vulnerabilities and a semi-automated correction of these findings with AOP. The usage of an integrated tool to provide support for security bugs detection and mitigation has several advantages. It benefits several stakeholders at the same time. First, security teams are able to distribute the maintenance of the code to the people writing their code and let them mitigate security bugs whenever they are detected. They can interact closely to decide of the best solutions for a given situation, and apply security across development teams. Developers benefit from this approach, having an operational tool already configured for their development. They can focus on writing their functional code and, from time to time, verify the accuracy of their implementation. Security concerns are often cross cutting the application, which tends to have security checks spread around application. Using one central tool to have an overview is more efficient and productive, and gives the possibility to track all applied protection code. The automation allows a broader and consistent application of security across applications. The use of AOP eases the deployment and change of security protection code, in a single environment and during the development phase. The overall vision we would like to achieve in the future is the specification and maintenance of security concerns in one central place, and usage by developers of these concerns by defining some places in application where they should be active.

We have designed an Eclipse plugin for an improved awareness of security concerns from a developer point of view. It is important to notice that correcting vulnerabilities doesn't make the whole system secure. It only means the code tends to be free of security bugs. Other parts of the application, such as authentication flow, authorization checks, etc. are not covered by our analysis. Besides, we encourage developers to look further in vulnerabilities' descriptions, as the automated correction proposed might not be the best choice in all situations. We do not want developers to believe our solution is bullet-proof. It leads to a false sensation of security, which is the opposite of our goal.

This contribution brings a programming approach to the modularization of security. We start from the source code of an application and try to inject security code at correct places. Such integrated tool has several benefits mentionned above, but it suffers from limitations due to implementation decisions. For instance, when we are developing a tool such as an Eclipse plug-in, we are targeting a platform and a language, thus voluntarily restricting the scope of application. From the tool itself, we have designed a working prototype that we have validated on projects internally at SAP and compared to commercial software. In several cases, the agile approach that consists in daily code scan and vulnerability remediation leads to a reduction of false positives and an absence of false negatives. Also, the approach of providing support for correcting the vulnerability is novel and we focus now on improving accuracy of the protection code as well as the accuracy of interaction between the several security snippets we introduce with aspects.

Chapter 4

Automation of input validation verification in application

We address the problem of modular input validation for web services as a countermeasure to several kinds of code injection attacks. The solution relies on annotations that provide meta-data concerning the application's input parameters. This information is then used to automatically insert validation code in the target application, using an aspect-oriented approach. The solution allows to mitigate risks and to maintain security functionality separated from the application logic. The difference with the previous chapter mainly relies on the method to gather points where to inject security. Whereas in previous solution we were using a static analyzer, we now leverage annotations in the application source code to indicate specific meta-data.

4.1 Introduction

Many web applications and web services are prone to input validation vulnerabilities. Well known instances of this class of vulnerabilities include Cross-Site Scripting, SQL Injection, and Command Injection. Although Input Validation vulnerabilities are well-known and have been well studied in the past decade, Input Validation Vulnerabilities such as SQL Injection and Cross-Site scripting dominate the charts for many years now. One well-known security awareness program is the Top Ten Project hosted by the Open Web Application Security Project (OWASP)[1]. It aims to identify some of the most critical risks facing organizations by publishing lists of these risks. In the past decade, several versions of this list have been released. SQL Injection and Cross-Site scripting vulnerabilities have always been among the top positions on this list.

Input validation vulnerabilities all have the same root cause: an improper sanitization of user-supplied input that results from invalid assumptions made by the developer about the input of the application. Injection attacks, that exploit input validation vulnerabilities, are attacks in which an attacker creates inputs containing special characters and/or markers that alter the behavior of the targeted application in some undesired way. Such attacks can have devastating

[1]`https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`

consequences, ranging from information leakage to privilege escalation in which the attacker can gain full control of the system under attack.

Injection attacks, also called code injection attacks, can take several forms:

- SQL injection is the insertion of a SQL query via the input data from the client to the application. Via this attack, one can obtain sensitive data from the database, to modify it, or to execute administrative operations on it.

- Command Shell injection allows to insert and to execute commands specified by an attacker from the input to a vulnerable application, making it possible to execute unwanted system commands.

- Cross-site scripting (XSS) attacks: In this type of attack, malicious scripts are injected into the otherwise benign and trusted web sites. Cross-site scripting (XSS) attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Cross-Site scripting vulnerabilities are quite widespread and occur whenever a web application uses input from a user, that is not validated or encoded, and that is propagated in the output of the application.

- Other kinds of injection are possible, but the mitigation strategy is similar and covered in this work. We can mention for instance XML and XPath injection, which occur when a web site uses user-supplied information to construct an XPath query for XML data. By sending intentionally malformed information into the web site, an attacker can find out how the XML data is structured, or access data that he may not normally have access to. He may even be able to elevate his privileges on the web site if the XML data is being used for authentication (such as an XML based user file).

Preventing input validation vulnerabilities is a complex task. Scholte et al. have shown in [SBK11] that despite security awareness programs and tools for detecting input validation vulnerabilities, this class of vulnerabilities is still very prevalent across web applications and the number of reported vulnerabilities is not decreasing. Moreover, the complexity of the attacks exploiting this class of vulnerabilities has not been increasing.

In order to prevent input validation vulnerabilities, every input read by the program must undergo a validation and sanitization process. We focus on input validation which is, essentially, the process of assigning a semantic meaning to unstructured and untrusted inputs of an application, and ensuring that those inputs respect a set of constraints describing a well-formed input. Depending on the data type, additional validation checks might be necessary. For example, a string might contain only allowed characters. As another example the length of a string should stay within certain boundaries. As a third example, the validation process for numerical input might check if the value stays within the expected range and if the value is signed or not.

One of the reasons behind the prevalence of input validation vulnerabilities is that the application of any techniques to prevent them relies entirely on the developers. Although several frameworks do provide libraries containing validation and sanitization functions, these still need to be explicitly called from the application logic in order to validate or sanitize the input provided by users. This has two distinct and important disadvantages: first, developers simply forget (or ignore) to use the already available input validation functionalities. Second, it is hard to maintain, update and evolve the application logic independently - since validation function calls would be scattered all along the application code. Moreover, the validation functionalities built in web application frameworks do not have the necessary degree of granularity to handle the validation of a large number of different datatypes that an application typically handles.

Since preventing input validation vulnerabilities relies entirely on developers, prevention techniques that are part of the design and implementation phases of the software development lifecycle will help in making web applications and web services more secure. We develop in this work a method that prevents input validation vulnerabilities through strictly separating input validation code from application code. In this way, the assignment of data types to input can be enforced while maintaining consistency between the input validation and the application logic. More specifically, it consists in the non-invasive use of Aspect-Oriented Programming for the automatic generation of input validation code, without altering the business logic of the concerned application.

The term Aspect-Oriented-Programming [KLM$^+$97] (AOP) has been coined around 1995 by a group led by Gregor Kiczales, with the goal to bring proper separation of concerns for cross cutting functionalities. As we have explained in previous chapters, the aspect paradigm is uses advice and pointcut. Pointcuts allow to define where (at which points in the source code of an application) or when (based on which events during the execution of an application) aspects should apply modifications. Pointcuts are expressed in pointcut languages and often contain a large number of aspect-specific constructs that match specific structures of the language in which base applications are expressed, such as a pattern language based on language syntax. Advices are used to define the modifications an aspect may perform on the base application. Advices are often expressed in terms of some general-purpose language with a small number of aspect-specific extensions, such as the *proceed* construct that allows the execution of the behavior of the base application that triggered the aspect application in the first place.

The solution that we have designed has a very high intrinsic business value, especially within SAP landscape. Many customers around the world expose back-end software functionalities through web services. Providing this kind of automation would improve the overall security of software and also the protection of customer data. It is even more important as applications as a service (such as SAP Business By Design) are spreading, with the release of numerous web applications highly exposed to security risks.

The next sections are organized as follows:

- Section 4.2 details the components and method of our solution.
- Section 4.3 presents a concrete example.
- Section 4.4 presents some evaluation using black-box tools to uncover vulnerabilities.
- Section 4.5 discusses related work that we can compare to our technique in this domain.
- Section 4.6 summarizes the advantages of our solution.

4.2 Aspect-based modularization with enhanced data-types

4.2.1 Architecture and methodology

Our solution comprises a methodology and a tool for mitigating input validation vulnerabilities. The methodology requires that the application developers annotate source code of the application components to be protected. Annotations are a simple way to extend a given programming language in a non-invasive way. In our case, the annotations indicate what the input parameters are and their corresponding enhanced data-types individually. After the programs are annotated, the tool will generate new executable or object code, using aspect-oriented programming techniques [KLM$^+$97]. The obtained code will intercept the execution flow whenever an input is received in order to check whether the input is in conformity to some pre-defined format. In the case an incorrect input is read by the application, then a programming exception is raised.

The methodology we propose assumes that all input parameters in the code must be annotated by the developer, otherwise the application will not be executed. However, this feature can be turned off, allowing the developers to partially annotate the code, or to disregard completely the annotation phase. In other words, annotating all input parameters in the source code is mandatory by default.

Correctly annotating the input parameters is critical as it ensures the future verification of all incoming data. An incorrect validation mechanism can compromise the risk mitigation process. In order to correctly bind the parameters and variables of interest, we adopt a semi-automatic approach combining user-based knowledge as well as an automatic detection of data types. We have not implemented this part, which as then been covered in [SRBK12]. The automatic detection of data type can come from several sources, for instance by using information gathered from model repositories, database schemas, and so on.

4.2.2 Definition of enhanced data types at the design phase

At the design phase the developer has to define the enhanced data types, also called Global Data Types, that are used across the application. Enhanced data types have business semantics and convey more precision on the expected user inputs. Therefore, these data types differ from the basic primitive and built-in types of the programming language. Examples of enhanced data types are types with business semantic that are specific. A mail address is more specific than a string, as one can define constraints. It is also the case for phone number, addresses *etc.* These enhanced types are added as an intermediate layer between the language types and our model, in order to obtain a fine-grained and stronger typing related to variables and parameters used in the application.

For instance, in a declaration such as `String email;` the developer would add the annotation `@Email String email;` indicating that only strings obeying a certain pattern for email addresses shall be accepted. The actual validation of an input can take several form, such as pattern validation, but also check of mail existence, *etc.* Here, we consider that the set of enhanced data types is extensible as well as the corresponding validation functionalities for each extended data types.

The tool is built from three main components as illustrated in Figure 4.1. The pointcut interface adaptor keeps a mapping between enhanced types and validation functions. This component can also extract data-type information from external knowledge bases to add meta-data information necessary to the input validation. Examples of external information sources are service repositories, such as the SAP Enterprise Services Repository, database schemas, WSDL files, etc. These sources can provide information about the type structure used in the application parameters, such that we can infer enhanced data types associated to them. In these knowledge bases one can find further information, such as the required length for data fields, or enumerated values, which can be useful to gain accuracy in the input data validation.

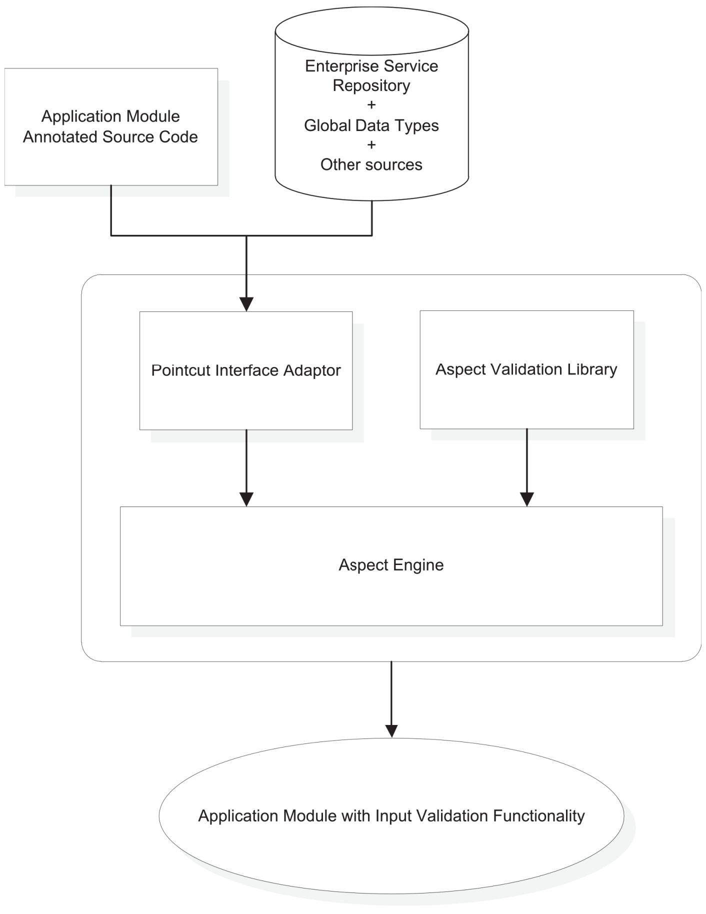


Figure 4.1: Solution components

4.2.3 An aspect-based tool for validation

The solution comprises different phases using several components to correctly implement an automatic validation during the execution of the application. The Figure 4.2 presents the different components that we use in our approach, along with the optimal separation of roles in the processing of concerns. The numbers also present the order of development required to achieve the solution. In an optimal situation, there are two teams communicating to elaborate together the exchange interface of the application. They define in the application a business model layer that defines all business objects, and then they agree upon enhanced data types. The developers are responsible for developing the application, and can use enhanced data types when they introduce new entry points in the application. The developers can also use enhanced data-types in the business model, while building the application. Security experts are responsible of providing a validation library for the enhanced data types that they can adapt to the business model specificities.

Security experts propose a set of enhanced data types, that can be extended over time. The list of enhanced data types can also be extended by any developer, although a security expert would have the most appropriate role to provide clear and accurate information to mitigate risks.

Beside the optimal situation, we have designed this approach to fit in an already defined application. The business model can hardly evolve, which benefits to a non-intrusive approach using the enhanced data types. Our non-intrusiveness is made possible by the annotation handling in existing classes, that do not modify the control flow of the application, and do not necessitate code refactoring.

The Aspect Engine is responsible for the detection of validation points during the execution of the application. The Aspect Engine is capable of modifying the application control flow. It takes into account the type annotations and inserts data validation code whenever there is an assignment for an input parameter, called validation point, that is, whenever data is read from untrusted sources or received from clients. A validation point refers to the validation of a specific parameter or variable from the base application. Upon detection of a validation point, the Aspect Engine extracts the parameter's enhance data-type that is indicated in an annotation and looks for a corresponding validation library for the specified type. If the aspect finds a corresponding library, it applies the validation mechanisms defined in the validation module. The last component in the architecture consists of an extensible aspect library where the validation functions for each enhanced data type are given. This library maps each enhanced data type (Global Data Types), for example types present in SAP's Enterprise Service Repository (ESR), to validation functions that are represented as advices in the library. The implementations of validation functions have a standardized interface in order to ensure compatibility and ease the introduction of new validation functions. We provide concrete examples in the coming sections.

The regular process to create a new enhanced data type is the following. When someone identifies a specific data type, he creates an identifier name for it. This name is released among the application developers and stakeholders. The Listing 6 is an example used in our application to share data types as a Java enum.

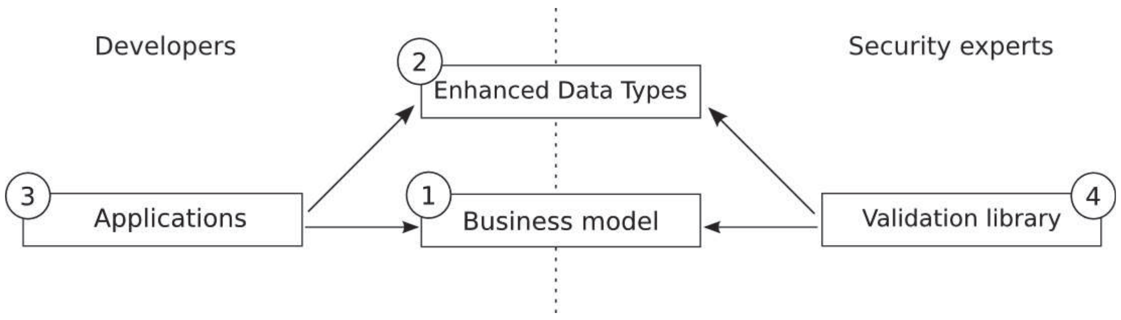


Figure 4.2: Components and ideal roles

Listing 6 Available enhanced data types presented in a Java enum. These types can be used by developers in dedicated annotations

```java
public enum DataType {
    FLIGHT_NUMBER,
        DATE,
        EMAIL,
        NAME,
        ID,
        TITLE,
        SSN,
        PHONE,
        ADDRESS,
        SALARY,
}
```

The enhanced data types can be used to taint variables and parameters along the base application. The Listing 7 shows the different use of our method: the annotations can apply to a method parameter, a constructor argument, or a class variable. It provides a large range of possibility from application to business model tainting. The listing present a Customer object that has several fields: a name, a firstname, and an email. The value of the fields change in several places, and we would like to cover the different possibilities with our approach. We use code annotation at three different places to showcase three different situations in which our approach can indicate a need for input validation. The first possibility is at line 6, which is field annotation. Such annotation will allow us to attach a tag to the field and later detect any change to the field value. For instance, any affectation to this field will be *seen* by an aspect engine. Such positioning is desirable when a developer want to monitor all changes to a field object. In other situations, he still has two other possibilities. At line 9, a second possibility is to annotate a method parameter. In our case, it happens that the method is also the email setter, but the behavior is thoroughly different. Any method in an application can be annotated this way. For instance, the behavior is desirabled when one wants to tag parameters coming from a servlet, from property files, *etc.* A third possibility, shown at line 13 is to tag a constructor's parameter. Even if there are less cases, one might still want to tag inputs to apply a validation during instantiation of an object.

Listing 7 Business model can be annotated in several places

```java
public class Customer {

    private String name;
    private String firstname;

    @Type(DataType.EMAIL)
    private String email;

    public void setEmail(@Type(DataType.EMAIL) String email) {
        this.email = email;
    }

    public Customer(String name, String firstname,
            @Type(DataType.EMAIL) String email) {
        this.name = name;
        this.firstname = firstname;
        this.email = email;
    }

    /* ... */

}
```

In parallel of tagging through annotations to augment the type system information of the application, security experts can rely upon the enhanced data types (acting as the exchange interface layer) to develop corresponding validation aspects and put them in the validation library. The validation behaviour is represented by several code advices. It is possible to define multiple validation aspects to a single identifier in the validation library. Having several validation mechanisms for a similar enhanced data type in the validation library will enforce as many validation as mechanisms present in the library.

In most cases, the behaviour to validate a type can be given in terms of regular expressions. For instance, a telephone number might check length and digits with a pattern. The handling of regular expressions is frequently provided as a built-in functionality in many programming languages. The downside of such expressions, is that they depends on application context: several format of telephone number exists depending on customer's area for example. When it comes to complex types (as opposed to a phone number which represent one element), a complete validation can be introduced by the validation library as the business model is available from security experts (who write validation code). The validation can therefore validate complex business types, and validate them through different means: functional validation, additional technical checks, etc. For instance, one can verify existence of an e-mail address by contacting a mail transfer agent, or wire transfer validation might involve third parties services. More sophisticated attack vectors would require advanced pattern matching, therefore the valid input would need to be specified through XML-Schema validation, for example. This would allow for a more expressive class of languages to be accepted as input, that is, context-free languages. Once the advice code for a specific enhanced data type is created, one needs to encapsulate the validation code into an aspect and deploy the validation library.

In Figure 4.3, we represent the activity diagram of the solution we have implemented. We consider that aspects are inserted at deploy time into the target application. We assume at this

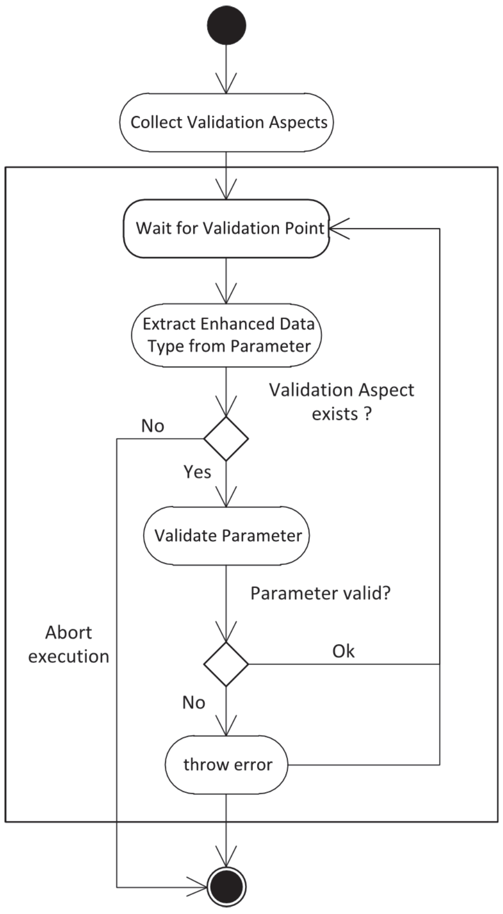

Collect Validation Aspects
Wait for Validation Point
Extract Enhanced Data Type from Parameter
Validation Aspect exists ?
No
Yes
Validate Parameter
Parameter valid?
Abort execution
Ok
No
throw error

Figure 4.3: Internal activity diagram

point that several aspects exist in the validation library. The second assumption is that the application about to run has accurate tagging through annotation with enhanced data types. In this implementation the Aspect Engine is a specialized class loader which bootstraps all target applications. The first action of the Aspect Engine is to search for available aspects in the aspect library. As the application code is loaded, the Aspect Engine discovers the points in the code that will need a validation and also the applicable validation aspects at those points. If no annotation for an input parameter is found, the application execution is aborted. This behavior can of course be adapted, depending on the desired strategy. The Aspect Engine will also abort the execution if there is no validation aspect corresponding to a used enhanced data-type.

Next, the Aspect Engine will proceed with the execution of the application code and observe the application execution until it reaches a validation point. At this point, it detects an enhanced data type annotation used by the base application and searches among the loaded validation library one or more corresponding validation aspects. The Aspect Engine then applies the validation function for the parameter found. It finally loops to monitor application execution to cycle until the end of the application. These steps correctly enforce input validation in the application, albeit the application itself has been slightly modified to enhance type system of input parameters through annotation tags. The mechanisms to enforce the validation of the types is managed independently and use aspects mechanisms to apply a systematic protection in the aspectized application.

4.3 Use case

In order to illustrate the technical solution we have developed, we apply the concept to a subset of the loan scenario we have developed in [DGM$^+$10]: the application contains components to manage customers. A manager has the responsibility to create, read, update, delete managers. It can perform these actions through web-services and web-applications. The server application is object-oriented, allowing to easily encapsulate behaviors related to the customer's manipulation. In order to provide a consistent and systematic validation of customer's fields, we decide to test our solution in this environment.

Listing 8 presents a servlet in java. We enhance the data type of the servlet at several points to properly execute a validation mechanism at runtime. At these points, the concrete validation mechanism are not yet defined nor linked to the application, but we provide additional information to later let the program verify automatically the input. Thus, the actual binding is deferred for a maximal flexibility and to provide concrete mechanisms related to the actual environment the application runs in. The actual locations to enhance data-type are manifold. The example shows various points in comments. The possibilities are:

- Field attribute. It allows to indicate an enhanced data type to a class attribute. This method is to choose when applicable as all classes and services manipulating an object will transparently execute the type verification. The application is notified when a modification to this field is run, *i.e.,* when a new assignment occurs.

- Constructor attribute indicates an enhanced data type on a constructor parameter, thus a verification should takes place at the object construction the correctness of the element.

- Method attribute indicates an enhanced data type on a method parameter. It is a generalization of the constructor parameter, but that can be applied to any situation.

Listing 8 StatusServlet.java with enhanced type

```
public class StatusServlet extends HttpServlet {

    public static void test(@Type(DataType.ID) String value) {
        /** */
    }

    public static void test2(@Type(DataType.STRING2) String value) {
        /** */
    }

    /**
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse
     *      response)
     */
    protected void doGet(
            @Type(DataType.STATUS_ARGS) HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
        test(request.getParameter("arg1"));
        test2(request.getParameter("arg2"));
    }

}
```

The tagging is yet defined manually, but there are some mean to automatically infer the correct enhanced type from several data-model information: database schemas linked with the application, data type description in documentation, constraints expressed in external framework to assist the modification of the business objects, *etc.*. These components have already provide a data description with specific constraints. One can use these definitions to verify at the applicative level that data is properly formatted. The existence of enhanced data types are validated during the compilation process, although they are considered as a virtual and intermediate layer. The Listing 9 shows the definition of Java annotations . In the left part, we present the definition of the $@Validation$ annotation, which takes an object of type $DataType$. This annotation indicates what $DataType$ the currently annotated class validates. The right part presents the $@Type$ annotation which is the annotation used for tagging inputs in application.

Listing 9 Annotation definition

```
/***
 * This annotation indicates what type is
 * covered by the validation functions
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface Validation {
    DataType value();
}
```

```
/***
 * Type annotation is to indicate in the code
 * of which type is a given parameter
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.PARAMETER, ElementType.FIELD})
public @interface Type {
    DataType value () default DataType.STRING;
}
```

We present in the following an example of a validation class to verify correctness of the field in Listing 10. The validation class is indicated to mitigate vulnerabilities for a given enhanced data type thanks to the $@Validation$ annotation at line 6. In this case, it validates an email field, introduced in previous section in Listing 7. The validation class is one of the possible implementation. It heavily relies on the application context, and can change. For instance, we can envision an automatic verification of the email field through the sending of a mail to confirm the mail reality and user's legitimate request. Such a request brings additional cost in term of performance or delay, but can be worth spending depending on the legal compliance, or business policies. The validation code which is executed if one of the *doProcess* methods, depending on the Java type of the input being validated. One can validate an email from a string, like at line 27, or from a complex object, like at line 34. For instance if an application is using a Customer input and that an enhanced data-type is defined to valide this input, the shown validation class for email will verify the customer's mail.

Listing 10 Validation class example for the Email enhanced data-type

```
1  package validation;
2  import java.util.regex.*;
3  import annotation.Validation;
4  import data.Customer;
5
6  @Validation(DataType.EMAIL)
7  /**
8   * For example <a>http://en.wikipedia.org/wiki/Email_address#Syntax</a>
9   */
10 public class ValidationEmail implements ValidationInterface {
11
12     /**
13      * Excellent recap from http://www.regular-expressions.info/email.html,
14      * then adapted
15      */
16     final String emailpattern = "[\\w!#$%&'*+/=?^_`{|}~-]+(?:\\.[\\w!#$%&'*+/=?^_`{|}"
17         "~-]+)*@(?:[a-zA-Z0-9](?:[A-Za-z0-9-]*[A-Za-z0-9])?\\.)+[a-zA-Z]{2,}+\\.?";
18     final Pattern p = Pattern.compile(emailpattern);
19
20     /**
21      * Validate an email string.
22      * The policy is :
23      * <ul><li>the email has no value -> no validation is performed</li>
24      * <li>the email has a value -> a validation check is performed and raise an
25      * exception if not correctly formatted</li></ul>
26      */
27     public boolean doProcess(String email) throws Exception {
28         if (0 == email.trim().length())
29             return true;
30           Matcher m = p.matcher(email.trim());
31           return m.matches();
32     }
33
34     public boolean doProcess(Object o) throws Exception {
35         if (o instanceof Customer)
36             return doProcess(((Customer) o).getEmail());
37         return doProcess(o.toString());
38     }
39 }
```

Listing 11 presents an excerpt of what is present in the validation aspect. The validation aspect is the enforcement part which unify the different components: application and validation library. It is executed upon detection of enhanced data-type tagging during execution of the application, which trigger a code a retrieve the input value and validate it against the indicated enhanced data type. The three pointcuts presented here are those for setter detection, constructor argument detection, and method parameter detection.

Listing 11 Excerpt of the validation aspect to present the annotation detection with aspectJ.

```
@Pointcut("execution(* *(.., @annotation.Type (*), ..))")
public void pointcutMethodDataTypeAnnotedParams() {
}

@Pointcut("execution(*.new(.., @annotation.Type (*), ..))")
public void pointcutConstructorDataTypeAnnotedParams() {
}

@Pointcut("set(@annotation.Type * *) && args(val)")
public void pointcutFieldDataTypeAnnotedParams(Object val) {
}
```

4.4 Validation

In order to evaluate our solution, we have chosen to apply it to existing vulnerable applications and to verify the actual benefits of our technique. We protect against a whole range of vulnerability that are derived from a common weakness: *improper input validation* [MIT09]. We can measure how exposed the sensitive assets are, as we are able to count the number of vulnerabilities that can be exploited, first in absence of our solution, second in presence of our solution.

The solution that we propose makes a static analysis difficult and potentially incomplete. There are different limitations brought by both the approach and the implementation. The approach itself uses techniques to defer the introduction of the protection code to the last moment. While it brings more flexibility to have a late change or a hot-swap change of protection libraries, this prevents a static analysis based on source code or prior the final deployment to correctly analyze the application. In the first case, the source code available from the the static analysis component represents the application with enhanced data type annotations. There is no trace of the validation mechanisms, which are developed separately. In the second case, *i.e.,* the deployment of validation mechanisms, the static analysis can access both the application binary with enhanced data-types and the validation mechanisms that will be executed at several determined points in the application. Even in such case, with a flexible static anlayzer, there is a dose of uncertainty, due to the rate of false positives and negatives that can limit the usefulness of the methodology. Therefore, we use a different technique to evaluate our approach.

The evaluation of our approach is measured by the ability to mitigate security risks while allowing the correct execution of the web applications. To validate the correct mitigation of security vulnerabilities, we apply a rigorous testing to deliberately insecure web applications, and compare the results prior to and after the correction. We verify that our correction does not break the normal flow of the application manually, and by intensively testing the validation library.

We have chosen a black-box approach for testing, as it allows to analyze the potential application attack surface available externally. Furthermore, the aspect-oriented approach has several strategies for injecting validation, such as the static weaving or the load-time weaving of cross cutting concerns (validation code) into the application. A white-box analysis would introduce several layers of complexity, whereas we are only interested in an overall protection rate. There are several tools trying to abuse software inputs to exploit certain parts of a system. Even though the solution better suits web-based applications, with the traditional set of SQL injections, cross-site scripting, cross-site request forgery, *etc.*, it can also be used in regular applications. Tools are either fuzzing tools or web application scanners with a more or less smart behavior to handle stateful requests and to deepen the test coverage.

We have decided to use two specific tools specialized in web application security and audit. They are classically introduced in penetration testing phase to support automated analysis and collect of security vulnerabilities.

4.4.1 Penetration testing tools

Arachni

Arachni [Las13] is a web application security scanner framework. It comes as an open-source Ruby framework to assist testers and administrators evaluating the security of web applications. It provides all features going from web application crawling using a spider module to deep packet analysis using blind SQL module for example. The application is able to activate modules for common web application vulnerabilities: code injection (through several channels), cross-site scripting, cross-site request forgery, path traversal, remote and local file inclusion, SQL injection, *etc.*.

W3AF

W3AF [Ria11] is a web application attack and audit framework. It helps in finding and exploring web application vulnerabilities written in Python. It comes with several modules to crawl the application and analyze common vulnerabilities. The set of vulnerability is similar to the ones covered by Arachni, with additional modules.

4.4.2 Analysis

We have applied our methodology to concrete insecure web application, using the *wavsep* project [Che12] to support our tests. The project we have chosen is designed to evaluate web application security scanner and deliberately contains vulnerabilities: path traversal, remote file inclusion, reflected XSS, blind and direct SQL injection, plus additional other tests. The downside of this project is from an analysis point of view. It defines separate projects with JSP pages as a different test case for a given vulnerability. It means there is no specific data model with business value. Hence, the annotations in our case would require several adaptations. An example of such JSP is defined in Listing 12. The code is used to display a form to let user enters information. Then, the code rends the user input within the HTML of the page. The problem occurs from line 25 to line 28: a reflective cross script scripting occurs if one enters html specific characters. For instance, a userinput <*script*>*alert(document.cookie);*</*script*> would render on client's browser and execute. Such a problem has serious consequences on real-world web applications, as one attacker can force a web browser to point to the vulnerable JSP to execute code under attacker's control. It leads to stealing of session of the attacked browser, denial of service targeting other web sites, *etc.*

Listing 12 Vulnerable JSP from the wavsep project.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Case 4 - RXSS via tag injection into the scope of an HTML comment</title>
</head>
<body>

<%
if (request.getParameter("userinput") == null) {
%>
    Enter your input:<br><br>
    <form name="frmInput" id="frmInput" action="Case04-Tag2HtmlComment.jsp"
        method="POST">
        <input type="text" name="userinput" id="userinput"><br>
        <input type=submit value="submit">
    </form>
<%
}
else {
    try {
              String userinput = request.getParameter("userinput");
             out.println("The reflected value is within an html comment: "
                 + "<!--" + userinput + "-->");
              out.flush();
    } catch (Exception e) {
        out.println("Exception details: " + e);
    }
} //end of if/else block
%>

</body>
</html>
```

The evaluation of our methodology with this project is a good example of how we can address input validation in different applications. JSP are transformed to obtain compiled servlets. The methodology has first been defined for web applications with business model to factorize input validation behavior. The *wavsep* project allow us to showcase that even with reflected servlets, we can still benefit from our approach to write validation mechanisms once and apply them at several places in the application. For instance, the project contains 61599 lines of code in JSPs, and 563 lines of code in Java. There are also more than a thousand JSP files (we count 1787 for all JSP files, but the project has developped 1134 JSP files containing vulnerabilities). A manual approach to validate the input of all these JSPs would take ages.

The first action is to add enhanced data-types if necessary to cover cases that are not already defined in our validation library. The enhanced data-types are efficient when applications use a consistent data model. As the wavsep provides no data model, but reflected use of input parameters through the *request.getParameter()* function of the *HTTPServletRequest* object, we have adapted the enhanced data-types and the pointcut interface adaptor module.

The web application uses the following custom types, among others: username, password, target, msgid, transactionDate, minBalance, description, *etc.*. We enhanced the validation library with these types. Listing 13 presents a class validation following our methodology for the *Name* enhanced data-type. The validation consists of a pattern to accept from two to fifteen word characters.

Listing 13 Simple validation for a Name data type (used for username for example).

```java
@Validation(value = DataType.NAME)
public class ValidationName implements ValidationInterface {

    final Pattern p = Pattern.compile("\\w{2,15}");
    @Override
    public boolean doProcess(String str) throws Exception {
         Matcher m = p.matcher(str);
         return m.matches();
    }
}
```

As this project contains too many different JSP files, we have decided to use another mean to indicates the sensitive inputs that we need to protect with the validation library. Therefore, rather than a manual annotation with $@Type$ to indicate the enhanced data-type, we created an aspect. The aspect defines a pointcut that wraps around the *request.getParameter()* calls, as shown in Listing 14 from line 1 to 3. The advice that is executed determines the actual enhanced data-type from the parameter name, from line 7 to 9. Then, it uses the validation library to validate the value, or to detect any conflict between the claimed enhanced data-type and the real value.

Listing 14 Custom adaptation of our methodology to wrap getParameter() inputs for validation.

```java
1  @Pointcut("call(* *..HttpServletRequest.getParameter(..))")
2      public void pointcutGetParameter() {
3  }
4
5  @Around("pointcutGetParameter()")
6  public String wavsepCustom(final ProceedingJoinPoint jp) throws Throwable {
7      String paramName = (String) jp.getArgs()[0];
8      String value = (String) jp.proceed(new Object[] { paramName });
9      DataType type = DataType.valueOf(paramName);
10     validate(type, value);
11     return value;
12 }
```

We provide in Table 4.1 the details of our executions. We launched the test application on a Tomcat servlet container, with and without the validation library, and performed tests through both Arachni and W3af. We retain only the vulnerabilities flagged as critical/high, and don't report the medium or informational issues. The miscellaneous category includes either unclassified vulnerabilities reported as is by w3af, or phishing vector and code injection detected by Arachni.

		Original	Protected
SQL Injection	arac.	214	3
	w3af	107	0
Cross Site Scripting	arac.	185	0
	w3af	119	0
Directory Path Traversal	arac.	662	0
	w3af	231	0
Remote File Inclusion	arac.	120	0
	w3af	0	0
Miscellaneous	arac.	44	0
	w3af	552	278

Table 4.1: Number of vulnerabilities detected by the arachni (arach.) and w3af web application security scanners with a black-box approach on original and protected wavsep application.

To enable our validation library in web applications, we have configured AspectJ to provide a load-time weaving of aspects at the startup of the web application. The results show a reduction of the detected vulnerabilities. Albeit it doesn't mean the application is free of bugs, we can confidently claim that we achieved a pretty high coverage of this specific application. It has been possible as we spent some time analyzing inputs and systematically providing a validation library for all type of inputs. Especially, we have adopted an automated approach that shows how we can adapt our methodology to benefit from aspects.

4.5 Related work

To the best of our knowledge, the solution we develop is the first one that addresses the problem of enforcing input validation through a strict separation between data type definitions and application logic with the given environment we have. However, in the past decade, much research effort has been spent on making web applications and web services more secure. Researchers have focused on detection mechanisms including static analysis, dynamic taint analysis and client-side security mechanisms. In addition to detection techniques, researchers have also worked on techniques to prevent security vulnerabilties. We give an overview of the different techniques below. The static analysis has been widely discussed in previous chapter. Refer to Section 3.7 at Page 70 to get related works for static analysis.

4.5.1 Dynamic Taint Analysis

In contrast to static analysis, dynamic taint analysis checks the program at runtime. In general, approaches based on dynamic tainting assign meta-data to user-supplied inputs. All user-supplied data is set to be tainted. When operations are performed on the input data, this meta-data is preserved. After the sanitization of user-supplied data, the data is set to be 'untainted'. This allows the detection if untrusted data reseaches a sensitive sink.

Nguyen-Tuong [NTGG^{+}05] and Pietraszek [PVB05] worked both independently from each other on dynamic taint propagation. They proposed an extension to the PHP interpreter that tracks tainted input data. The extension proposed by Pietraszek can either prevent the execution of code or sanitize the input. The approach proposed by Halfond et al. [HOM06] introduces positive tainting, in this case, only trusted data are tracked.

Dynamic tainting has also its problems. First of all, the technique has a relative large overhead in terms of performance. Moreover, the input data has to be untainted after a sanitization function. As [BCF^{+}08] shows, implementing sanitization functionality is far from trivial. Furthermore, preventing second order attacks is difficult as it requires the tracking of data through persistent data stores.

4.5.2 Client-Side Security Mechanisms

Unfortunately, not all developers of web applications protect effectively and in-time their applications against input manipulation attacks. It exists some client-side solutions to protect users of these web applications. Several approaches exist that aim to provide client-side protection, which are components in the middle (in between the client application and the server application).

In [IEKY04], the authors propose a client-side proxy that detects the use of special characters such as '<' in HTTP traffic. When the proxy detects that the application response reflects these presumably malicious requests, the traffic is blocked. Also Noxes [KKVJ06] is based on the concept of a client-side proxy firewall. However, this work aims to improve the user experience of personal firewalls by introducing some heuristics. In [VNJ^{+}07], the authors propose the use of browser plugin that uses static and dynamic tainting techniques to check whether sensitive data are sent to a different domain than where the Javascript code is downloaded from.

BEEP [JSH07] tries to achieve client-side security *by design*. It is a policy-based mechanism that forces the browser to execute only those scripts that are explicitly allowed to run as specified by the policy.

4.5.3 Prevention Techniques

Besides the solutions to detect code injection vulnerabilities, there exist several approaches that prevent code injection vulnerabilities based on the sanitization of data. Data sanitization is the process of transforming data such that the resulting data only contains safe characters. In contrast to the traditional practice of sanitization checks that a developer has to implement in an ad-hoc way, these frameworks and/or language extensions ensure that documents and/or queries are automatically protected. Thus, injection vulnerabilities are prevented *by construction* or *by design*.

William Robertson et al. propose in [RV09] a framework that statically enforces a separation between the structure (code) and content (data) of a software. In the framework, an (X)HTML document is represented by nodes that are connected to each other. The document is a tree of nodes and each node is an instantiation of the *Node* type. As a result, the document is strongly typed. Once the document is constructed, a rendering function converts the document into a string that can be sent to the client. The rendering function automatically sanitizes unsafe characters. The framework also allows developers to specify dynamic SQL queries using an embedded domain-specific language. The only way to execute SQL queries and construct documents is through the interfaces provided by the framework. In this way, sanitization is enforced.

In [JBGP10], Johns et al. propose a datatype to enforce the separation between data and code. With this approach, the developer is forced to use the ELET datatype to construct foreign code. Once the developer has specified the foreign code using the ELET datatype, a pre-processor translates the foreign code to an API representation in the hosting language. Data provided by the hosting language can be inserted in the foreign code by using a special function. The main limitation of this approach is that the dynamic construction of foreign code within the foreign code (e.g. JavaScript's eval function) is not supported. Moreover, the dynamic creation of identifier tokens in the foreign language is not supported. In [JB07], Johns et al. proposes a mechanism to secure web applications implemented using an interpreted language. A pre-processor marks foreign code found in the source code as legitimate. After the work performed by the interpreter, a post-processor identifies all the foreign code that has been injected by the user/attacker and masks it such that it will not be executed. The main problem with this approach is that the pre- and post-processors introduce false positives and false negatives.

These approaches are related to concrete implementations. We consider in this work that the problem comes from a insufficient typing of the data. Type system of the programming language used is important as it can influence the detection of type errors. For instance, if a developer uses type system which is considered to be sound, a well-typed program would not cause type errors.

4.5.4 Input Validation

Several web application frameworks (and persistence layer frameworks) support input validation through the use of annotations. Frameworks such as Spring MVC [Sou11], Hibernate [JBo11] and Struts 2 [Fou11] support a limited set of input validation types. Hibernate is based on the JSR 303 Bean Validation standard [BP09]. In contrast to our work, the set of possible input validation types cannot be extended. Furthermore, these frameworks do not support the enforcement of validation functions, e.g. a developer is not forced to validate input. The solutions proposed in [BP09] and in [Hoo05] allow a complete decoupling of validation code and application logic. However, also these solutions do not force the developer to specifiy the inputs along with the types resulting in less secure applications and a decreased level of quality of data.

Besides frameworks supporting input validation, there exist web application firewalls that are capable of performing input validation. Web application firewalls are placed in front of the web application or web service and all HTTP traffic is routed through the firewall. A firewall can block known malicious requests (blacklist-approach) or only allow known benign requests (whitelist-approach). Scott et al. proposed in [SS02] to secure web applications using web application firewalls. Since then, the technique has been commercialized and many vendors offer application-level firewalls as appliances [Imp11, Inc11, Tru11]. In contrast to our approach, web application firewalls do not allow to establish and maintain consistency between the input validation specification and the application code. Moreover, application-level firewalls support a very limited set of input types which is not extensible.

The correction of input manipulation often includes sanitization. The correctness of such sanitization process is important, as an incorrect solution would let the developers think they are protected. This problematic is addressed in Balzarotti et al. [BCF+08]. They introduce an analysis of the sanitization process to detect incorrect or incomplete sanitization. They provide for such purpose a tool called Saner.

4.6 Language approach for security modularization

We present a novel method and tool to prevent from the major cause of vulnerabilities to applications nowadays, which is the acceptance of malicious input. By adding simple and precise type annotations to existing code, The solution brings a lightweight approach to enrich type information concerning the expected input for an application. The solution derives validation functions that are modularly integrated into existing code. The main originalities of our solution can be summarized as follows

- Non-invasive use of aspect-oriented programming, which discharges the developers from learning a new programming paradigm
- High degree of automation and the increase program security with minor effort. Moreover, developer applying our innovation does not require security knowledge
- Extensibility: allowing developers to create business-specific enhanced data types and their validation aspects
- Modular integration of new security functionality without disrupting existing code
- Security is adopted by design, considering that annotations to all input parameters must be provided, but, in order to provide more flexibility to the solution, an administrator can disable the obligation to annotate all code.

Moreover, we have also created internally a small demonstrator that proves the feasibility of the concept.

Part II

Modularization of constructive security

The modularization of constructive security can be simplified to the modularization of business security concerns. These are concerns that are specified to make the application working in its environment. Generally, the concerns are not trivial and developers are prone to introduce either bugs or vulnerabilities as they are not familiar with these concepts. The type of security properties affected is large, and usually involves security mechanisms that needs to be correctly defined and injected in the application. One vision of security architecture for distributed systems *Authentication, Authorization and Accounting* (AAA) gives a first glance of properties. But we cover a larger range, such as privacy, confidentiality and integrity of messages, *etc.* These concerns are often exposed through security policies.

In the following, we present two specific contributions showing modularization of security properties at two different layers of the stack for distributed applications. The first contribution introduces a security protocol for message security, including transmission of token securely, confidentiality and integrity of transiting data. This particular protocol targets light and said-to-be flexible web service using HTTP transport layer as an applicative layer: RESTful services. The second contribution present an architecture to provide seamlessly integrated privacy within cloud platform, which ease the integration of such security concerns for both the platform provider and the application developers that will deploy application on the cloud.

Both of these approaches are in the context of distributed systems, hence systems executing in different environments that communicate through communication mediums. All of these components are possibly under different administrative domains.

Chapter 5

Service framework modularization for message-based security

The modularization of security can be achieved by changing the way security is injected into the application. Instead of waiting a complete definition of security inlined within the application, we can obtain a flexible and modular security by letting the platform inject the security in pre-defined points. This notion is often referred to as the inversion of control pattern, that let containers decide and manage the orchestration of dependencies and cross-cutting concerns. It is then possible to define security and provide late binding, taking in account custom needs in the specific context the application is executing in.

In this contribution, we propose to introduce a new message security model for RESTful services that allows to carry authentication tokens and protect resources in a fine-grained manner. The security properties brought by this approach can be easily introduced by security policies. To enhance the flexibility in transformation, we propose a module that intervene in the web service framework layer, or as a reference monitor.

The security and dependability of cloud applications require strong confidence in the communication protocol used to access web resources. The mainstream service providers nowadays are shifting to REST-based services in the detriment of SOAP-based ones. REST proposes a lightweight approach to consume resources with no specific encapsulation, thus lacking of meta-data descriptions for security requirements. Currently, the security of RESTful services relies on ad-hoc security mechanisms (whose implementation is error-prone) or on the transport layer security (offering poor flexibility). We introduce the REST security protocol to provide an end-to-end secure service communication, and explain to which extent it allows flexible security.

5.1 Introduction

With the growing interest of cloud computing, systems are getting inter-connected faster, as applications and cloud API's make intensive usage of RESTful services to expose resources to consumers. There has been a shift from SOAP-based services to more lightweight communication, based on REST which allowed a number of advancements in the way resources are used on the web. As REST web services are self-described, resources can be manipulated through a set of verbs already provided in the communication protocol, accelerating the adoption of the REST philosophy. On the other hand, REST suffers from the absence of meta-descriptions, specially concerning security requirements.

Different solutions have been developed to provide a common way to address service description and communication. For SOAP-based web services, the standard defines envelopes to transmit requests and responses. In contrast, the REST concepts coined by Roy Fielding in his Ph.D. dissertation [Fie00] simplify access to web services by reusing existing and widespread standards instead of adding new layers to the communication stack. The reuse of HTTP protocol contributed to the large industry adoption of RESTful services, supported by the simple CRUD set of operations (Create, Read, Update, Delete).

RESTful services suffer from the lack of a specific security model, unlike SOAP-based services which rely on the message security model defined in WS-Security [OAS06] standard. Especially, the security of existing RESTful API's rely on transport layer security and on some home-made message protection mechanism. The former protects efficiently point-to-point communication channels, but becomes a burden for mobile systems, as the TLS channel need to be frequently reset. It is also difficult to have multiple parties involved in a secure communication, as each of the peers would require to rely on each other entity. The latter can be error-prone, as security protocols are difficult to design and implement. Thus, a custom security might lead to inconsistencies, incompatibility with other standards, *etc.*.

In this chapter we provide a security protocol to make message security implementation as lightweight and efficient as possible, and yet to respect the REST principles. We show how message signature and encryption can address communication security for RESTful services at a fine-grained level. We then present the interest of such protocol in presence of multiple stakeholders spanning several administrative domains. We present results of the benchmark we conducted on our implementation and compare it to the equivalent realization using SOAP and WS-Security.

The chapter is organized as follows: Section 5.2 presents the motivations for such an approach, Section 5.3 introduces the REST security protocol and the threat model we aim to mitigate. In Section 5.4, we position our protocol with regards to WS-Security via a benchmark. Then we discuss related works in Section 5.5 and conclude in Section 5.6.

5.2 Motivation

REST-Security provides tools to enhance flexibility of many use cases in which RESTful services are exposed. In a nutshell, it allows business owners to define security properties for application, disregarding the actual implementation of the web services layer. In this section, we present the motivation of such a protocol through two different use-cases, in which REST-Security provides key advantage, by removing unnecessary specifications from protocols, or by facilitating enforcement of multi-party security policies.

REST security protocol is a pendant of WS-Security, which already provides a set of tools and methods to encapsulate security behavior in dedicated modules. It is possible to define the security of messages, per service. The service framework is responsible to interpret the configuration of security and verify incoming messages to detect policy conflict, but also to take action and modify outgoing messages to add required security tokens or perform security transformations.

The novelty comes with the flexibility that we introduce with the new security protocol, but also in the way we envision the transmission of security metadata across the application. Usually, the security of messages is configured at the service framework level. The notion of protection is therefore limited to the transport and the service framework itself. The business application that handles the request has no specific information on the origin of the message, nor its validity. Some frameworks propose the propagation of principal identity to the business application, but the notion of integrity, confidentiality in the transmission as well as the signature is lost in transit. Such behavior renders the handling of security properties at the business level problematic. For instance, one might have a business rule of non-repudiation with proof of receipt that certifies the receiver has received the message. The acknowledgment contains cryptographic proofs that are carried along the message, and then generally interpreted by the service framework.

We would like to introduce at this stage another layer of flexiblity, that provide security proofs to the business application, disregarding the underlying technology used for message security. The REST security protocol and web service security framework would provide accurate concrete mechanisms, that one can use to define security properties. The security properties are propagated to the business application, with the use of aspect-oriented techniques. We further discuss the mechanisms in the perspective part of the conclusion in Section 8.

In the following, we intend to present two situations in which a flexible security protocol would benefit.

5.2.1 OAuth 2.0 token protection

The OAuth 2.0 web authorization protocol allows services to act on behalf of users when interacting with other services. It avoids sharing username and passwords across services, thus, in principle protecting users from several threats. However, it is known that the implementation of this kind of authorization protocol is complex, and potentially leads to vulnerable web services.

OAuth 2.0 proposes a multi-party environment in which the client (resource owner) uses its credentials to request protected resources held by the server. If a third-party wants to access these protected resources in accordance with the resource owner, then the resource owner has to share its credentials with the third-party. This situation may lead to several undesired behaviors, such as resource owner's credential's multiple storage (at the third-party location), complete access to the protected resources (no limitation of rights to third-party) or difficulty of right's revocation (the resource owner needs to change its credentials). OAuth's purpose is to mitigate these concerns by granting access without credentials sharing.

The initial protocol has been defined by Hammer et al. in a standard defined by the IETF as OAuth 1.0 [Ha10]. The current version is OAuth 2.0 [Rea12] and is a disruptive evolution in which major companies have been involved to cover different use cases. The original author Eran Hammer withdrew its name from the specifications, and OAuth 2.0 presents a framework rather than a security protocol. The development is already mature and target industrial use-cases, which is the reason why we chose to focus on OAuth 2.0. For readability reasons when we are referring to OAuth 2.0 we will simply speak about OAuth, and when needed we will specify the version.

The OAuth protocol defines four different roles:

- The resource owner is an entity that holds protected assets. This entity is capable of granting access to the assets under its control.
- The resource server is the server that hosts the resource owner's protected assets.
- The client is the third party entity that needs to access the protected assets on behalf of the resource owner.
- The authorization server is the server that issue authorization claims and generally verify authentication and authorization of the different entities involved.

The general flow defined by OAuth is depicted in Figure 5.1. The flow starts when a client (mobile application, third-party service, any application which needs to access resources of a resource owner) needs to access resources on a resource server that he doesn't own. To access these resources, the client request an authorization to the resource owner (step A). The resource owner generally gets an authorization grant by getting one from the authorization server and send it back (step B). As an alternative, the client can directly request the authorization grant from the authorization server. In step C, the client requests an access token by authenticating with the authorization server and presenting the authorization grant. If the authorization grant is validated by the authorization server, an access token is issued and sent back to the client (step D). At this point, the client has an access token that indicates that a resource owner allows the client to access a certain number of its resources on a specific server. The client can use this access token to access the resources (step E). The resource server validates the token an send back requested information (step F).

Cherreau et al. [CDR^{+}13] present relevant problems faced by OAuth 2.0 implementations. For instance, OAuth uses the concept of bearer token that is defined as "A security token with the property that any party in possession of the token (a "bearer") can use the token in any way that any other party in possession of it can. Using a bearer token does not require a bearer to prove possession of cryptographic key material (proof-of-possession)". The specification also details "TLS (Transport Layer Security) is mandatory to implement and use with this specification; other specifications may extend this specification for use with other protocols". It means that to convey bearer token, the services need to communicate through a secure channel to avoid disclosure of tokens, thefts, replay attacks, *etc.*

The author of OAuth argues against using only transport layer security. It provides a potential false sensation of security [Ham10]. For instance, there are issues occurring during implementation, like several developers who do not try to understand the rational behind transport layer security and break the hierarchy of trust (for example, they try to disable verification of certificate authority). Such activation would void any effort in protecting tokens. In addition, several attacks target the transport layer security with success. For instance, the latest attack use a simple trick to extract content from a secure HTTPS channel [GHP13]. BREACH (Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext) approach takes advantage of HTTPS compression to byte by byte extract some part of the user provided content,

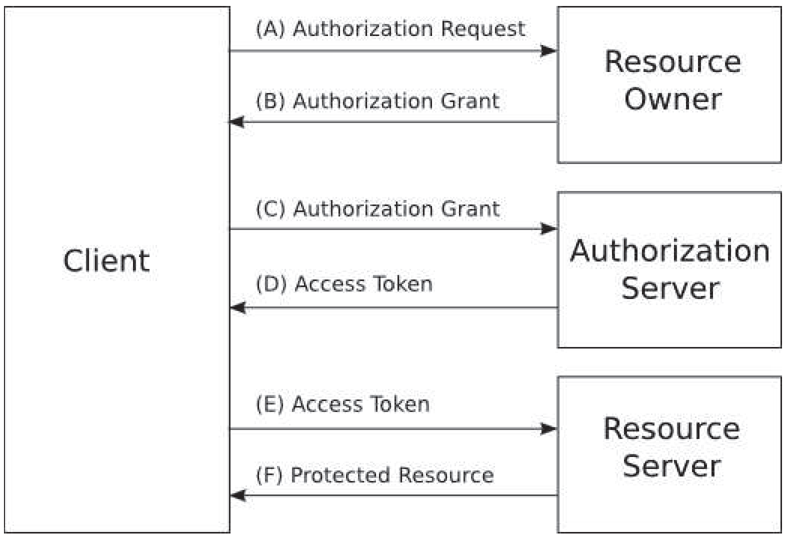


Figure 5.1: OAuth generic flow

through HTTP responses. They adapted the approach from another approach which was targeting HTTP requests, the CRIME security exploit [RD12].

With the REST security protocol framework, we are given new tools to protect these sensitive tokens, and thus propose an automatic protection of certain tokens along the HTTP messages. We do not intend to replace completely the transport layer security required in the protocol, but we aim to extend use cases in which you can transmit securely tokens to propose new possibilities for service providers and consumers. For instance, the protocol can handle other security constraints. We can think of carrying encrypted basic authentication tokens, signed P3P claims, or even convey authorization token decisions. We position the REST security protocol as an alternative to transmit securely data over the wire.

5.2.2 Flexible enforcement of security properties with cross-domain collaboration

The definition of the protocol comes to fill a gap in existing scenario involving RESTful services. The definition of security, on a fine-grained basis is hardly possible, and makes the transmission and securisation of tokens difficult. It leads to situations in which specifications to newly defined protocols force the usage of secure transport channel, like we have seen in the previous section. In other situations, the secure transport channel is not sufficient to respect security policies and attach security properties to flows. For instance, we have introduced in [DGM+10] a loan origination scenario in which several parties have to collaborate to complete a business process. The parties have their own specific environment, with their own security adaptations. In order to collaborate, they need to agree upon a common process to handle security. We discus the security properties and requirements that affect this system in [SdOS12].

For the specific needs that we have in our scenario, we have developed a security policy language described in [DSI+12a]. HiPoLDS stands for A Hierarchical Security Policy Language for Distributed Systems. It has been designed to enable the specification of security policies in distributed systems in a concise, readable, and extensible way. HiPoLDS design focuses on decentralized execution environments under the control of multiple stakeholders. It represents policy enforcement through the use of distributed reference monitors, which control the flow of information between services. We are going to see in this section how the REST security protocol protocol would assist the scenario's needs in this kind of environment.

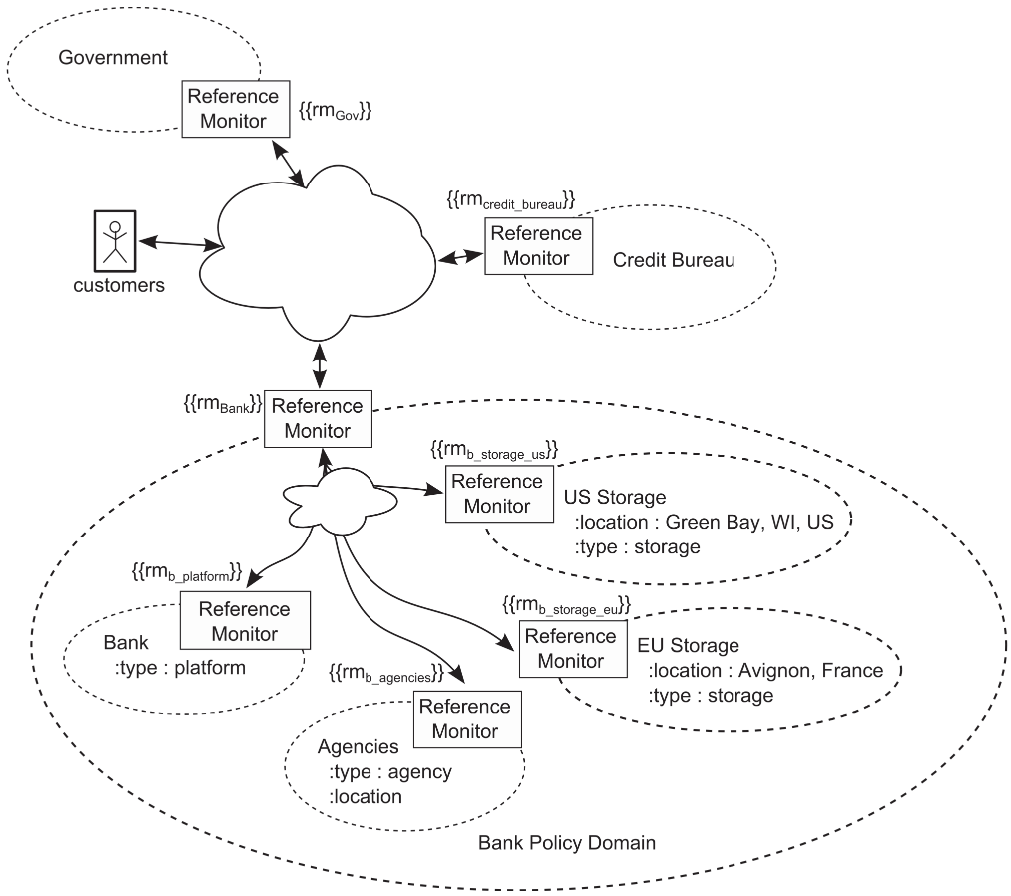


Figure 5.2: Domains of the scenario with hierarchical reference monitors per domain

The domains are presented in Figure 5.2. There are a total of four domains: the customer domain, the government domain, the credit bureau domain, and the bank domain. The security rules are handled by reference monitor in this approach. The reference monitors can either verify the status of messages going through, or modify the messages to respect the security policy. For example, the top level runtime monitor in the Bank domain hierarchy is referred to as $\{\{rm_{Bank}\}\}$. The monitor has no visibility of the architecture of the outside world, but it can detect some properties carried along the messages, or already known by the monitor (i.e range of addresses corresponding to a specific actor). With respect to the internal domain, the monitor $\{\{rm_{Bank}\}\}$ is able to communicate with its direct sub-components, but might not know their architecture. This grey-box view is important to define global rules that are then enforced locally.

The bank reference monitor needs to address several security requirements according to a policy that we have defined:

- Non repudiation : An external party shall receive a proof of receipt when sensitive loan operations are performed
- Confidentiality : An incoming or outgoing loan resource needs to remain confidential between requester and receiver.
- Signature : an incoming or outgoing loan object needs to carry a proof of authenticity
- Separation of duty : the platform needs to guarantee that at least two individuals are involved in the verification of loans.
- Logging : The runtime monitor shall log incoming and outgoing messages that has for origin or destination a domain outside the bank

Such security policy description is translated in an HiPoLDS rule, which is a domain specific language. The detail of the language is explained in [DSI+12a]. For example, for a message that contains loan information, the message should carry proof of integrity and the resource itself should only be disclosed to the Bank and the Government. Such a requirement would translate to the following HiPoLDS *abstract* rule:

$$\begin{aligned}&\texttt{m} : \mathit{message},\ \texttt{x} : \textit{loan-info}\ \epsilon\ \texttt{m}.\mathit{contents}\\ &\rightarrow \texttt{x}\ \mathit{is}\ \texttt{confidential}(\mathit{Bank}, \mathit{Government}),\\ &\ \texttt{m}\ \mathit{is}\ \texttt{integrity\_verified}\end{aligned} \tag{5.1}$$

The policy language specifies the security requirements, in a hierarchical way, and for distributed domains. These domains are not all under the same administrative domains. They have to collaborate all together to process messages, add security proofs when they manipulate data, but they would not put a blind trust in their partner. For instance, if a document requires the successive validation of three different actors (a bank, a government agency, and a customer), each of the actors would put a signature and transfer the document to the next actor. Such a situation requires the signature to be attached to the document. The REST security protocol allows to transmit the document with several security proofs attached to it without additional processing. The tokens in this case would convey multiple signature for the different actors at the same time. The elegant approach comes from the possibility for the different actors to simply get the document, sign it, and transfer it to the next actor. They don't need to interpret signature of the previous actors. They don't need to verify authenticity of other signatures: as the signature is attached to the resource (the document), any party can verify the validity of the document at any time. The tokens, transmitted in headers of the documents are flexible enough to allow quick signature and verification from the different involved parties. In addition, the security verification and transformations can be introduced directly by the reference monitors.

Another possibility with the REST security protocol is to encrypt partial content of resources. In the context of cross-domain business processes, it allows new interactions such as a first actor which partially encrypt a document for an intermediate, and the remaining of the document is encrypted for a different recipient. A second actor would affix a signature to validate the message authenticity before transmitting it to the other actors. These behaviors, that one can describe with HiPoLDS security policy language can be enforced transparently thanks to reference monitors. We describe in [DSI+13] the details of the enforcement architecture.

5.3 REST Security

In the following, we present the materialization of the protocol to secure messages and resources in case of RESTful services. We provide *Encryption*, *Signature* and their combination. We do not aim to provide an equivalent of *Secure Conversation* from the WS-Security standards for RESTful services, as it relates to some transport layer security for HTTP which is already addressed in protocols such as TLS.

5.3.1 Message Security Model

We specify an abstract message security model based on confidentiality and digital signatures to protect RESTful messages. The associated threat model is exactly the same as the one described in Web-Service Security standard [OAS06]: "The message could be modified or read by attacker or an antagonist could send messages to a service that, while well-formed, lack appropriate security claims to warrant processing". For instance, a malicious attacker can intercept messages on any intermediary between peers. We want messages to carry tokens for non-repudiation (via digital signatures), to provide data confidentiality by encrypting its content, and to have replay attack protection.

The need to develop our own security for RESTful services comes from the frequent possibility to have man-on the middle attack on secure channels. The mechanisms to provide confidentiality, integrity and non-repudiation for most of the RESTful services exposed by the industry rely on transport layer security over the application protocol, such as HTTPS. There are several attacks, or even programmatic mistakes that render the layer less secure. For instance, there has been high coverage of SSL-attacks such as BEAST or CRIME that render possible plaintext recovery from a partial controlled environment, as well as other attacks such as SSL stripping or other manipulation regarding spoofing or man on the middle attacks [Jee13]. Georgiev et al. present a paper in which they explain some basic failure of SSL-validation in general applications, which is the fundation step of the whole PKI infrastructure [GIJ+12].

Although secure channel provide a first defense line against most eavesdropping use cases, the client and server can not guarantee the security from end to end, but rather from point to point. In a multi-party environment, which is a frequent use case in modern computing communication and routing of internet, several hops trust each other to transmit the information from one location to another one. The secure channel blur the intermediates to deliver the message to a server, from which we don't know the exact processing. With our approach, it would be possible to intermediate to add secure tokens to messages, in addition to already existing ones. It make senses in hierarchical environments that require high control on data coming in and out. For instance, security policies in large companies can benefit from this approach by allowing enforcement of security properties at different places in the company infrastructure landscape. We have already presented such need in the previous section.

5.3.2 PKI-based message exchange

We assume that a PKI landscape is in place and that certificates have been exchanged between clients and servers prior to the communication. In this way we are able to transmit a certificate identifiers within the messages instead of full certificates, what would bring unnecessary overhead.

In order to distinguish a certificate on both client and server sides, we rely on a unique identifier, called *Certificate ID*, known to all entities. The *Certificate ID* is the aggregation of a serial number and an issuer name. The RFC 5280 [IET08] specifies that serial numbers "MUST be unique for each certificate issued by a given CA, i.e., the issuer name and serial number identify a unique certificate". The issuer name in our case can be represented by the Distinguished Name of a X509 certificate.

5.3.3 The REST Security principle

The principle of our protocol is to propose secure communication at the message level with the minimum overhead: we try to respect the philosophy of RESTful services and to reuse HTTP protocol to its full advantage. For example, we take into account the specificity of HTTP verbs in the design of the protocol. The REST security protocol is closely related to the WS-Security standard: it proposes a fine-grained approach to provide authenticity, non repudiation, and confidentiality to messages. But the approach targets another type of service. We claim that our approach is complementary to provide consistent application of security policies, disregarding the type of service being addressed. When comparing both approaches, we can highlight the reduced development effort and also less computation at runtime. This is a consequence of the optimization in the message size while we have performed, yet respecting the compatibility with service's definition and implementation.

We propose a set of HTTP-headers for transmitting meta-data, unlike WS-Security which modifies messages to add its own container describing the security meta data. The headers are described in Table 5.1. They start with a prefix "X-JAG" to distinguish them from other application headers. The main difference with the WS-Security approach, is that we are agnostic about the information format. WS-* services use a strict approach to determine the transformations of XML-based messages to ensure the correct handling by interpreters at both sides. In our approach, we consider the information as a set of multiparts, and protocol headers. It allows us to gain flexibility in terms of fine-grained signature and encryption of attached documents, and/or to restrict visibility of a number of headers.

In the following, we present the REST security protocol process. For illustration purposes, we present the interaction trace produced by the request of a RESTful service in the Listing 5.1. A client requests customer information to the service and expects a JSON-encoded result. One can notice the expected result can be in any format accepted by the server (*e.g.,* XML, YAML,

Header keys	**Value**
X-JAG-CertificateID	Unique identifier for a certificate
X-JAG-DigestAlg	Algorithm used to obtain digest
X-JAG-DigestValue	Value of the digest(s)
X-JAG-SigAlg	Algorithm used to obtain the signature
X-JAG-SigValue	Value of the signature(s)
X-JAG-EncAlg	Algorithm used to encrypt headers and messages' part
X-JAG-EncKeyAlg	Algorithm used to encrypt the symmetric key
X-JAG-EncKeyValue	Encrypted value of the symmetric key
X-JAG-MultiParts	Designation of headers and messages' part

Table 5.1: REST security protocol headers

plain text, audio file, binary content, *etc.*). The response produced by the application server starts at line 6 .

```
 1 GET /customer/123 HTTP/1.1
 2 Accept: application/json
 3 Host: 127.0.0.1:8080
 4 Connection: keep-alive
 5
 6 HTTP/1.1 200 OK
 7 Server: Apache-Coyote/1.1
 8 Content-Type: application/json
 9 Content-Length: 77
10
11 {"Customer":{"firstname":"Gabriel","id":123,"lastname":"Serme","title":"Mr
       "}}
```

Listing 5.1: RESTful request and response

5.3.4 Message Signature

Providing digital signature along with requests gives confidence on the data being transmitted. A server might need information on the authenticity of a message to launch internal orders and to render the service correctly. A digital signature brings non-repudiation: a requester cannot deny the request. Also, the service cannot later repudiate the response if it includes signed token linked to the initial request. Additionally, digital signature protects from unintentional or malicious modifications during the transmission.

Algorithm 1 presents the steps to attach signature information to the message after a "digest then encrypt" processing. It starts with a message m or part of it, with: the digest algorithm, the signature algorithm, the *Certificate Id* of the sender, and the private key of the sender. The algorithms can be decided by the sender itself, or imposed by the server policy. In our implementation, we allow the client to decide about the algorithm to be used, but the server can deny access if its policy considers the protection to be insufficient. We have defined a "digest then encrypt" function over the message payload, security parameters, and header information. The digest always takes as input the timestamp to obtain a different value over time and thus prevent future replay attacks. The algorithm vary slightly depending on the concrete signature algorithm. The values are then attached to the message along with algorithm information.

Algorithm 1 Signature of REST messages

```
Require: $m$ is a message, $sig$ is a signature algorithm name, $dig$ is a digest algorithm name, $cid$
    is a Certificate Id, $pk$ is the sender private key, $urlpath$ the requested path, $hds$ are headers
    element to protect, $ts$ is the current timestamp
  $dv \leftarrow$ digest($m.payload$, $ts$, $dig$)
  $url \leftarrow$ ''
  if $m$ is a request then
      $url \leftarrow$ urlpath
  end if
  $bytes \leftarrow$ concat($dv, url, sig, dig, cid, hds$)
  $digValue \leftarrow$ digest($bytes, ts, dig$)
  $m.sigValue \leftarrow$ encrypt($digValue, sig, pk$)
  $m.\{url, sig, dig, cid, hds, ts\} \leftarrow url, sig, dig, cid, hds, ts$
```

In Algorithm 2, we present the signature verification function. It starts from a message m, or part of it, and with the public key of the sender. The steps are the reverse of the previous "digest then encrypt" algorithm. We first calculate the digest value of a set of headers and the payload. Then, we retrieve the digest value calculated by the sender. The encrypted value is transmitted along with the message, on a specific header. When we decrypt the value, we are then able to detect any corruption in the payload and headers but also to guarantee message safety and authenticity, as it has been digitally proved by the sender.

Algorithm 2 Verification of REST Signature

```
Require: $m$ is a message, $Pk$ is the sender public key
  $dv \leftarrow$ digest($m.payload, m.ts, m.dig$)
  $bytes \leftarrow$ concat($dv, m.url, m.sig, m.dig, m.cid, m.hds$)
  $calculatedDigest \leftarrow$ digest($bytes, m.ts, m.dig$)
  $retrievedDigest \leftarrow$ decrypt($m.sigValue, m.sig, Pk$)
  if $retrievedDigest \equiv calculatedDigest$ then
      return true
  end if
  return false
```

The Listing 5.2 presents a HTTP trace with concrete headers and payload value. The request starts at line 1 and the response starts at line 10. We can observe for example that message request is issued by a sender identified as the 4102^{th} certificate issued by the CESSA Authority. This sender protects the request of the customer 123. The response is given by another peer,

```
 1 GET /sign/customer/123 HTTP/1.1
 2 Accept: application/json
 3 X-JAG-CertificateID: CN=CA CESSA, <...>O=SAP Labs France, C=FR;4102
 4 X-JAG-DigestAlg: w3.org/2000/09/xmldsig#sha1
 5 X-JAG-DigestValue: 2jmj7l5rSw0yVb/vlWAYkK/YBwk=
 6 X-JAG-SigAlg: w3.org/2000/09/xmldsig#rsa-sha1
 7 X-JAG-SigValue: CwgrRTaC0oGBMpLPF6m<...>+gjtCMnuC+2svEdI5zJvITbM=
 8 Host: 127.0.0.1:8080
 9
10 HTTP/1.1 200 OK
11 Server: Apache-Coyote/1.1
12 X-JAG-CertificateID: CN=CA CESSA, <...>O=SAP Labs France, C=FR;4
13 X-JAG-DigestAlg: w3.org/2000/09/xmldsig#sha1
14 X-JAG-DigestValue: RUAYhPTuXqwChvIGrclAyRtA22Y=
15 X-JAG-SigAlg: w3.org/2000/09/xmldsig#rsa-sha1
16 X-JAG-SigValue: pmpc347XG/8a9QIFWYaHHsbt79hCwF<...>G/buHnjsHQvZhaggilRuM=
17 Content-Type: application/json
18 Content-Length: 77
19
20 {"Customer":{"firstname":"Gabriel","id":123,"lastname":"Serme","title":"Mr
       "}}
```

Listing 5.2: Signed request and response

identified as the 4^{th} certificate issued by the CESSA Authority, on line 12. The request and response are here signed, which allows the party consuming the message to verify the identity of the producer and the validity of the security token, to detect if the message has been tampered with. A replay attack can be avoided by binding the messages to elements with unique characteristics: MAC, timestamp , session related nonce, *etc.*.

5.3.5 Message Encryption

Message encryption provides confidentiality to sensitive assets so that no eavesdropping and data modification happen during messages transmission. In requests, several assets are transmitted, such as payload, session headers in cookies, etc. In our approach, we focus on payload and header protection mainly. We envisage extensions to address parameter encryption in GET requests in future versions of the protocol. The encryption has the property to modify the payload and headers, unlike signature which needs read-only access to the message. The encryption mechanism is also process-intensive.

The Algorithm 3 processes the payload of a message, or part of it for encryption. The PKI environment gives us mechanisms to share information between actors: the public and private keys. However, asymmetric algorithms are too heavy in order to perform an encryption on large amounts of data. Instead, we generate a symmetric key for encryption, using the function $generateSymmetricKey$ that takes two parameters: A symmetric algorithm like AES with indication on the exact parameters, and the current timestamp that will be used to generate the symmetric key. This second parameter can be seen as a salt value. The generated key is small enough to be encrypted with an asymmetric algorithm and sent with the message. Thus, the message contains an encrypted symmetric key for the receiver, the encrypted payload, and several headers expressing the algorithm used for encryption.

Algorithm 3 Encryption of a REST message

```
Require: $m$ is a message, $P_k$ is the receiver public key, $enc$ is a symmetric algorithm name,
  $aenc$ is an asymmetric algorithm name, $hds$ are headers element to protect, $ts$ the current
  timestamp
  $skey \leftarrow$ generateSymmetricKey($enc, ts$)
  $m.payload \leftarrow$ encrypt($m.payload, skey$)
  for all $name, value \leftarrow hds$ do
    $hds[name] \leftarrow$ encrypt($value, skey$)
  end for
  $m.keyValue \leftarrow$ encrypt($skey, aenc, P_k$)
  $m.\{enc, aenc, hds, ts\} \leftarrow enc, aenc, hds, ts$
```

The Algorithm 4 presents the reverse operation with respect to the above algorithm, to be executed on the receiver side. The procedure is performed on an encrypted message m or part of it. The message usually contains meta-information about encrypted parts and algorithms used for key encryption and data encryption. Otherwise, these information should result of a previous agreement between the sender and the receiver. To decrypt the data, the receiver retrieves the symmetric key and uses it to replace the headers and the payload.

Algorithm 4 Decryption of a REST message

```
Require: $m$ is a message, $p_k$ is the receiver private key
  $skey \leftarrow$ decrypt($m.keyValue, m.aenc, p_k$)
  for all $name, value \leftarrow m.hds$ do
    $m.hds[name] \leftarrow$ decrypt($value, m.enc, skey$)
  end for
  $m.payload \leftarrow$ decrypt($m.payload, m.enc, skey$)
```

The Listing 5.3 presents a HTTP trace where the request does not contain custom information apart from the *Certificate Id*. The service has been configured to send back all messages encrypted. The service then processes and encrypts the message content for the requester. In the Listing, the payload is protected and no eavesdropping can be performed during the transmission. The protection mechanisms described in the previous section for replay attacks are also apply here.

5.3.6 Signature and Encryption

Signature combined with encryption is an important feature. Signature alone brings non-repudiation to the system, but an attacker can still read the content of messages and remain unnoticed. Providing encryption-only brings data confidentiality, but do not prevent against data tampering: any intruder can replace the payload and security tokens with its own, as there is no binding

```
GET /encrypt/customer/123 HTTP/1.1
Accept: application/json
X-JAG-CertificateID: CN=CA CESSA, <...>O=SAP Labs France, C=FR;4102
Host: 127.0.0.1:8080

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-JAG-CertificateID: CN=CA CESSA, <...>O=SAP Labs France, C=FR;4
X-JAG-EncKeyValue: RHvEjpmkt2QF3ZPCtqFbflDzA48<...>/
    UYNCYPbB265W2ZjYhL5VQSyv1Xs3Skm0=
X-JAG-EncAlg: w3.org/2001/04/xmlenc#aes128-cbc
X-JAG-EncKeyAlg: w3.org/2000/09/xmldsig#rsa-sha1
Content-Type: application/json
Content-Length: 101

eIdV39/XV/IHgPNWB2Hpo2jWglsI9p<...>k5c4+vVs9d53o6OEoh7M0bybmtGwdZE=
```

Listing 5.3: Encrypted payload during a request

with the proof of identity. For this purpose, the combination of encryption and signature at the message level provides confidence that data is kept confidential from intruders, and that no modification have been made to it. The signature testifies authenticity of the encrypted content, and only the receiver can retrieve the original data. In the current version of our work, we do not address ordering between the two mechanisms, therefore it is not yet possible to encrypt a signature.

5.3.7 Multiparts

We consider the case where one request or response message contains several parts. It is the case for example when forms are submitted with several fields containing user data, or when several files are attached along the same request. In such case, we might have general-purpose information and sensitive-information. To encrypt sensitive information, we need a mechanism that specifies the format of the different parts. We have several choices: we can apply the security requirements on the entire request/response of the RESTful service, or just on some parts/elements. HTTP makes usage of the Multipurpose Internet Mail Extensions (MIME) standard[1] to separate the content in several parts. We can take advantage of this usage to distinguish parts of the data along requests. Therefore, if a request contains multiple parts, we can choose to sign and encrypt some of them without affecting the others.

The approach differs from what is implemented in WS-Security standards and S/MIME standard. In our approach, we are independent from the actual content-type, and proposes to gather in one place all security meta-data. WS-* standards deal with XML-based content, so they propose a fine-grained approach at the XML-data level. Our approach is more general, and provides resource-grained encryption and signature. The Listing 5.4 highlights this principle. It represents the signature for the first multipart element identified by <*root*>. In a multipart environment, the meta-information vary depending on the part subject to encryption or signature.

[1]http://www.ietf.org/rfc/rfc2045

```
PUT /sign/customer/111/file HTTP/1.1
Content-Type: multipart/form-data; boundary="uuid:7d156074-35"; start="<
    root>";
X-JAG-CertificateID: CN=CA CESSA, <...>O=SAP Labs France, C=FR;4102
X-JAG-DigestAlg: w3.org/2000/09/xmldsig#sha1
X-JAG-DigestValue: 0;8X3Ci4M+bhWKMg+f83CXoXXjjns=
X-JAG-SigAlg: w3.org/2000/09/xmldsig#rsa-sha1
X-JAG-SigValue: 0;lcj7v4UAMxFOkhBoX+8<...>NKo393OQ=
X-JAG-Multiparts: 0;<root>
Host: 127.0.0.1:8080
Transfer-Encoding: chunked

--uuid:7d156074-35
Content-Type: application/octet-stream
Content-Transfer-Encoding: binary
Content-ID: <root>
Content-Disposition: attachment;filename=data.dat
<..binary content..>

<.. HTTP Response ..>
```

Listing 5.4: Multipart signature example

The header *X-JAG-Multiparts* contains a set of multipart elements and some headers referenced by identifiers. These identifiers are used to reference digest and signature values in the other security headers.

5.4 Evaluation of REST security protocol

The REST security protocol is close to the WS-Security standard. WS-Security [OAS06] describes enhancements to SOAP messaging to provide protection through message integrity, confidentiality, and single message authentication. More precisely, it is an open format for signing and encrypting message parts leveraging XML Digital Signature and XML Encryption protocols, for supplying credentials in the form of security tokens, and for securely passing those tokens in a message. As explained in previous sections, the REST security protocol has been designed to be an equivalent alternative to WS-Security for RESTful services, with some differences in the way messages are secured and possible applications. In this section, we present few indicators to compare REST-Security, both in term of performance and configuration.

5.4.1 Environment & Methodology

In order to position the protocol performance with respect to the state of the art, we have run several performance tests to compare WS-* and RESTful based services. In order to have a clear methodology and to reproduce performance tests, the evaluation has been made on the same environment to eliminate network side-effects. We limited resource starvation on the server to obtain accurate data. The Table 5.2 lists server characteristics. In order to compare the different services, we evaluate them in a single framework proposing coverage of both JAX-RS and

Processor	Intel Core i7-2600 @ 3.40GHz
Installed RAM	16 GB
Hard Drive	Seagate ST3500413AS Barracuda 7200 500 GB
Application Server	Tomcat 7.0.21
Server JVM Memory	-Xmx 8000m
WS framework	CXF 2.4.2
Server certificate	RSA 1024
Client's certificates	RSA 4096

Table 5.2: Benchmark environment

JAX-WS specifications. The CXF service framework[2] allows us to compare the complexity of the two kinds of web services under the same conditions.

We have defined and implemented three scenarios, corresponding to real-use cases. In this way, we simulate several scenarios in order to evaluate and compare performance, message size, *etc.* The three scenarios correspond to:

Simple Get In the following, we identify this scenario with the acronym *Get*. The scenario retrieves information without further processing. It is materialized by the invocation of a method in WS-* to retrieve customer information, from customer identifier. In RESTful services, the client requests a customer through a *GET* action, and the service renders the customer in the requested format.

Modify Post In the following, we identify this scenario with the acronym *Post*. In this scenario, the data is transmitted in the request phase, and the response phase is just an indicator of the success or failure. Some additional processing is made on background to modify objects on the server. The modification of a remote resources is materialized by a method invocation with WS-* services, whereas it is a *POST* request in REST.

Large payload In the following, we identify this scenario with the acronym *Large* or *Big*. It corresponds to the transmission of large amount of data between client and server. The size of messages brings out the real impact of the protocol. Each operation gives rise to accurate observation of the cost in terms of size and performance. It is materialized by a method invocation for a customer document in the input for WS-* services, and by a *PUT* request in the RESTful version. The reply contains indication of success or failure.

The different scenarios provide heterogeneous tests to verify several properties of the REST security protocol, in different conditions. They cover the most problematic situation one can face in a real production environment. They are a good basis for protocol comparison. For each of the

[2]`http://cxf.apache.org/index.html`

scenarios, we have configured and run several tests with different security capabilities: signature, encryption, signature & encryption and no-security acting as the baseline. The experiments were performed couple of times to ensure consistent and valid results for comparison. The REST security implementation uses the same cryptographic algorithms as in the WS-Security configuration. For instance, both SOAP and REST services are set to use the "Basic128Rsa15" security algorithms suite: it determines the algorithms for digest, symmetric encryption, asymmetric encryption, as well as key derivation algorithms and key-wrap algorithms.

5.4.2 Size comparison

The Table 5.3 indicates the measurement in size to compare REST and WS-* services in the different scenarios. It lists the incoming and outgoing message sizes with distinction between headers and payload size. The results correspond to the different scenarios, with an equivalence between the *Get* and *Post* scenarios in terms of total size. The *Large* scenario sends a resource of around 3311kB. In the *Get* scenario, a client sends a request to the server in order to retrieve a *customer* object. In SOAP messages, the request is embedded in a SOAP envelope. The envelope grows with the type of security used. For each type, the SOAP headers comprise secure data to indicate the type of algorithm, the encrypted or signed parts, and sometimes full certificates. In REST messages, the request is directly represented by the HTTP verb used to query the server. Therefore, no additional payload is necessary than the actual data plus some meta-data headers.

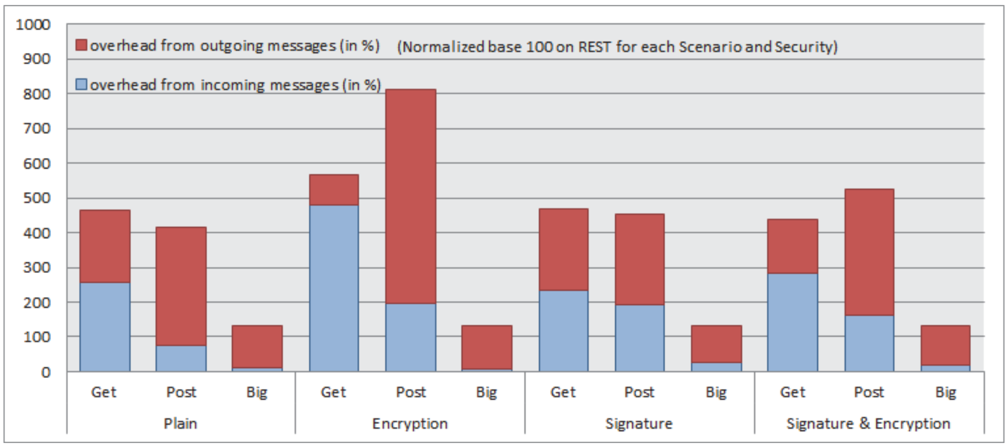


Figure 5.3: Overhead of SOAP messages compared to REST. For each scenario and security, the REST size represents the base 100

The Figure 5.3 highlights the global overhead using SOAP with any security mechanisms for the different scenarios. The REST size represents 100 for each scenario and security. We compare then the message overhead of different security mechanisms with its REST equivalent. For example, a SOAP-signed message size with the *Get* scenario represents around 460 when its counterpart in REST is 100. In the figure, we distinguish a second dimension: the origin of the overhead - from incoming message or outgoing message. The message increase for the previous scenario is half due to the incoming message, and second half by the outgoing message. In all tests, the usage of SOAP services instead of REST services is less efficient in terms of message size. The minimal overhead impact in all scenarios is 33%, which is the case where message payload is really large. We can explain it by the minimal impact of SOAP overhead compared to the actual data to transmit. This number is the result of our measurements, where the size of messages (including incoming and outgoing payload and headers) is larger when WS-* services are used compared to REST services, with all security mechanisms. The experimental cases where REST security protocol is the most efficient compared to WS-Security is on encryption of small set of data. The *Get* and *Post* scenarios present high SOAP overhead when data to transmit is small. For such cases, SOAP adds to much meta-data compared to the actual information, which multiply up to eight times the message size for a request and response in our measurements.

5.4.3 Processing performance comparison

In this paragraph, we present the processing performance comparison. The server has a certificate with RSA 1024 bits key, and the different clients have RSA 4096 bits. The difference of key size for the clients and the server impacts the time of processing depending on actions performed by the different actors. This behavior is directly linked to the performance of asymmetric algorithm that differs from encryption and decryption [Dai09]. For instance, the encryption algorithm is straightforward has it uses a small value for the exponentiation (typically $0x10001$). The decryption algorithm requires more computation as the exponent is of the size of the private key (1024 or 4096 bits in our benchmarks). Thus, the server can decrypt faster than clients at the cost of less security. The calculated factor shows server decryption is around 20 times faster than client decryption. In our benchmarks, it impacts the performance comparison between the different scenarios we have defined. For instance, the server processes messages from the *Get* scenario with one encryption (fast operation) when messages from the *Post* scenario needs to be decrypted (slow operation) which lowers the processing time and throughput.

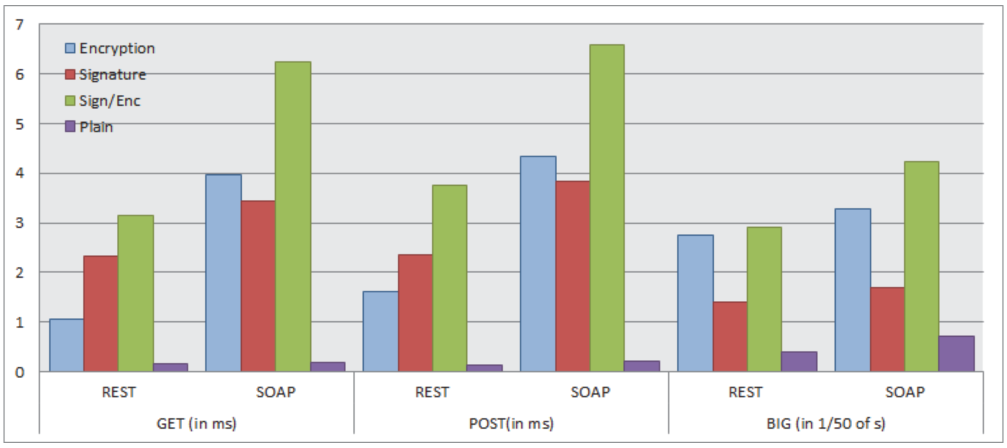


Figure 5.4: Average processing time comparison for the different scenarios

We have calculated the average processing time calculated under the same conditions. Each scenario has been launched for 60 seconds, with a single client emitting requests. The client sends messages sequentially to not overload the server and to extract the optimal processing time. The Figure 5.4 depicts the differences between the different scenarios. The difference between REST and SOAP average processing time differs depending on the algorithm scheme and scenario used. In the *Get* and *Post* scenarios, REST is twice more efficient than SOAP when cryptography is used. It can be explained by the ratio of data related to XML format and SOAP meta-information that impact size of messages. For thin SOAP messages, the ratio doubles the size compared to REST messages. The time spent to process message is directly impacted by this size. For large messages, the encryption scheme is shown to be slower than signature.

We can notice differences in term of performance with regards to encryption and signature, depending on the size of data to be processed. Although SOAP encryption is always more costly than SOAP signature, REST shows better performances with encryption when amount of data remains low like in the *Get* and *Post* scenarios. If the data size growths, signature is faster than encryption.

5.5 Related Work

In this section, we present some security models adopted by existing web services to expose their REST API's. Then, we provide alternative approaches to address REST security and performance issues.

The security model adopted by Amazon S3 [Ama06] supports authentication and custom data encryption over HTTP requests. The requests are issued with a token to prevent unauthorized users from accessing, modifying or deleting the data. The token conveys a signature value calculated per request which transmits a proof of identity, ensuring the authenticity of the request, similar to our protocol. The data encryption can be performed by the client itself, or by the server prior storage. The communication is supposed secured through SSL endpoints. Our approach brings more flexibility as actors decide of resources and headers to protect and transform. The server benefits of the PKI environment to render services to its clients without the need to generate and maintain a set of secret keys. The clients can also enable the REST security protocol with different service providers by simply uploading their public key.

The other models adopt a slightly different approach, making intensive usage of the OAuth 2.0 protocol. Yahoo [Yah] uses OAuth Authorization protocol (OAuth Core 1.0 [Ha10]) which is a simple, secure protocol to publish and share protected data when several actors require access to the resource. Yahoo demands the usage of an API Key to sign requests and provide end-user authentication. Twitter [Twi11] leverages the transport layer security by exposing REST APIs over SSL. Facebook [Fac12] requires the OAuth 2.0 protocol [Rea12] for authentication and authorization. They distribute SSL Certificates to consumers so that they can create signed requests and force users to use HTTPS. The Dropbox model [Dro12] allows third-party applications to use their services on behalf of users. Their model forces the requests through SSL and requires additional authenticity checks on messages. Like the previous approaches, they are combining transport layer security and application security. In our approach, we simplify the access of resources by unifying security at the message level. For instance, performing a request to retrieve a file with Dropbox transmits content metadata in an header. This content can be visible when the packet reaches the endpoint of a SSL tunnel, whereas our approach protects the header until its consumption.

The idea of having RESTful security as an equivalent of WS-Security has been expressed in a blog entry [Las10], using a similar approach but with no implementation and concrete specification. An approach to sign and encrypt multiparts have been drafted in [GMCF95]. They do not refer to REST services, but rather propose a model integrated to the multipart separation content to describe meta-information. Our approach benefits from multipart to split the payload in several resources, but we prefer centralizing security meta-data in headers to avoid service disruption, and to incorporate other field protection: headers, parameters, *etc.*. Our lightweight approach modifies content only when necessary.

Pautosso *et al.* [PZL08] describe the differences between REST services and “big” services with a number of architectural decisions about which type of service is more appropriate. We have used this work to compare security of both approaches and to provide an extension to REST services for more security. The work in [RS07] addresses attacks targeting SOAP-based services. Although attacks are based on the XML message format, we advocate that the approach presented can be easily introduced in our implementation using particular header fields to inform about the document structure.

Optimizing service consumption in terms of performance has been addressed for a long time. The problem is rather to balance usability and composability while allowing cross-cutting concerns such as security to protect the messages with a variable level of granularity. We can mention work on Fast Web Services [SPGK$^+$03] which defines binary-based messages to lower bandwidth and memory consumption. The price is the loss of self-description so that intermediaries cannot process the messages. In [STT05], Suzumura *et al.* propose a different approach, which is based on SOAP messages. They boost performance by considering partial regions of messages that differ from previously processed ones. Albeit the approach gives interesting results, they can not help with encrypted SOAP messages in the current state of the protocol.

5.6 Summary

We have presented a novel approach to provide security for RESTful services equivalent to WS-Security. Our solution respects the REST philosophy by minimizing the processing overhead to service consumers, without interfering in the service composition already in place. We are able to keep messages confidential and to sign them with a fine granularity. The custom and ad-hoc processing on a per-message basis is a valid alternative to the existing approaches, which consider mainly transport layer security for securing all REST services. The advantage of our approach is to hide the complexity for the consumers, with no pollution on request parameters, while still carrying security tokens processable and verifiable by recipients.

In addition, the REST security protocol allows us to build new secure collaborations between systems. We have presented two cases in which REST security protocol shows benefits: providing proofs for the transmission of other tokens that are part of a flow, and easy application of security properties in a flow of messages that pass through reference monitors to validate and enforce security policies. This is made possible as we propose to work on part of the messages, with custom security to specific business purposes. Also, the security properties are propagated with the resource as we provide an end-to-end application of security.

We also conducted a performance evaluation considering several use-cases to analyze the impact of message protection to the performance of the web services. The analysis comprises heterogeneous scenarios to compare different security mechanisms among them, but also the behavior of the application server when dealing with RESTful services versus SOAP-based web services. The results show that RESTful services are processed more efficiently from any point of view, which is inherent to the service's purpose. RESTful services are oriented to handle resources, whereas SOAP-based services forge requests for operation invocation. The protocol is self-descriptive, so all information about the message verifications and transformations are specified to let the recipient informed about the message state.

The modularization of security properties can be introduced on either the application layer or in between distributed systems, through gateways or proxies. Although we don't use aspects to introduce the security concerns, we rely on points external to an application to detect the security in place, and react in consequence. This work is a first step towards a larger platform that taint messages and propagate resource state across layers and across systems. We introduce it in the perspective part of the conclusion (Section 8).

Chapter 6

Modularization of privacy in cloud platform around persistance layer

In this contribution, we propose the modularization of a different security properties than we have seen in other chapters: privacy. The modularization is provided down to the platform level on a cloud application server. Cloud platform providers compete to propose the best solution for their customer. The attractivity of cloud platforms is growing as they propose on-demand and flexible solutions for many situations. They free their users from managing complexity of configuration, installation, and above all scaling of network traffic. In exchange of the contract with these platforms, the customers agree to delegate data control with the platform. Security of the cloud platform provider is a key differentiator in the choice of a suitable platform to host their applications. The role of the platform provider is then to release reliable and security capabilities to help their clients. Providing a modular and consistent process to properly ensure security for both platform and users is complex, and thus we bring our solution for a modularization of privacy concerns down to the platform. It means that the cloud platform provides new APIs and tools to gather privacy requirements of the application to enforce privacy during the runtime of the application.

Privacy in cloud computing is a major concern for individuals, governments, service and platform providers. In this context, the compliance with regards to policies and regulations about personal data protection is essential, but hard to achieve, as the implementation of privacy controls is subject to diverse kinds of errors. In this chapter we present how the enforcement of privacy policies can be facilitated by a Platform as a Service. Cloud applications developers can use non-obtrusive annotations in the code to indicate where personally identifiable information is being handled, leveraging the aspect-oriented programming (AOP) features. Subsequently the evaluation of user defined preferences is performed by trustful components provided by the platform, liberating developers from the burden of designing custom mechanisms for privacy enforcement in their software.

6.1 Introduction

In order to speed up the deployment of business applications, and to reduce overall IT capital expenditure, many cloud providers nowadays offer the Platform as a Service (PaaS) solutions as an alternative to leverage the advantages of cloud computing. We can mention for instance SAP NetWeaver Cloud, Google App Engine, or VMware Cloud Foundry, to cite a few. PaaS brings an additional level of abstraction to the cloud landscape, by emulating a virtual platform on top of the infrastructure, generally featuring a form of mediation to the underlying services akin to middleware in traditional communication stacks.

As the consequence of that shift, we observe that more and more personally identifiable information (PII) is being collected and stored in cloud-based systems. This is becoming an extremely sensitive issue for citizens, governments, and companies, both using and offering cloud platforms. The existing regulations, which already established several data protection principles, are being extended to assign new responsibilities to cloud providers with respect to private data handling.

The provision of privacy preserving services and tools will be one of the arguments favoring the choice of one PaaS provider over the other when a company is hesitating where to deploy new cloud application. The proposed reform of the European data protection regulation points out that privacy-aware applications must protect personal data by design and by default: “Article 22 takes account of the debate on a ’principle of accountability’ and describes in detail the obligation of responsibility of the controller to comply with this Regulation and to demonstrate this compliance, including by way of adoption of internal policies and mechanisms for ensuring such compliance. Article 23 sets out the obligations of the controller arising from the principles of data protection by design and by default. Article 24 on joint controllers clarifies the responsibilities of joint controllers as regards their internal relationship and towards the data subject[1].”

The correct enforcement of privacy and data usage control policies has been recently subject of several incidents reported about faulty data handling, perhaps on purpose, see for instance the cases of Facebook[2].

Therefore, addressing compliance requirements at the application level is a competitive advantage for cloud platform providers. In the specific cases where the cloud platform provider is also considered a joint controller, a privacy-aware architecture will address the accountability requirement for the PaaS provider with regards to the next generation of regulations. Such architecture can enable compliance also for the Software as a Service delivery model, if we assume the software was built over a privacy-aware platform. On the other hand, this could be hardly achieved in the context of Infrastructures as a Service, since there would be no interoperability layer on which the privacy controls can rely on.

In order to achieve this, the PaaS must implement some prominent, possibly standardized, privacy policy framework (such as EPAL[AHK$^+$03], P3P[Cra03]), where privacy preferences can be declared in a machine-readable form, and later enforced automatically. In such a setting,

[1]http://ec.europa.eu/justice/data-protection/document/review2012/com_2012_11_en.pdf

[2]http://mashable.com/2011/10/21/facebook-deleted-data-fine/

the privacy enforcement controls could be easily incorporated into new deployment landscape accelerating the development process of compliant applications. Furthermore the cloud platform can offer the guaranties ensuring the correct implementation of the enforcement components. This could be offered either via a certification mechanism or an audit of an existing cloud landscape that would be executed by the governing entities.

In this contribution we present work towards the implementation of privacy-aware services in a PaaS. We aim to empower the cloud platform with capabilities to automatically enforce the privacy policy that is result of the end-user consent over the application provider privacy policy. End-user policies and service provider terms of use are defined in a state of the art privacy and usage control language [BNP11]. In order to leverage the provided implementation of privacy-aware services, cloud application developers need to introduce simple annotations to the code, prior to its deployment in the cloud. These indicate where PII is being handled, towards automating privacy enforcement and enabling compliance by design and by default. The idea is outlined in Figure 6.1, and consists of design-time steps (declaring policies, annotation of the code and deployment in the cloud); and run-time steps (including policy matching, privacy control and obligation execution).

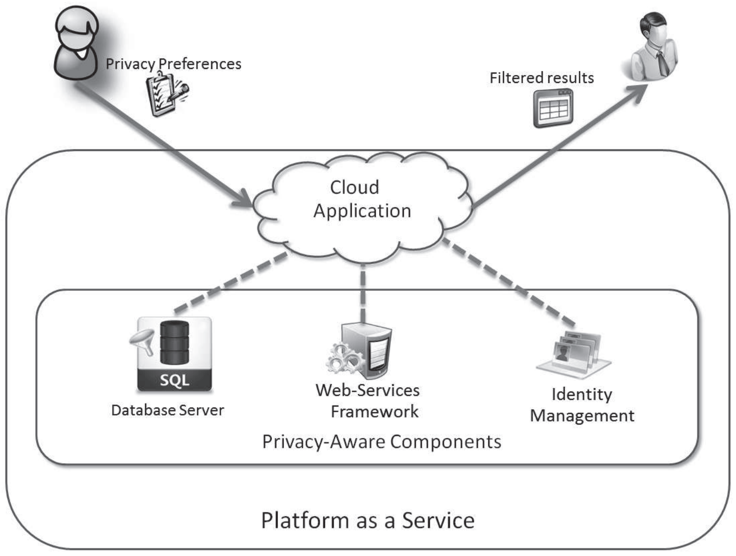


Figure 6.1: Privacy aware PaaS components

The enforcement mechanisms are provided by the platform with the help of a new approach for aspect-oriented programming where aspects can be manipulated at the process and at the platform levels [ISR$^+$11]. That approach gives a possibility to maintain a more flexible configuration of the enforcement mechanisms. The mechanisms interpret end-user preferences regarding handling of the PII, presented in form of opt-in or opt-out choices among available privacy policies of a cloud application, and later perform the required actions (filtering, blocking, deletion, etc). We experimented on a Java-based Platform as a Service, SAP NetWeaver Cloud, to demonstrate how privacy preferences can be handled automatically thanks to the use of simple Java annotation library provided in our prototype. The platform provider can make in this way an important step towards providing built-in compliance with the personal data protection regulations transparently, as we describe in the next sections.

The remainder of the contribution is organized as follows: in Section 6.2 we present our use case and we give a brief overview of the privacy policy language we adopt in this work, Section 6.4 brings a discussion on related works, in Section 6.3 we introduce the technical architecture allowing to enforce privacy on multiple PaaS layers and Section 7 presents future perspectives along with summary of the contribution.

6.2 Privacy-Aware Applications in the Cloud

In this section we present our use case involving multiple stakeholders accessing users' PII in the cloud, as well as some background on privacy policy language that we used.

6.2.1 Use case

In our use case we consider a loyalty program offered by a supermarket chain, accessible via a mobile shopping application that communicates with back-end application deployed on the PaaS cloud offering. The supermarket's goal is to collect the information about consumers' shopping behavior that results in the creation of a consumer profile. This profile could then be used to provide consumers more precise offers and bargains. Supermarket's business partners may also want to access this information in order to propose personalized offers to the mobile shopping application users themselves.

The back-end application for the supermarket loyalty program is developed using Java programming language and uses the cloud persistency service to store application data. The interface to access the persistency service is based on Java Persistence API (JPA)[3], which is nowadays one of the most common ways of accessing a relational database from Java code.

The supermarket employees can access detailed results of database queries regarding the consumers' shopping history and also create personalized offers, via a web-based portal. Moreover, the cloud application exposes web services through which third parties interact with the back-end system to consume collected data: both for their own business analysis, but also to contact directly the consumers for marketing purposes.

The interface for the consumers makes it possible to indicate privacy preferences with respect to the category of products (health care, food, drinks, etc) that one wants to share his shopping habits about. The consumer can also indicate whether he permits the supermarket to share personally identifiable information with its business partners, among other usages. This choices are then reflected by the private data access control mechanism that we will describe in Section 6.3.

[3]http://docs.oracle.com/javaee/5/tutorial/doc/bnbpz.html

6.2.2 Background: Privacy Policy Language

The users of the mobile shopping application are asked to provide various kinds of personal information, starting from basic contact information (addresses, phone, email) to more complex data such as shopping history or lifestyle preferences. Service providers describe how users' data are handled using a privacy policy, which is explicitly presented to users during the data collection phase.

We adopt the PrimeLife[4] Policy Language (PPL) [BNP11], which extends XACML with privacy-related constraints for access and data usage. PPL policy is then used by the application to record its privacy policy. It states how the collected data will be used, by whom, and how it could be shared. On the other hand, the end-user also selects among the possible choices as to the conditions of the data usages, that are derived from privacy policies specific to the application. This user opt-in/opt-out choice is managed by the application and as such is not part of the generic enforcement mechanism developed by us. Before disclosing personal information, the user can match his preferences against the privacy policy of the service provider with the help of a policy matching engine. The result of the matching process is an agreed policy, which is then translated into the set of simple rules that are stored together with users' data inside the cloud platform's database servers.

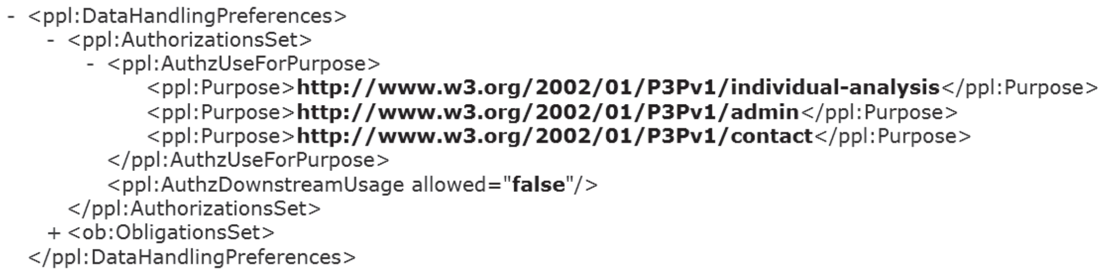
In summary a PPL policy defines the following structures [BNP11]:

- Access Control Elements: inherited from the XACML attribute-based access control mechanism to describe a shared resource (in our case PII) in general, as well as entities (subjects) that can obtain access to the data.
- Data Handling Preferences: expressing the purpose of data usage (for instance marketing, research, payment, delivery, etc.) but also downstream usage (understood here as sharing data with third parties, e.g. advertising companies), supporting a multi-level nested policy describing the data handling conditions that are applicable for any third party retrieving the data from a given service.
- Obligations: specify the actions that should be carried out with respect to the collected data, e.g. notification to the user whenever his data is shared with a third party, or deletion of the credit card number after the payment transaction is finished, etc. Obligations in PPL can be executed at any moment throughout whole lifetime of the collected data and can affect future data sharing transactions, e.g. with third parties.

An excerpt of a policy is shown in Figure 6.2. It shows part of a policy rule, stating the consent to use the data collected for three distinct purposes (described using P3P purpose ontology), but forbids downstream usage.

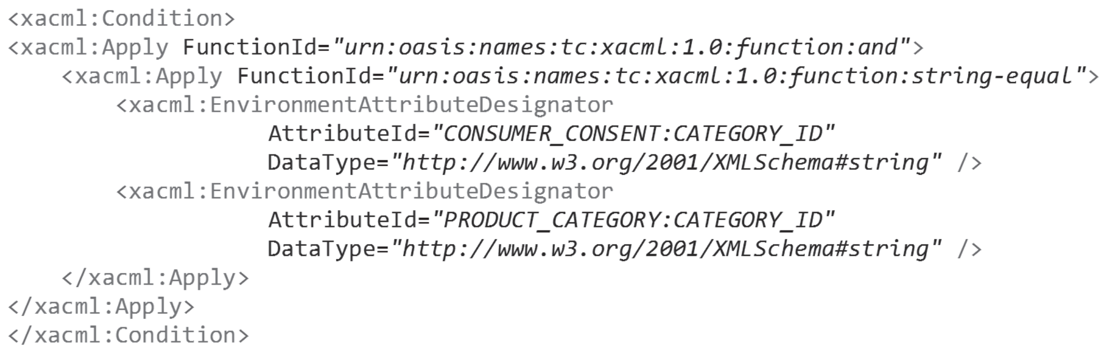
Consumer opt-in/opt-out choice is linked with PPL policy rule via XACML conditions that we adopted for this purpose. We have reused *EnvironmentAttributeDesignator* elements syntax to refer to the actual recorded consumer choice in the application data model, as shown in Figure 6.3. The location is provided as the *AttributeId* value and can be read as `TABLE_NAME:COLUMN_NAME` of the database table where this choice is stored (`CONSUMER_CONSENT`) as well as a foreign

[4]`www.primelife.eu`



```
- <ppl:DataHandlingPreferences>
   - <ppl:AuthorizationsSet>
      - <ppl:AuthzUseForPurpose>
            <ppl:Purpose>http://www.w3.org/2002/01/P3Pv1/individual-analysis</ppl:Purpose>
            <ppl:Purpose>http://www.w3.org/2002/01/P3Pv1/admin</ppl:Purpose>
            <ppl:Purpose>http://www.w3.org/2002/01/P3Pv1/contact</ppl:Purpose>
         </ppl:AuthzUseForPurpose>
         <ppl:AuthzDownstreamUsage allowed="false"/>
      </ppl:AuthorizationsSet>
   + <ob:ObligationsSet>
  </ppl:DataHandlingPreferences>
```

Figure 6.2: Excerpt of a PPL policy rule



```
<xacml:Condition>
<xacml:Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:and">
    <xacml:Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <xacml:EnvironmentAttributeDesignator
                    AttributeId="CONSUMER_CONSENT:CATEGORY_ID"
                    DataType="http://www.w3.org/2001/XMLSchema#string" />
        <xacml:EnvironmentAttributeDesignator
                    AttributeId="PRODUCT_CATEGORY:CATEGORY_ID"
                    DataType="http://www.w3.org/2001/XMLSchema#string" />
    </xacml:Apply>
</xacml:Apply>
</xacml:Condition>
```

Figure 6.3: Excerpt of a PPL policy condition

key to the product category table (`CATEGORY_ID`) that is used to join products table. This information is used when enforcement mechanism is put in place to take consumer consent into account whenever information about consumer's shopping history (for certain product categories) is requested. This policy definition of how user consent is linked to the rest of the application data model is left in charge to the application developer as he is the one possessing full knowledge of the application domain.

6.3 Privacy Enhanced Application Programming

We have designed a framework able to modify, at the deployment time, the architectural elements (such as databases, web service frameworks, identity management, access control, etc) enriching it with the further components in order to enforce user privacy preferences. In this landscape the new applications deployed on the modified platform can benefit from privacy-aware data handling.

6.3.1 Programming Model

The privacy-aware components are integrated seamlessly with cloud application at the deployment time, so that the enforcement of privacy constraints is done afterwards automatically. They mediate access to the data sources, enforcing privacy constraints. In this case we are taking full

```
@Entity
@PII
public class ShoppingHistory implements Serializable
{ ... ... }
```

Figure 6.4: JPA entity class annotation indicating persistency of private information

benefit of the uniform database access in the PaaS landscape that is exposed via standard Java database interfaces such as JDBC (Java Database Connectivity) or JPA.

Usually the application code handling privacy related data is scattered and tangled over the application, being difficult to handle and to maintain if any changes in the privacy policy are introduced. As we observed in the existing applications the operations, which are performed on the private user data to ensure that privacy policies are enforced, are typically cross-cutting concerns in aspect-oriented programming paradigm. Inspired by this, we designed a process for the application developer that contributes to simplifying a way the data protection compliance could be achieved. It consists of adding meta-information to the application code via Java annotation mechanism in the JPA entity classes. Entity class in JPA terms is the one that is mapped into a database structure (usually a table, but also more complex type of mappings exist, e.g. to map object inheritance hierarchy) and enables the objects of that class to be persisted in a database. We provide also a second type of annotations, for the methods that make use of a private data, to indicate the purpose of the data usage.

The modifications to the code are non-intrusive, in the sense that the application business functions flow will stay exactly the same as before, except for the data set it will operate on, that will be obtained from database by adhering to the privacy policy. The changes are as transparent as possible from the application point-of-view as new platform components propose the same set of API as in the traditional platforms (in our case this API is JPA) and additional functionality is obtained via non-obtrusive code annotations that in principle could be easily removed in case described features are not required or not available.

This approach adds value with respect to legacy applications while allowing privacy management when needed. Another advantage is that the cloud service provider can easily move to another cloud platform without being locked into the certain vendor, apart from the fact that the guarantees given by the platform about private data handling could not be the same.

The platform we used to develop our prototype offers the enterprise level technologies available for Java in terms of web services and data persistency (JEE, JPA). In most of the examples we present along the use case we assume that the application developer will likely use a framework such as the JPA to abstract the database access layer.

In our approach developers are required to add annotations to certain constructs, such as *@PII* annotation in the JPA entity class (Figure 6.4). This annotation indicates that the class comprises one or more fields having private data (that usually are represented in database as columns) or that all fields are to be considered as PII (thus whole database table row needs to be either filtered or kept during the privacy enforcement, as JPA entity is by default mapped to a database row).

In the business code that is handling the private data we propose to use two other annotations to indicate class and method that processes PII sets. An example of annotated code is shown in

```
 5  @PiiAccessClass
 6  public final class ShoppingHistoryDAO extends DaoImpl<ShoppingHistory>
 7  {
 8     ...
 9     @Info(purpose= "http://www.w3.org/2006/01/P3Pv11/marketing" )
10     public final List<ShopHistory> getHistories()
11     { ... }
12     ...
13  }
```

Figure 6.5: Annotating private data usage class with PII meta-information

Figure 6.5. In this figure the method annotation holds the information that the shopping history list items will be processed for marketing purpose.

In summary our library provides three different annotations:

***@PII*:** It is a flag to indicate personally identifiable information inside a JPA entity class definition. Such information is usually stored in a database as a table or a column. In Figure 6.4 this annotation involves the scope of the class declaration, see lines 2 and 3.

***@PiiAccessClass*:** This annotation should be put in the class to indicate where it contains access methods to personal data (see line 5 in Figure 6.5). We assume that PII access method performs queries to the database that are requesting private user data.

***@Info*:** This annotation is applied to PII access method, to describe the purpose or set of purposes of the query performed in that method (see lines 9 and 10 in Figure 6.5).

We expect the application developers to use this annotations to mark each usage of personal data as well as to indicate correct purposes. Ultimately they seek compliance to regulations, therefore we trust them to correctly indicate via the annotations the intended usage of the data. One can envisage that automated code scanners and manual reviews can take place during an audit procedure in order to check whether the annotations are rightfully used.

6.3.2 Implementation

In this section we detail the components of our prototype architecture. Technically our code components are packaged as several OSGi (Open Services Gateway initiative framework[5]) bundles. A bundle is a component which interacts with the applications running in the cloud platform. Some of them are to be directly deployed inside the PaaS cloud landscape and managed by the cloud provider while the other are part of the library to be used by the cloud application developers. Cloud providers can easily install or uninstall bundles using the OSGi framework without causing side effects to applications themselves (e.g. no application restart is required if some of the bundles are stopped). In the context of our scenario, we have three main bundles managed by the cloud provider (JDBC Wrapper, Annotation Detector and SQL Filter) and one additional bundle (Policy Handler) that is providing a translation from an application privacy policy file written in the PPL language into an internal representation stored in the Constraints

[5] `http://www.osgi.org`

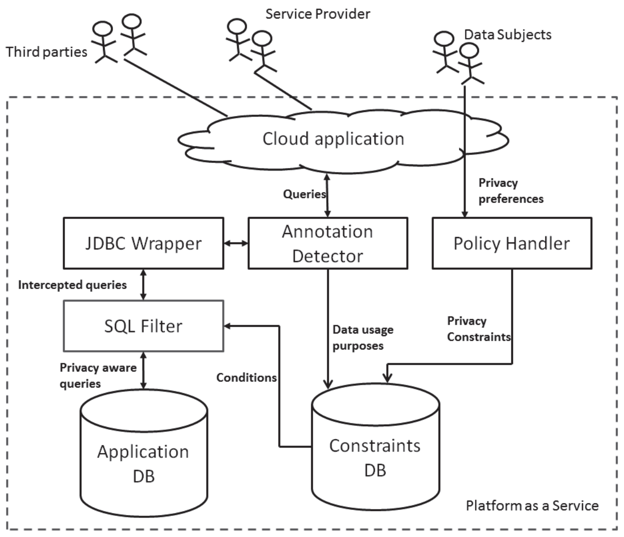


Figure 6.6: Enforcement components

Database. The diagram in Figure 6.6 presents the architecture of the system, which we are going to describe in more details in the following subsections.

JDBC Wrapper

The Wrapper intercepts all queries issued by the cloud application directly or by the third parties which want to consume the collected data containing shopping history of the fidelity program participants. This component is provided on the platform as an alternative to the default JDBC driver in order to enforce consumers' privacy preferences. Actually the wrapper makes use of the default driver to eventually send the modified SQL calls to database.

JDBC Wrapper bundle implements the usual interfaces for the JDBC driver and overrides specific methods important to the Java Persistence API, necessary to track the itinerary of SQL queries. As a matter of fact, it is wrapping all JDBC methods that are querying the database, intercepting SQL statements and enriching them with proper conditions that adhere to privacy policy (e.g. by stating in the `WHERE` clause conditions that refer to the consumer consent table). In order to identify the purpose of each query, its recipient and the tables referred, we retrieve the call stack within the current thread thanks to the annotations described in the previous section. We look for the PII access class, then we look for the method that sends the request to get the further parameters that help properly enforce privacy control.

Annotation Detector

First task of this component is to scan Java classes at the deployment time and look for the JPA entities that are containing privacy-related annotation in its definition (`@PII`). List of such classes is then stored inside the server session context. Information about entities considered as PII is used to determine which database calls need to be modified in order to help preserve consumer privacy preferences.

In the second run the annotation detector scans the application bytecode in order to gather information concerning the operation that the application intends to perform on the data, annotated with `@PiiAccessClass` and `@Info` annotation. It is important to recall that the annotations are not a "programmatic" approach to indicate purpose, as they are independent from the code, which can evolve on its own. The assumption is that developers want to reach compliance, thus the purpose is correctly indicated, in contrast to [BBL05], where it is assumed that end-users themselves indicate the purposes of the queries they perform. The cloud platform provider can instrument the annotation detector with a configuration file where the required annotations are declared. The detector can recognize custom annotations and stores information about related entity class in the runtime for future use.

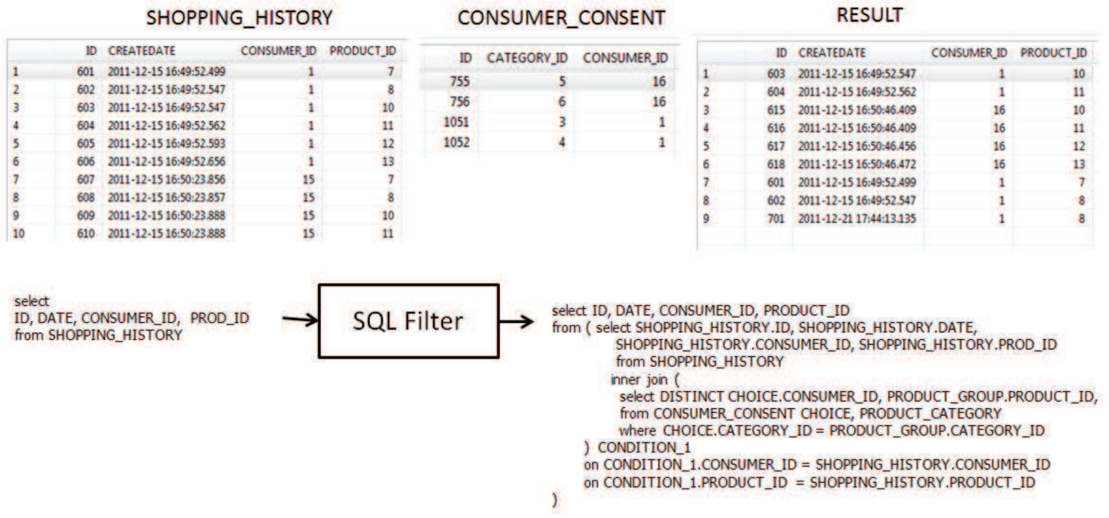
SQL Filter

This component allows us to rewrite original queries issued to the database by replacing the requested data set with a projection of that data set that takes into account consumers' privacy choices. SQL Filter modifies only the `FROM` part of a query, implementing an adapted version of the algorithm for disclosure control described in [LAE^{+}04], also similar to the approaches described in [AKSX03], [MT06], and [RMSR04].

The query transformation process makes the use of the pre-processed decisions generated by the Policy Handler that concerns each possible combination of the triple purpose, recipient and PII table.

The transformation of the SQL query happens at the runtime. Consumer's privacy preferences are enforced thanks to the additional join conditions in the query, relating data subject consent, product category and filtering rules. The output is a transformed SQL query that takes into account all stated privacy constraints and is still compatible with the originally issued SQL query (it means that the result set contains exactly the same type of data, e.g. number of columns and their types). From a business use-case perspective, it was always possible to visualize relevant data, e.g. shopping history information, etc, without disclosing personal data when user didn't give his consent. The process is illustrated in Figure 6.7. It depicts the process of query modification when application is accessing data from the **SHOPPING_HISTORY** table (top-left corner of this figure). Original query (bottom-left) is transformed so that it takes into account the information derived from privacy policy that was put by the Policy Handler in the **CONSUMER_CONSENT** table (top-center). This table stores the association between the consumers and the different product categories with which these consumers opt to reveal their shopping history. Modified query (bottom-right) yields the data set of the same structure as original query but without disclosing the information that consumers declined to share, as it can be seen in the **RESULT** table (top-right).

The negotiated privacy policies are stored under the form of constraints together with the data in the database component provided by the cloud infrastructure. Whenever a query is launched by the application, we use the information collected by the annotation detector in order to modify queries on the fly, thus using the constraints to filter out the data that is not allowed to appear in the query results.


SHOPPING_HISTORY

	ID	CREATEDATE	CONSUMER_ID	PRODUCT_ID
1	601	2011-12-15 16:49:52.499	1	7
2	602	2011-12-15 16:49:52.547	1	8
3	603	2011-12-15 16:49:52.547	1	10
4	604	2011-12-15 16:49:52.562	1	11
5	605	2011-12-15 16:49:52.593	1	12
6	606	2011-12-15 16:49:52.656	1	13
7	607	2011-12-15 16:50:23.856	15	7
8	608	2011-12-15 16:50:23.857	15	8
9	609	2011-12-15 16:50:23.888	15	10
10	610	2011-12-15 16:50:23.888	15	11

CONSUMER_CONSENT

ID	CATEGORY_ID	CONSUMER_ID
755	5	16
756	6	16
1051	3	1
1052	4	1

RESULT

	ID	CREATEDATE	CONSUMER_ID	PRODUCT_ID
1	603	2011-12-15 16:49:52.547	1	10
2	604	2011-12-15 16:49:52.562	1	11
3	615	2011-12-15 16:50:46.409	16	10
4	616	2011-12-15 16:50:46.409	16	11
5	617	2011-12-15 16:50:46.456	16	12
6	618	2011-12-15 16:50:46.472	16	13
7	601	2011-12-15 16:49:52.499	1	7
8	602	2011-12-15 16:49:52.547	1	8
9	701	2011-12-21 17:44:13.135	1	8

```
select
ID, DATE, CONSUMER_ID,  PROD_ID
from SHOPPING_HISTORY
```

```
select ID, DATE, CONSUMER_ID, PRODUCT_ID
from ( select SHOPPING_HISTORY.ID, SHOPPING_HISTORY.DATE,
           SHOPPING_HISTORY.CONSUMER_ID, SHOPPING_HISTORY.PROD_ID
           from SHOPPING_HISTORY
          inner join (
           select DISTINCT CHOICE.CONSUMER_ID, PRODUCT_GROUP.PRODUCT_ID,
           from CONSUMER_CONSENT CHOICE, PRODUCT_CATEGORY
           where CHOICE.CATEGORY_ID = PRODUCT_GROUP.CATEGORY_ID
      ) CONDITION_1
      on CONDITION_1.CONSUMER_ID = SHOPPING_HISTORY.CONSUMER_ID
      on CONDITION_1.PRODUCT_ID  = SHOPPING_HISTORY.PRODUCT_ID
)
```

Figure 6.7: SQL transformation example

This approach is interesting because the behavior of the application itself is not modified. The impact on the performance of the system is minor, as the policy enforcement is actually pushed into a database query and also the complexity of this query transformation algorithm is low, as shown in previous works [LAE$^+$04]. The work in [AKSX03] brings some performance evaluation for the same kind of transformations. We advocate that the ability to implement privacy controls is more important than these performance questions when dealing with private data in cloud computing.

6.4 Related Work

There are many similarities between our approach and the work described in [MT06]. It proposes a holistic approach for systematic privacy enforcement for enterprises. First, we also build on the state of the art access control languages for privacy, but here with an up-to-date approach, adapted for the cloud. Second, we leverage on the latest frameworks for web application and service development to provide automated privacy enforcement relying on their underlying identity management solutions. We also have similarities on the way privacy is enforced, controlling access at the database level, which is also done in [AKSX03].

Although the query transformation algorithm is not the main focus of our work, the previous art on the topic [CNS99, RMSR04, BBL05] present advanced approaches for privacy preserving database query over which we can build the next versions of our algorithm. Here we implemented an efficient approach for practical access control purposes, but we envisage to enrich the approach with anonymization in the future.

On the other hand, we work in the context of the cloud, where a provider hosts applications developed by other parties, which can in their turn communicate with services hosted in other domains. This imposes constraints outside of the control of a single service provider. We went further in the automation, by providing a reliable framework to the application developer in order to transfer the complexity of dealing with privacy preferences to the platform provider. Our annotation mechanism provides ease of adoption without creating dependencies with respect to the deployment platform. More precisely, no lock in is introduced by our solution. However, changes in the database schema that involves PII data require an application to be redeployed in the platform in order to process the eventually new annotations. An alternative is to approach [TS12] to provide sticky policies in the cloud.

The work in [Lan02] presents an approach based on privacy proxies to handle privacy relevant interactions between data subjects and data collectors. Proxies are implemented as SOAP based services, centralizing all PII. The solution is interesting, but it is not clear how to adapt the proxy to specific data models corresponding to particular applications in a straightforward way.

Our work is aligned with the principles defended in [PC09], in particular we facilitate many of the tasks the service designers must take into consideration when creating new cloud-based applications. In [MP09], a user-centric approach is taken to manage private data in the cloud. Control is split between client and server, which requires cooperation by the server, otherwise obfuscated data must be used by default. This is a different point of view from our work, where we embed the complexity of the privacy enforcement in the platform itself.

Automated security policy management for cloud platforms is discussed in [Lan10]. Using a model driven approach, cloud applications would subscribe to a policy configuration service able to enforce policies at run-time, enabling compliance. The approach is sound but lacks of fine-grained management for privacy policies, as it is not clear how to deal with personal data collection and usage control.

In this work, we use privacy policy based on the data owner's preferences, to then monitor use of sensitive data and filter out the results. We have decided to let the application developer in charge of linking user consent to the rest of the application data, so that no specific administration model for the privacy policies is defined. In reality, the application hosted by the cloud platform needs to address a combination of security policies coming from the cloud platform, the application's policy and the user consents. For such approach, we can benefit from Ajam et al. [ACBC10] which propose three administration enforcement approaches of the privacy policies. From their work, we would have to define several types of context and clarify the different policies from the different entities.

In [IKC09], cryptographic co-processors are employed to ensure confidentiality of private data protection. The solution is able to enforce rather low level policies using cryptography as an essential mechanism, without explicit support to design new privacy compliant applications. Several works exist on privacy protection in Web 2.0 and peer-to-peer environments, such as in [TSGW09], where an access control mechanism is adopted for social networks. Some of these ideas can be reused in the context of cloud applications, but our approach differentiates from this line of work in the sense we empower the cloud applications developers with ready to use mechanisms provided directly by the cloud platform.

In [CW07], aspect-oriented programming is used as well to enforce privacy mechanisms when performing access control in applications. The work adopts a similar approach to ours, but privacy mechanisms are created in a per-application basis. In our approach, by targeting the platform as a service directly, we are able to facilitate enforcement in multiple applications.

In [Oul13], the author presents an approach that enforces the security and privacy requirements based on pre-processing of the queries to the data-sources. Although the technologies are not the same, we used the same approach to rewrite queries. The author go into the problem in depth, as they propose a model to enforce privacy and security policies without changing the implementation of their services. They adapt the response of the services with a fine grained control of information disclosed by services: depending on the call issuer and the purpose of the invocation, they adapt the response.

6.5 Summary

We presented a solution to simplify the process of enabling personal data protection in Java web applications deployed on Platform as a Service solution. We augment cloud applications with meta-data annotations and private data-handling policies which are enforced by the platform almost transparently from the developer perspective (the major overhead is only in placing the right annotations in the code). We differentiate ourselves from other approaches by proposing tools at the application layer. We intercept set of queries and base our adaption from the resource description. The description is directly inlined in the application.

The cloud consumer applications indicate how and where personally identifiable information is being handled. We adapt the platform components with privacy enforcement mechanisms able to correctly handle the data consumption, in accordance with an agreed privacy policy between the data subject and the cloud consumer.

The advantages of our approach can be summarized as follows: the implementation details of the privacy controls are hidden to the cloud application developer; compatibility with legacy applications, since the annotations do not interfere with the existing code; cloud applications can gracefully move to other platform providers that implement privacy-aware platforms in different ways. Sensible changes in the database schema, specifically those modifying PII, require the application to be redeployed in the cloud, possibly with new annotations.

Some future directions include the orchestration of other components such as event monitors, service buses, trusted platform modules, etc, in order to provide real-time information to users about the operations performed on their personal data. We would also be able to provide k-anonymization We plan to generalize our approach to enforce other kinds of policies, such as service level agreements, separation of duty, etc.

An important improvement of this work is the integration of advanced k-anonymization [Swe02] process at the database access level. Such solution would be more adapted to business applications than access control, since the end-users could obtain more meaningful information, without fully disclosing the identities of the data subjects.

Part III

Conclusion and perspectives

Chapter 7

Conclusion

In the course of the thesis, I have developed several contributions related to the secure development of applications in a broad range. Initially, it was targetting service-oriented applications, but we have observed that the techniques and concepts were applying to the discipline of software engineering. The proper encapsulation of behavior, and application of cross-cutting concerns in applications, components, platforms, as well as every piece of computer system is the result of decades of research. The field is becoming more mature over the time, by differentiating business code, technical code, and adapting behavior of the program to several environments. The thesis falls in an industrial context, where operational safety is central. It means that developers have several constraints in their application, being both the implementation of business-specific concerns, *i.e.,* how to make the ERP creating business value to its customers, but also transverse concerns. We leverage existing technologies related to software engineering and cross-cutting modularization to adapt them towards secure softwares. Security is a complex field, whom we concentrate on automation of secure programming best practices, but also modularization of business security concerns.

The contributions can be separated in two categories: contributions that address secure programming, and contributions that introduce security properties into application's landscape. We first assist developers in writing secure software with the minimal effort. We provide the tools directly integrated into developers' landscape to minimize the human factor and avoid the common pitfalls. The contributions are a direct application of secure programming best practices in which we provide additional checks and boundaries to offer additional services. In the second type of contributions, we propose to introduce security properties at a different level of abstraction. Instead of developers' assistance, we provide methods at the platform level. The problematic differs from a single programming language, and one needs to combine several tools to gather security requirements, enforce them in the application, and verify their correct application during execution. We have developed a security policy language agnostic of concrete technology, and highlighted the usage with a newly REST-services framework. It means that the security policies where capable of orchestrating the components and augment their security without the need to know the concrete service technology used. The specialization is deferred to the last moment to provide greater flexibility in application of security. In another contribution, we have also demonstrated the need of platform-wide components to the direct application of privacy. Such components are desirable to offer a consistent approach to properties that are generally custom made, therefore error prone.

The different contributions show the variety of approaches in modularizing security properties. From these works, we have observed common pitfalls for which we propose further directions. First, security properties were discussed as they are one single domain which crosscut application, and in which methodologies applies without distinction. In reality, we differentiate two kind of security, that we express as *defensive security* and *constructive security*. The two kinds of security are completely different. The two generally crosscut the application, but methodologies to handle their enforcement are different. Second, aspect-oriented approach have been proposed as one solution to cover all security cases. In fact, there are shade of differences depending on the scope of the security properties. Aspect-oriented programming is best when one can easily detect the weaving points . The applications we were analyzing does have points such as inputs that are fixed and easy to detect. But when it comes to analysis in the dataflow, detection becomes harder with several decisions that can be taken at runtime only. In such case, we need to provide custom solutions, and generally think at a meta level to respect program semantic and provide the best solution to the specific problem. Third, tackling security properties at the program-level only might lead to inconsistencies. Programs are inter-connected and evolve in complex landscape with different configuration, policies, environments. The encapsulation of security behavior needs to be adapted to the distribute nature of programs.

In order to overcome these pitfalls, we describe in the following a perspective towards cross-layer and parametric aspect system.

Chapter 8

Perspectives: towards a cross-layer and parametric aspect system

Aspect languages have evolved over the years to ease the integration of aspects in application. New pointcut primitives have been defined to allow aspect developers to describe complex use cases. The architecture of aspects has also evolved to follow the new java language specifications, such as annotations. Yet, the aspect languages are confined to their local execution environment. The problem is that cross-cutting concerns span different administrative layers, but also different technical layers. It is difficult to propose at the same time a solution that allows flexible interaction between actors and that ensure definition and enforcement of security properties across layers. We present in the following requirements for a solution that what we would like to achieve: consistent and systematic application of cross-cutting security properties across layers. For this purpose, we advocate new constructs in aspect languages to provide new extension to existing pointcut languages and new context exposition for advices. It also means that we have to provide tools for the aspect system. In a nuthsell, we would like to attach more information to resources when going in and out the application. It provides means to aspect system to work on resources into the execution environment of the platform, but also to share the state of a resource with other applications. In the following, we provide a first section to present the motivation that conducted to this work. Then, we give requirements for a language that we briefly present. Then, we express how such a language would have assisted us in the course of the previous contributions.

8.1 Motivation

Aspect systems are powerful to model cross-cutting concerns and apply them on an application. Belblidia [Bel08] has presented in her thesis several AspectJ extensions that have been suggested by the community and by her lab to express security issues that cannot be handled with the AspectJ language. We already introduced some in the previous sections, like the *dataflow pointcut*, which allows the detection of an input value in the application that is propagated to a sink. Masuhara introduced such approach to protect against XSS, although there is no current implementation and that the pointcuts need to explicitly declare propagation of data flow. The other extensions are oriented towards more expressivity from an aspect developer point of view: a *predicted control flow pointcut*, a *loop pointcut*, inline wildcard in pattern matching, local variables tracking (and not only parameters of methods and return types), *etc.*

All of these extensions are there to enhance existing pointcut language with new primitives in pointcut. In the contributions we have presented in the previous chapters, we also suffer from aspect expressiveness limitations. In the following, we go through the different contributions of the thesis and highlight the limitations we experienced with the state of the art aspect languages. Finally, we present snippets of code to present our vision to handle such problems.

8.1.1 Towards static analysis replacement

In Chapter 3, we presented a solution that use a static analysis phase to gather points in the application where vulnerabilities might occur.

The tools and methods that we provide in our solution allows developers to have an overview on the application they develop, and quickly gather vulnerabilities. We have presented guided steps towards mitigation of the vulnerabilities, depending on the detected categories and by analyzing the paths between vulnerability sources (entry points) and sinks (exit points).

Using Aspect-Oriented programming for automation of validation library is a natural choice as we introduce cross-cutting concerns in applications: security transformations in the application are scattered and tangled in the application, which makes them difficult to address one by one. The solution we have developed generates aspects, and builds the list of vulnerability protection by adding new joinpoints to a particular aspect. There are several pointcuts: one for each of the couple $(category, type)$, in which category represents a malformed input vulnerability, or cross-site scripting, *etc.*, and in which the type represents the correction module type to apply in order to correct the problem. Examples of such correction modules are $encodeForJavascript$. Our vulnerability detection approach relies on a built-in detection algorithm to pass directly the static analysis results to the correction part (validation). The two parts of our approach are dedicated components, from which the detection component extracts sensitive points in a program, and elicit the points where security transformation can be injected. The validation component uses the output of the previous component to forge a specific pointcut.

We have adopted this approach because of the lack of straightforward information flow and data flow interception with aspects. The mainstream pointcut language of AspectJ allows a partial detection of information flow that we abused by recomposing the hierarchy of calls. In these situations, our problem is the correlation between the pointcut language and the program syntax. We rely on a specific component to statically analyze the program and elicit sensitive points, whereas we would like to write a single pointcut that retrieve the sensitive points for us. The problem is that such pointcut is not yet addressed. The closest approach with aspects, the *dflow pointcut*, requires us to indicate entry points and sinks in advance, which we are not aware of.

We present hereafter two snippets that present how we would use such constructs to propose an inline analysis of the data through the pointcut language, to directly select joinpoints in which security protection is injected. Hence, such a construct would limit the dependencies to external static analysis tool and would provide higher coherence wihtin the security coverage.

Listing 15 presents the first construct that categorizes data when passing through an entry point. Our underlying system needs to follow the propagation of the data, hence following the dataflow to propagate the tag information. Listing 16 presents another usage of the new constructs to build and control the tags.

Listing 15 Dataflow tagging with the new construct

```
@Pointcut("call(* javax.persistence.Query.get*(..)) && settag(EntryPoint.REPOSITORY)")
public void pointcutTagDataSource() {
}

@Pointcut("call(* javax.xml.ws.Dispatch.invoke*(..)) && settag(EntryPoint.WEBSERVICE)")
public void pointcutTagWebServiceDispath() {
}

//...
```

Listing 16 Intercepting and tagging with the new constructs

```
@Pointcut("call(* com.sap.businessmodel.accounting.cashflow.*(..))"
    "&& hastag(EntryPoint.WEBSERVICE) && settag(Business.ACCOUNTING)")
public void pointcutTagAccounting() {
}
```

Listing 17 presents the second step. Once the data has been tagged by our system, we can define new pointcuts that, depending on the applied tag and the control flow, determine new joinpoints with en enhanced expressivity. The pointcut would allow parameters passing to the advice to enable complete context retrieval: control flow history including entry points, tags that have been applied, etc.

Listing 17 Parametric advice with new construct

```
@Around("hastag(EntryPoint.WEBSERVICE) && call(java.io.PrintWriter.print(arg)) && String(arg)")
public void doApplyValidationField(final JoinPoint jp
    , final ControlFlow flow, String value)
    throws Exception {
    final Signature signature = jp.getSignature();
    final ControlFlowHistory history = flow.getHistoric();
    final List<Tag> tags = flow.getTags(value);
}
```

These small snippets are for example purpose only to indicate what kind of interaction one would like with new primitives.

8.1.2 Automated source classification for input verification

The model and tools we present in this contribution are close to the type of pointcut constructs we would like to introduce in this thesis. In Chapter 4, we are combining different techniques, and delegating manual processing to the developer for few tasks, such as annotating code in the application to indicate enhanced data-types. The goal is to have a consistent view on data meta-information for a systematic validation of inputs. The manual processing can be easily automated with our pointcut constructs that observe entry points of a program and decide of the correct verification code to introduce.

Our focus is to augment aspect expressivity to let an aspect developer intercept points in the application where he can introduce the verification code for a given data-type. The actual type of the input is a-priori not known in the context of the application. This is the reason for which we were annotating the source code. With our approach, we would like to replace the annotation phase by an automated discovery of the actual enhance data-types. There are sources that provide complete description of the type, such as a database schema, or an xml schema. But the information might also comes from a policy file, or directly from a client along the data. In any case, our solution would give the possiblity to extract information from elements outside of the scope of the application to integrate the context into the application.

Then, an aspect developer would create aspects that rely on this information. For this purpose, we need a system that intercept elements between layers (technical layers such as repositories, controller, service framework, communication, *etc.*).

8.1.3 Cross-layer context passing

In Chapter 5, we introduce REST Security, which allows flexible application of security on messages. It simplifies the problem of handling security properties in the context of multiple actor collaboration. We presented use cases in which REST Security provides novel approach. The enforcement of security properties in cross-domain collaboration is implemented thanks to dedicated reference monitors. Although reference monitors fill the requirements to analyze traffic and apply security transformations, one would need to preserve the resource state in term of security. For instance, a security policy would force a resource to come encrypted in an application, and the reference monitor would transparently decrypt it. The problem in such case, is that we loose the transformation history. The application which handle the resource is unable to know if the resource was actually protected, and from which moment the resource has been transformed in clear text.

Such behavior can benefit to the handling of any resource on the application. At anytime, a developer would be able to retrace the history of actions, its provenance, *etc.* We need to have mechanisms that pass context through layers, and that allow us to propagate the context during the execution of an application. We would also be able to investigate an automated way to configure services to enable security transformations when necessary, i.e. when the resource is sensitive or contains restricted information.

8.1.4 Inference protection

The work we have presented in Chapter 6 does not allow us to prevent inference of data, as we don't collect enough context to detect such inference. In [BBA$^+$13], the authors present a privacy preserving composition execution system. This approach is interesting as it allows the composition of several services to provide results that respect a k-anonymization. They target *data Services*, which are somehow atomic services to access data source. Instead, we position our solution to run at the application layer, while still communicating with data-source layers and service layer.

In the approach described in Chapter 6, we are relying on custom declaration from developers or administrators in order to provide privacy for customer's of the application that is hosted in a cloud platform. The problematic when we introduce such solution is to ensure to the different stakeholders that they can provide protection against data inference to their customers. It means that we need to have protection against a person or a group of persons that correlate application's results to extract personal identifier data.

We would like to assert that privacy requirement is fully respected without leaving the application level. The problem is that for inference protection, we need to gather several requests coming in the application and analyze them.

The requirements to achieve such detection is to have a system capable of capturing the numerous requests and responses during the application's execution. In the following (cf Listing 18, we are highlighting a potential implementation that could rely on our set of pointcut and advice proposal. In a natural language, the protection against such attack would lead to the following process

- Determine identity of the requestor and the query intent
- Tag the data extracted from the table
- Compare the data from the history of extracted data
- Allow or deny access in case of data leakage risk

Listing 18 Example of protection against data inference with data access for several purposes

```
1  @Pointcut("within(@PiiAccessClass *) && call(@Info * *(..))")
2  public void pointcutPiiPurpose(){}
3
4  @Before("pointcutPiiPurpose()")
5  public void setRequestorIdentity(JoinPoint joinpoint, ControlFlow flow ){
6
7      User user = (User) SecurityContextHolder.getContext().getAuthentication().getPrincipal();
8      Purpose purpose = getPurposeForCurrentJointPoint(joinpoint);
9      flow.addPrincipal(user, joinpoint);
10     flow.setTag(joinpoint, "Purpose", purpose);
11 }
12
13
14 @AfterReturning(pointcut="pointcutPiiPurpose()", returning="list")
15 public void setDataFlowInformation(JoinPoint joinpoint, ControlFlow flow, List<ShopHistory> histories){
16     flow.setTag(joinpoint, EntryPoint.REPOSITORY, histories);
17 }
18
19 @Pointcut("hastag(EntryPoint.REPOSITORY, purpose)")
20 public void pointcutTaggedData(){}
21
22 @Before("pointcutTaggedData()")
23 public void verifyInference(JoinPoint joinpoint, ControlFlow flow, History history){
24     Purpose purpose = flow.getTag("Purpose");
25     if (!history.containsPrincipal(flow.getPrincipal(data)))
26         return;
27
28     Purpose previousPurpose = history.getTags(flow.getPrincipal(data)).getTag("Purpose")
29     if (purpose != previousPurpose)
30         throw new DataLeakInformationException("Principal user already access some data with different purpose");
31
32     }
33 }
```

Although we haven't implemented the solution presented above, it gives few indicators and prerequisites to handle such problematic. The Listing 18 describes at line 1 a pointcut to detect methods annotated with the $@Info$ annotation and that are contained in a class with

$@PiiAccessClass$ annotation. The corresponding advice that get principal information is presented at line 5. In our snippet, we also tag the control flow context. The advice at line 14 indicates that a specific object has for origin a repository. At line 23, another advice is executed when data is loaded from a repository and that there is a specific purpose associated.

8.2 Requirements

In the previous sections, we have presented several motivations towards new constructs for pointcuts and for a complete aspect system that cross-cut layers. The requirements were first discussed in [ISR$^+$11].

The aspect model we envision is based on the pointcut-advice model for aspects, with some important extensions to be applied. The pointcut-advice model is characterized by three main abstractions: aspects, pointcuts and advice that together provide means for the concise definition and efficient implementation of so-called cross-cutting functionalities of a base application, such as security, that cannot typically be modularized with existing structuring and encapsulation mechanisms, such as services or components. We address these requirements by the following set of major characteristics that the aspect model has to fulfill. These characteristics are for most of them general in the sense that they apply to all three basic aspect abstractions (aspects, pointcuts and advice) - except if stated otherwise in the following:

- ***Basic abstractions and relations***: The pointcut language should enable referencing all relevant abstractions of the model and the concrete infrastructures; the advice language allows to manipulate these entities. Relevant relationships between them include relations between adjacent abstraction levels or the ability to protect some of them using certain security mechanisms, such as access control, while others may not be modified by that security mechanism.

- ***Composition model***: The aspect model should provide a gray-box composition model, i.e., aspects may access parts of application implementations. However, such access can be restricted by explicit fine-grained conditions on the structure and behavior of the underlying base system. The aspect model will therefore provide strong control over invasive composition. Corresponding conditions will be defined as part of evolution tasks through the aspects that realize them. The conditions may then be integrated before execution in the runtime representations of aspects or the underlying infrastructure, or enforced, possibly at execution time, on implementations.

- ***Dynamic application***: Aspects should be applicable dynamically even though static application strategies may also be used, especially for the introduction of security mechanisms that would suffer from an excessive overhead. Many current aspect models only support a static or load-time application of aspects, which severely limits their applicability for many composition tasks. Our model therefore significantly broadens the use of aspects to many real-world scenarios that involve highly dynamic applications. Another general characteristic of our model is that the model enables the aspect-based definition of service evolutions whose (security) properties can be formally analyzed.

- ***Protocol support***: The pointcut language should include direct support for matching (parts of) protocols that govern the collaboration (choreography etc.) between entities of the model. The advice language permits the manipulation of protocols.

- ***Local state***: Aspects may contain local state that can be used to modify state of the base application. Aspect definitions may, however, restrict the kind of state that can be defined and used.

Even though our solution is not fully developed yet, we highlighted the need for a new approach to ease cross-layer propagation of resource state. Aspect-based techniques meet our requirements to express and apply concerns that are normally difficult to address as their impact is scattered in a distributed environment.

8.3 Aspect System and Language

Despite the advances and the flexibility of the solutions we propose in this thesis, we lack of a correct aspect support in some situations. For instance, the analysis of the different projects we conducted lead us to a situation in which pointcuts were not expressive enough, and advices were not carrying enough information to treat the problems. In situation of security issues, it is important to work on a snapshot view of the application context. The aspect system give us tools to quickly introduce some code tangled and scattered over the application.

We establish some foundations to new pointcut primitives with an underlying system to monitor data flow, while integrating the solution in the existing aspect languages. The Figure 8.1 provides an overview of the interaction between the components of the solution.

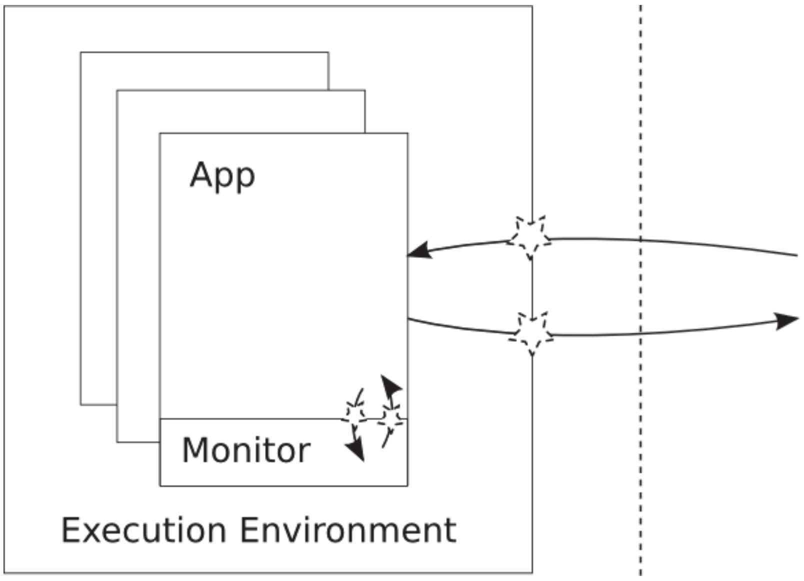


Figure 8.1: Architecture of the solution

The execution environment depends on the actual implementation. It is generally a JVM for java applications, but it can also be an application server. The important notion is to have a component with clear boundaries between the external domain and the internal domain. A first adaptation is processed for all inputs that go from external domain to the internal domain. The first point builds if necessary the state of the resource that is coming into the application, and passes the created context to the execution environment. The execution environment launches the application, which is itself under a constant monitoring. The monitoring component is able to store information either in a static way, or per request (equivalent to ThreadLocal in java). The monitoring component can behave like an aspect engine. The goal for this monitoring component is to maintain an history of transformations and propagation of resources within the application. When resource gets out of the application, the execution environment has the possibility to attach meta-information to the resource. The information can be the result of an analysis or taken from the monitor of the application.

An aspect system using these components would

1. detect inputs and classify input source
2. expose data state as a selector through new pointcut primitive. The primitive can be parameterized
3. expose application context and history of the dataflow at a specific joinpoint, when writing an advice

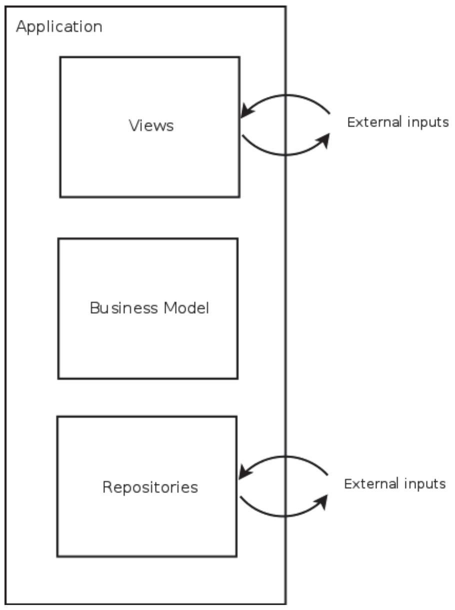


Figure 8.2: Application layers

The Figure 8.2 presents the traditional layers of an application. The layers intensively use data from external sources. Detecting all inputs to classify them is not necessarily trivial. We list in Table 8.1 the different source of origins of the external inputs. We also associate the potential actors that can provide these inputs. The trust level highly depends on the actor that provide the input. For instance, an administrator has potentially more weight than a client. Although a developer should verify all inputs, the most sensitive are the one from the clients.

Input	**Actor**
program arguments	developers or architects or system users
http parameters	clients
property files	developers
system properties	architects or system users
databases	developers or administrators
local and remote filesystem	system users
command line input	system users
inter process communication	developers

Table 8.1: Extensive list of source origins with their potential actors

Each of the input source might introduce some data, that is defined out of the scope of the application. From a security point of view, these data could not be trusted as they are issued by a domain external to the application domain we are taking care of. A new construct would be necessary, part of the aspect language, to support the simple tagging of data provenance.

We provide in Table 8.2 a list of constructs we would like to have in our system. These constructs are there to allow aspect developers to write pointcuts that maintain a context with several tags associated to resources within an application. The different components that we have presented in 8.1 are there to gather data, data that is used by the aspect system to determine if a pointcut matches or not.

The language would define two kinds of context. *Request context* is a context that is valid for the execution of one request. For example, if the execution environment detects that an input message contains a signature, it will attach a tag to the message. Then, in the application, one aspect developer might create a pointcut that verifies that a resource contains a signature to match only necessary joinpoints. *Application context* requests are tags that are valid for longer than a request. They can be assimilated to static tags, that are valid for the lifetime of the application. For example, in the case of the privacy use case we were describing in the previous section, to protect against inference of data, one aspect developer can use global tags to maintain context across requests.

8.4 Related work

Dynamic taint propagation in java is widely explained in [HCF05]. They present a solution that goes from sources to sinks through propagation means. Like our approach, the sources are identified in advance. The propagation mechanisms are the derivation of any tainted string to a new string. And the sinks are methods that consumes input or derivative of a user input. Although we adopt the same approach for the execution platform taint propagation, we approach

Construct	**Context**	**Definition**
settag(tagname, resource)	request context	associate a tag with name $tagname$ to the application flow. One can attach a resource to the tag.
gettag(tagname)	request context	retrieve a resource associated to the tag with name $tagname$
hastag(tagname, resource)	request context	determine if a tag is set for the current application flow or attached to the given resource.
setglobaltag(tagname, resource)	application context	associate a tag with name $tagname$ to the application. One can attach a static resource to the tag.
getglobaltag(tagname)	application context	retrieve a static resource associated to the tag with name $tagname$
hasglobaltag(tagname, resource)	application context	determine if a tag is set for the application or attached to the given static resource.

Table 8.2: Constructs for the language

the problem a bit differently. Our goal is not only to detect the problems, but to let aspect developers introducing any advice on this points. The work from Haldar et al. restricts the analysis to the execution environment. For our approach, we are in a distributed scenario in which we need to propagate the information from and to external consumers.

Another approach that is close to ours is from Suh et al. [SLZD04]. The authors are interested in secure program execution via dynamic information flow tracking. Their approach involves the collaboration of several layers altogether: operating system tags potential malicious data, that is then tracked in program execution. At the end, some restriction might be enforced to mitigate specific attacks. Hiet et al. [HTMM08] tackle the problem to detect intrusion detection rather than sensitive data, but the overall approach is similar. They combine several layers and components. They use two monitors, at the operating system level (Blare) and at the application level (JBLare) to track information flow. In top of these monitoring components, they allow policy-based detection of the attacks. Hauser et al.[HTFM13] continue in the same direction to provide intrusion detection based on taint marking. Our proposal approach would generalize a step further the composition between the different layers. We intend to combine distributed components to share automatically and transparently information about data. In addition, we are interessted in giving enough tools to the developers to correct the problem, but we don't want to directly apply a policy. We prefer to build a solid intermediate layer (the aspect pointcuts) to let any aspect developer the care to enforce security policies.

In [ABA$^+$12], the authors aim to provide an end-to-end security auditing approach for service oriented architecture. They use the concept of dynamic taint analysis by introducing analysis components in the flow of messages. From the messages, they can decide if there are violations of security policies. Although they introduce aspect-oriented programming (JBoss implementation) in their solution, they do not address the same problem as we are. They use AOP to instrument communication methods and thus, intercept service invocation. Yet, they are able to extract audit information from the traffic, but they don't connect the communication layer with the application layer. In our approach, we are interested to give tools and means to both developers within the application, but also for external components (reference monitors, auditing frameworks, etc.) that can access the security meta-data that we provide to further react.

8.5 Conclusion

We have presented in this chapter an approach to give new tools to aspect developers. We place ourselves in a distributed environment, that is composed of external components and a local execution environment. The application (running on the local execution environment) is communicating with several external parties, that all have potential security needs. In order to enhance trust in data that pass through all these intermediaries, and in order to facilitate verification and transformation of security properties, we define new tools and means to intercept data with aspects, extract security extra-information, and propose the extracted information to the aspect developers. We therefore allow local aspect developers to react on application events (encrypted data comes in, a locally modified data is about to gets out), as the system transparently attaches security meta-data to resources. Our solution wraps the application to extract information from the communication layer to the application layer when data comes in, and reverse the process when data goes out. Such approach opens new automatic collaborations between actors that are not always possible.

Bibliography

[ABA$^+$12] Mehdi Azarmi, Bharat K. Bhargava, Pelin Angin, Rohit Ranchal, Norman Ahmed, Asher Sinclair, Mark Linderman, and Lotfi Ben Othmane. An end-to-end security auditing approach for service oriented architectures. In *SRDS*, pages 279–284. IEEE, 2012.

[ACBC10] Nabil Ajam, Nora Cuppens-Boulahia, and Frédéric Cuppens. Privacy administration in distributed service infrastructure. In Sushil Jajodia and Jianying Zhou, editors, *SecureComm*, volume 50 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 53–70. Springer, 2010.

[Age13] Agence Nationale de la Sécurité des systèmes d'information. Centre d'expertise gouvernemental de réponse et de traitement des attaques informatiques. `http://www.ssi.gouv.fr`, 2013.

[AHK$^+$03] P. Ashley, S. Hada, G. Karjoth, C. Powers, and M. Schunter. Enterprise privacy authorization language (epal). *Research report*, 3485, 2003.

[AKSX03] Rakesh Agrawal, J. Kiernan, Ramakrishnan Srikant, and Y. Xu. Implementing p3p using database technology. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 595 – 606, march 2003.

[Ama06] Amazon. Amazon Simple Storage Service REST Security Model. `http://docs.amazonwebservices.com/AmazonS3/latest/dev/RESTAPI.html`, 2006.

[Apa04] Apache Foundation. Apache roller. `http://rollerweblogger.org/project/`, 2004.

[BBA$^+$13] Mahmoud Barhamgi, Djamal Benslimane, Youssef Amghar, Nora Cuppens-Boulahia, and Frédéric Cuppens. Privcomp: a privacy-aware data service composition system. In Giovanna Guerrini and Norman W. Paton, editors, *EDBT*, pages 757–760. ACM, 2013.

[BBL05] Ji-Won Byun, Elisa Bertino, and Ninghui Li. Purpose based access control of complex data for privacy protection. In *Proceedings of the tenth ACM sympo-*

sium on Access control models and technologies, SACMAT '05, pages 102–110, New York, NY, USA, 2005. ACM.

[BCF$^+$08] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 387–401, Washington, DC, USA, 2008. IEEE Computer Society.

[BDR00] Alexandre M. Braga, Ricardo Darab, and Cecília M. F. Rubira. A meta-object protocol for secure composition of security mechanisms. In Peri Tarr, Lodewijk Bergmans, Martin Griss, and Harold Ossher, editors, *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*, October 2000.

[Bel08] Nadia Belblidia. An aspect oriented approach for security hardening : semantic foundations. 2008.

[BHM$^+$04] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Mike Champion, Christopher Ferris, and David Orchard. Web services architecture. *http://www.w3.org/TR/ws-arch/*, 99(7):1–100, January 2004.

[BM04] David Balzarotti and Mattia Monga. Using program slicing to analyze aspect oriented composition. In Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors, *FOAL: Foundations Of Aspect-Oriented Languages*, pages 25–30, March 2004.

[BNP11] Laurent Bussard, Gregory Neven, and Franz-Stefan Preiss. Matching privacy policies and preferences: Access control, obligatons, authorisations, and downstream usage. In Jan Camenisch, Simone Fischer-Hübner, and Kai Rannenberg, editors, *Privacy and Identity Management for Life*, pages 117–134. Springer Berlin Heidelberg, 2011.

[Bod04] Ron Bodkin. Enterprise security aspects. In De Win et al. [DSJB04].

[BP09] E. Bernard and S. Peterson. Jsr 303: Bean validation, bean validation expert group, March 2009.

[Bun08] Bundesamt für Sicherheit in der Informationstechnik. Information security management systems, 05 2008.

[CDR$^+$13] Ronan-Alexandre Cherreau, Rémi Douence, Jean-Claude Royer, Mario Sudholt, Anderson Santana de Oliveira, Yves Roudier, and Matteo Dell'Amico. Reference monitors for security and interoperability in oauth 2.0. 6th International Workshop on Autonomous and Spontaneous Security, SETOP 2013, Egham, U.K., September 2013.

[CER] CERT. Operationally Critical Threat, Asset, and Vulnerability Evaluation (OCTAVE). `http://www.cert.org/octave/`.

[Cha07] Anis Charfi. *Aspect-Oriented Workflow Languages*. PhD thesis, TU Darmstadt, July 2007.

[Che12] Shay Chen. The web application vulnerability scanner evaluation project - v1.2. `https://code.google.com/p/wavsep/`, July 2012.

[Clo09] Cloud Security Alliance. Security guidance for critical areas of focus in cloud computing v2.1. `http://www.cloudsecurityalliance.orgcsaguide.pdf`, 12 2009.

[CM07] S. M. Christey and R. A. Martin. Vulnerability type distributions in cve. http://cwe.mitre.org/documents/vuln-trends/index.html, 2007.

[CMRW07] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. Technical report, W3C, June 2007.

[CMS03] Aske Simon Christensen, Anders Moller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium*, SAS'03, pages 1–18. Springer-Verlag, 2003.

[CNS99] Sara Cohen, Werner Nutt, and Alexander Serebrenik. Rewriting aggregate queries using views. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '99, pages 155–166, New York, NY, USA, 1999. ACM.

[Coh04] Tal Cohen. AspectJ2EE = AOP + J2EE Towards an Aspect Based, Programmable and Extensible Middleware Framework. In *European Conference on Object-Oriented Programming*, 2004.

[Coh07] Tal Cohen. *Applying Aspect-Oriented Software Development to Middleware Frameworks*. PhD thesis, Technion - Israel Institute of Technology, 2007.

[CPL09] Lawrence Chung and Julio Cesar Prado Leite. Conceptual modeling: Foundations and applications. chapter On Non-Functional Requirements in Software Engineering, pages 363–379. Springer-Verlag, Berlin, Heidelberg, 2009.

[Cra03] L.F. Cranor. P3p: making privacy policies more useful. *Security Privacy, IEEE*, 1(6):50 – 55, nov.-dec. 2003.

[CV01] Denis Caromel and Julien Vayssière. Reflections on mops, components, and java security. In Jørgen Lindskov Knudsen, editor, *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 256–274. Springer, 2001.

[CW07] Kung Chen and Da-Wei Wang. An aspect-oriented approach to privacy-aware access control. In *Machine Learning and Cybernetics, 2007 International Conference on*, volume 5, pages 3016 –3021, aug. 2007.

[Dai09] Wei Dai. Crypto++ 5.6.0 benchmarks. `http://www.cryptopp.com/benchmarks.html`, 2009.

[DBO05] Prattana Deeprasertkul, Pattarasinee Bhattarakosol, and Fergus O'Brien. Automatic detection and correction of programming faults for software applications. *Journal of Systems and Software*, 78(2):101–110, 2005.

[DFH06] Josh Dehlinger, Qian Feng, and Lan Hu. Ssvchecker: unifying static security vulnerability detection tools in an eclipse plug-in. In *Proc. OOPSLA Workshop on eclipse technology eXchange*, Eclipse'06, pages 30–34. ACM, 2006.

[DGM+10] R. Douence, H. Grall, I. Mejía, et al. Survey and requirements analysis. Deliverable D1.1, The CESSA project, June 2010.
`http://cessa.gforge.inria.fr/lib/exe/fetch.php?media=publications:d1-`
.

[Dro12] Dropbox. REST API. `https://www.dropbox.com/developers/reference/api`, 2012.

[DS06] Josh Dehlinger and Nalin Subramanian. Architecting secure software systems using an aspect-oriented approach: A survey of current research, 2006.

[DSI+12a] Matteo Dell'Amico, Gabriel Serme, Muhammad Sabir Idrees, Anderson Santana de Oliveira, and Yves Roudier. Hipolds: A security policy language for distributed systems. In Ioannis G. Askoxylakis, Henrich Christopher Pöhls, and Joachim Posegga, editors, *WISTP*, volume 7322 of *Lecture Notes in Computer Science*, pages 97–112. Springer, 2012.

[DSI+12b] Matteo Dell'Amico, Gabriel Serme, Muhammad Sabir Idrees, Anderson Santana de Olivera, and Yves Roudier. Hipolds: A security policy language for distributed systems. In *Workshop in Information Security Theory and Practice. WISTP 2012*, jun. 2012.

[DSI+13] Matteo Dell'Amico, Gabriel Serme, Muhammad Sabir Idrees, Anderson Santana de Oliveira, and Yves Roudier. Hipolds: A hierarchical security policy language for distributed systems. *Inf. Sec. Techn. Report*, 17(3):81–92, 2013.

[DSJB04] Bart De Win, Viren Shah, Wouter Joosen, and Ron Bodkin, editors. *AOSDSEC: AOSD Technology for Application-Level Security*, March 2004.

[Eur09] European Network and Information Security Agency. How to raise information security awareness, November 2009.

[Fac12] Facebook. Facebook Authentication. `http://developers.facebook.com/docs/authentication/`, 2012.

[Far01] Andrés Farías. Towards a security aspect for java. 2001.

[Fie00] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. Chair-Taylor,, Richard N.

[Fou] OWASP Foundation. The open web application security project (owasp foundation) owasp testing guide v3.0. `http://www.owasp.org/index.php/OWASP_Testing_Project`.

[Fou11] The Apache Software Foundation. Struts 2, 2011.

[FSJ08] Bruno De Fraine, Mario Südholt, and Viviane Jonckers. Strongaspectj: flexible and safe pointcut/advice bindings. In Theo D'Hondt, editor, *AOSD*, pages 60–71. ACM, 2008.

[GEKS11] Marco Guarnieri, Paul El Khoury, and Gabriel Serme. Security vulnerabilities detection and protection using eclipse. In *Proceedings of ECLIPSE-IT 2011*, aug. 2011.

[GHM$^+$07] M. Gudgin, M. Hadley, N. Mendelsohn, J.J.Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon. SOAP version 1.2 part 1: Messaging framework (second edition). W3C recommendation, W3C, April 2007.

[GHP13] YOEL GLUCK, NEAL HARRIS, and ANGELO PRADO. Breach: Reviving the crime attack. In *BlackHat*, 2013.

[GIJ$^+$12] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating ssl certificates in non-browser software. In *ACM Conference on Computer and Communications Security*, pages 38–49, 2012.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. Java(TM) Language Specification. http://docs.oracle.com/javase/specs/, January 2005.

[GMCF95] J. Galvin, S. Murphy, S. Crocker, and N. Freed. Security multiparts for mime: Multipart/signed and multipart/encrypted. Technical report, IETF, Network Working Group, October 1995. `http://tools.ietf.org/html/rfc1847`.

[Goo12] Google. Codepro analytix. `http://code.google.com/javadevtools/codepro/`, June 2012.

[GSD04] Carl Gould, Zhendong Su, and Premkumar T. Devanbu. Jdbc checker: A static analysis tool for sql/jdbc applications. In *ICSE*, pages 697–698. IEEE Computer Society, 2004.

[Ha10] Eran Hammer and al. The oauth 1.0 protocol. http://tools.ietf.org/html/rfc5849, April 2010.

[HAJ09] Munawar Hafiz, Paul Adamczyk, and Ralph Johnson. Systematically eradicating data injection attacks using security-oriented program transformations. In *Proceedings of the 1st International Symposium on Engineering Secure Software and Systems*, ESSoS '09, pages 75–90, Berlin, Heidelberg, 2009. Springer-Verlag.

[Ham10] Eran Hammer. Oauth bearer tokens are a terrible idea. http://hueniverse.com/2010/09/oauth-bearer-tokens-are-a-terrible-idea/, September 2010.

[HCF05] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for java. In *ACSAC*, pages 303–311. IEEE Computer Society, 2005.

[HDA11] Abdelhakim Hannousse, Rémi Douence, and Gilles Ardourel. Static analysis of aspect interaction and composition in component models. In Ewen Denney and Ulrik Pagh Schultz, editors, *GPCE*, pages 43–52. ACM, 2011.

[HG06] Bruno Harbulot and John R. Gurd. A join point for loops in AspectJ. In *Aspect-Oriented Software Development*, pages 63–74, 2006.

[HGSD07] Gabriel Hermosillo, Roberto Gomez, Lionel Seinturier, and Laurence Duchien. Aprosec: an aspect for programming secure web applications. In *ARES*, pages 1026–1033. IEEE Computer Society, 2007.

[HOM06] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 175–185, New York, NY, USA, 2006. ACM.

[Hoo05] J. Hookom. Validating objects through metadata, January 2005.

[HP12] HP. Fortify 360. `https://www.fortify.com/`, June 2012.

[HS04] Richard Hull and Jianwen Su. Tools for design of composite web services. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 958–961, New York, NY, USA, 2004. ACM.

[HTFM13] Christophe Hauser, Frédéric Tronel, Colin Fidge, and Ludovic Mé. Intrusion detection in distributed systems, an approach based on taint marking. In *IEEE ICC2013 - IEEE International Conference on Communications*, Budapest, Hongrie, July 2013.

[HTMM08] Guillaume Hiet, Valérie Viet Triem Tong, Ludovic Mé, and Benjamin Morin. Policy-based intrusion detection in web applications by monitoring java information flows. In Mohamed Jmaiel and Mohamed Mosbah, editors, *CRiSIS*, pages 53–60. IEEE, 2008.

[HWZ04] Minhuan Huang, Chunlei Wang, and Lufeng Zhang. Toward a reusable and generic security aspect library. In De Win et al. [DSJB04].

[HYH$^+$04] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 40–52, New York, NY, USA, 2004. ACM.

[IEKY04] O. Ismail, M. Etoh, Y. Kadobayashi, and S. Yamaguchi. A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability. In *Advanced Information Networking and Applications, 2004. AINA 2004. 18th International Conference on*, volume 1, pages 145 – 151 Vol.1, 2004.

[IET08] IETF. Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile. `http://tools.ietf.org/html/rfc5280#section-4.1.2.2`, 2008.

[IKC09] Wassim Itani, Ayman I. Kayssi, and Ali Chehab. Privacy as a service: Privacy-aware data storage and processing in cloud computing architectures. In *DASC*, pages 711–716. IEEE, 2009.

[Imp11] Imperva. The securesphere web application firewall, 2011.

[Inc11] Barracuda Networks Inc. The barracuda web application firewall, 2011.

[Inf12] Information Security Forum. The standard of good practice for information security, 09 2012.

[ISR$^+$11] Muhammad Sabir Idrees, Gabriel Serme, Yves Roudier, Anderson Santana de Oliveira, Hervé Grall, and Mario Südholt. Evolving security requirements in multi-layered service-oriented-architectures. In Joaquín García-Alfaro, Guillermo Navarro-Arribas, Nora Cuppens-Boulahia, and Sabrina De Capitani di Vimercati, editors, *DPM/SETOP*, volume 7122 of *Lecture Notes in Computer Science*, pages 190–205. Springer, 2011.

[Ist05] IsthmusGroup, Madison Wisconsin. Insecurewebapp. `insecurewebapp.sf.net`, October 2005.

[JB07] Martin Johns and Christian Beyerlein. Smask: preventing injection attacks in web applications by approximating automatic data/code separation. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 284–291, New York, NY, USA, 2007. ACM.

[JBGP10] Martin Johns, Christian Beyerlein, Rosemaria Giesecke, and Joachim Posegga. Secure code generation for web applications. In *ESSoS*, pages 96–113, 2010.

[JBo11] JBoss. Hibernate validator, 2011.

[Jee13] Zubair Jeelani. An insight of ssl security attacks. *International Journal of Research in Engineering and Applied Sciences*, 3:52–61, March 2013.

[JKK06a] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[JKK06b] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 27–36, New York, NY, USA, 2006. ACM.

[JSH07] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 601–610, New York, NY, USA, 2007. ACM.

[KDRB91] Gregor Kiczales, Jim Des Rivieres, and Daniel Gureasko Bobrow. *The art of the metaobject protocol*. The MIT press, 1991.

[Kic01] Gregor Kiczales. Aspect-Oriented Programming - The Fun Has Just Begun. In *Vanderbilt Workshop, New Visions for Software Design & Productivity: Research & Applications. Participant White Papers*, Vanderbilt University, Nashville, TN, 2001.

[KKVJ06] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *SAC'06*, pages 330–337, 2006.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin / Heidelberg, 1997.

[KYL09] Kaarina Karppinen, Lyly Yonkwa, and Mikael Lindvall. Why developers insert security vulnerabilities into their code. *International Conference on Advances in Computer-Human Interaction*, 0:289–294, 2009.

[LAE$^+$04] Kristen LeFevre, Rakesh Agrawal, Vuk Ercegovac, Raghu Ramakrishnan, Yirong Xu, and David J. DeWitt. Limiting disclosure in hippocratic databases. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *VLDB*, pages 108–119. Morgan Kaufmann, 2004.

[Lan02] Marc Langheinrich. A privacy awareness system for ubiquitous computing environments. In Gaetano Borriello and Lars Holmquist, editors, *UbiComp 2002: Ubiquitous Computing*, volume 2498 of *Lecture Notes in Computer Science*, pages 315–320. Springer Berlin / Heidelberg, 2002.

[Lan10] Ulrich Lang. Openpmf scaas: Authorization as a service for cloud & soa applications. In *CloudCom*, pages 634–643. IEEE, 2010.

[Las10] Francois Lascelles. RESTful Web services and signatures. http://flascelles.wordpress.com/2010/10/02/restful-web-services-and-signatures/, October 2010.

[Las13] Tasos Laskos. Arachni 0.4.2 - web application security scanner framework. http://www.arachni-scanner.com/, April 2013.

[Lis88] Barbara Liskov. Data Abstraction and Hierarchy. *Sigplan Notices*, 1988.

[Liv] Ben Livshits. Description of securibench applications. http://suif.stanford.edu/~livshits/work/securibench/descr.html, 2005.

[LKa06a] Kelvin Lawrence, Chris Kaler, and al. Kerberos Token Profile 1.1. http://www.oasis-open.org/committees/download.php/16788/wss-v1.1-spec-os-KerberosTokenProfile.pdf, 2006.

[LKa06b] Kelvin Lawrence, Chris Kaler, and al. SAML Token Profile 1.1. http://www.oasis-open.org/committees/download.php/16768/wss-v1.1-spec-os-SAMLTokenProfile.pdf, 2006.

[LKa06c] Kelvin Lawrence, Chris Kaler, and al. UsernameToken Profile 1.1. http://www.oasis-open.org/committees/download.php/16782/wss-v1.1-spec-os-UsernameTokenProfile.pdf, 2006.

[LKa06d] Kelvin Lawrence, Chris Kaler, and al. X.509 Certificate Token Profile 1.1. http://www.oasis-open.org/committees/download.php/16785/wss-v1.1-spec-os-x509TokenProfile.pdf, 2006.

[LKa09] Kelvin Lawrence, Chris Kaler, and al. Ws-securitypolicy 1.3. http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.3/os/ws-securitypolicy-1.3-spec-os.html, 2009.

[LL05] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[LM10] Yin Liu and Ana Milanova. Static information flow analysis with handling of implicit flows and a study on effects of implicit flows vs explicit flows. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, pages 146–155, Washington, DC, USA, 2010. IEEE Computer Society.

[Lop05] Cristina Videira Lopes. AOP: A historical perspective (What's in a name?). pages 97–122. Addison-Wesley, Boston, 2005.

[LSM05] Nicolas Loriant, Marc Séegura-Devillechaise, and Jean-Marc Menaud. Software security patches: Audit, deployment and hot update. In Yvonne Coady, Eric Eide, David H. Lorenz, and Olaf Spinyzck, editors, *Proceedings of the Fourth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, March 2005.

[LWLa05] Monica S. Lam, John Whaley, V. Benjamin Livshits, and al. Context-sensitive program analysis as database queries. In *Symposium on Principles of database systems*, PODS'05, pages 1–12. ACM, 2005.

[Man05] Anne Thomas Manes. Rest and soap and document-oriented services. http://atmanes.blogspot.com/2005/09/rest-and-soap-and-document-oriented.html, 2005.

[Mar99] Robert C Martin. Designing object oriented applications using uml, 2d, 1999.

[Mic] Microsoft Corporation. Crosscutting concerns. `http://msdn.microsoft.com/en-us/library/ee658105.aspx`.

[MIT09] MITRE. Cwe-20: Improper input validation. `http://cwe.mitre.org/data/definitions/20.html`, January 2009.

[MIT11] MITRE. CWE/SANS Top 25 Most Dangerous Software Errors. `http://cwe.mitre.org/top25`, September 2011.

[MK03] Hidehiko Masuhara and Kazunori Kawauchi. Dataflow pointcut in aspect-oriented programming. In Atsushi Ohori, editor, *APLAS*, volume 2895 of *Lecture Notes in Computer Science*, pages 105–121. Springer, 2003.

[MO12] Bruce Mayhew and OWASP Community. Webgoat 5.4. `https://code.google.com/p/webgoat/`, April 2012.

[Mou08] Azzam Mourad. An aspect-oriented framework for systematic security hardening of software. 2008.

[MP09] Miranda Mowbray and Siani Pearson. A client-based privacy manager for cloud computing. In Jan Bosch and Siobhán Clarke, editors, *COMSWARE*, page 5. ACM, 2009.

[MT04] T. Mens and T. Tourwe. A survey of software refactoring. *Software Engineering, IEEE Transactions on*, 30(2):126 – 139, February 2004.

[MT06] Marco Casassa Mont and Robert Thyne. A systemic approach to automate privacy policy enforcement in enterprises. In George Danezis and Philippe Golle, editors, *Privacy Enhancing Technologies*, volume 4258 of *Lecture Notes in Computer Science*, pages 118–134. Springer, 2006.

[NSV$^+$06] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly distributed AOP using AWED. In *Aspect-Oriented Software Development*, pages 51–62, 2006.

[NTGG$^+$05] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *SEC*, pages 295–308, 2005.

[OAS06] OASIS. Web Services Security : SOAP Message Security 1.1. `http://www.oasis-open.org/committees/wss`, February 2006.

[oST] National Institute of Standards and Technology. NIST. `http://nist.gov/`.

[Oul13] Said Oulmakhzoune. *Enforcement of Privacy Preferences in Data Services: A SPARQL Query Rewriting Approach*. PhD thesis, LUSSI - Dépt. Logique des Usages, Sciences Sociales et de l'Information (Institut Mines-Télécom-Télécom Bretagne-UEB), Lab-STICC - Laboratoire en sciences et technologies de l'information, de la communication et de la connaissance (UMR CNRS 6285 - Télécom Bretagne - Université de Bretagne Occidentale - Université de Bretagne Sud), april 2013. Th. doct. : Informatique, Institut Mines-Télécom-Télécom Bretagne-UEB, UMR CNRS 6285 - Télécom Bretagne - Université de Bretagne Occidentale - Université de Bretagne Sud, april 2013.

[OWA] OWASP. OWASP Top 10. `http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`.

[OWA10] OWASP. OWASP Top Ten Project. `http://www.owasp.org/index.php/OWASP_Top_Ten_Project`, 2010.

[Paw02] Renaud Pawlak. Jac (java aspect components). `http://jac.ow2.org/`, 2002.

[PC09] Siani Pearson and Andrew Charlesworth. Accountability as a way forward for privacy protection in the cloud. In Martin Gilje Jaatun, Gansen Zhao, and Chunming Rong, editors, *CloudCom*, volume 5931 of *Lecture Notes in Computer Science*, pages 131–144. Springer, 2009.

[PCE05] Matthew Payne, Emerson Cargnin, and Niel Eyde. Personal blog. `http://sourceforge.net/projects/personalblog/`, April 2005.

[PCG$^+$08] Christoph Pohl, Anis Charfi, Wasif Gilani, Steffen Göbel, and Birgit Grammel. Adopting Aspect-Oriented Software Development in Business Application Engineering. In *7th International Conference on Aspect-Oriented Development*, 2008.

[PE07] Keshnee Padayachee and Jan H. P. Eloff. An aspect-oriented approach to enhancing multilevel security with usage control: An experience report. In Sio Iong Ao, Oscar Castillo, Craig Douglas, David Dagan Feng, and Jeong-A. Lee, editors, *IMECS*, Lecture Notes in Engineering and Computer Science, pages 1060–1065. Newswood Limited, 2007.

[PMMD05] Jaime A. Pavlich-Mariscal, Laurent Michel, and Steven A. Demurjian. A formal enforcement framework for role-based access control using aspect-oriented programming. In Lionel C. Briand and Clay Williams, editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 537–552. Springer, 2005.

[PVB05] Tadeusz Pietraszek, Chris V, and En Berghe. Defending against injection attacks through context-sensitive string evaluation. In *In Recent Advances in Intrusion Detection (RAID*, 2005.

[PZL08] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. "big" web services: making the right architectural decision. In *WWW*, pages 805–814. ACM, 2008.

[RD12] Juliano Rizzo and Thai Duong. The crime attack. In *Ekoparty*, 2012.

[Rea12] David Recordon and Dick Hardt et al. The oauth 2.0 authorization framework. http://tools.ietf.org/html/rfc6749, October 2012.

[Ria11] Andres Riancho. W3af 1.0 - open source web application security scanner. `http://w3af.org/`, May 2011.

[RMSR04] Shariq Rizvi, Alberto Mendelzon, S. Sudarshan, and Prasan Roy. Extending query rewriting techniques for fine-grained access control. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 551–562, New York, NY, USA, 2004. ACM.

[Roo07] SANS Institute InfoSec Reading Room. XML Firewall Architecture and Best Practices for Configuration and Auditing. `http://www.sans.org/reading_room/whitepapers/firewalls/xml-firewall-architecture-practices-configuration-auditing_1766`, 2007.

[RS07] Mohammad Ashiqur Rahaman and Andreas Schaad. Soap-based secure conversation and collaboration. In *ICWS*, pages 471–480. IEEE Computer Society, 2007.

[RV09] William Robertson and Giovanni Vigna. Static enforcement of web application integrity through strong typing. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 283–298, Berkeley, CA, USA, 2009. USENIX Association.

[SBK11] Theodoor Scholte, Davide Balzarotti, and Engin Kirda. Quo vadis? a study of the evolution of input validation vulnerabilities in web applications. In *Proceedings of Financial Cryptography and Data Security 2011*, Lecture Notes in Computer Science, February 2011. To Appear.

[SBM08] Andreas Sewe, Christoph Bockisch, and Mira Mezini. Aspects and class-based security: a survey of interactions between advice weaving and the java 2 security model. In *Proceedings of the 2nd Workshop on Virtual Machines and Intermediate Languages for emerging modularization mechanisms*, VMIL '08, pages 3:1–3:7, New York, NY, USA, 2008. ACM.

[SdOS12] Anderson Santana de Oliveira and Gabriel Serme. Use-case analysis and aspect requirements. Deliverable D3.2, The CESSA project, April 2012. `http://cessa.gforge.inria.fr/lib/exe/fetch.php?media=publications:d3-2.pdf`.

[Sel58] O. G. Selfridge. Pandemonium: a paradigm for learning. In Mechanisation of Thought Processes. In *Proceedings of a Symposium Held at the National Physical Laboratory*, pages 513–526, London, 1958. HMSO.

[SGEKSDO12] Gabriel Serme, Marco Guarnieri, Paul El Khoury, and Anderson Santana De Oliveira. Towards assisted remediation of security vulnerabilities. In *The Sixth International Conference on Emerging Security Information, Systems and Technologies, 2012. SECURWARE 2012*, aug. 2012.

[SK04] Maximilian Störzer and Christian Koppen. Pcdiff: Attacking the fragile pointcut problem, abstract. In *European Interactive Workshop on Aspects in Software*, Berlin, Germany, September 2004.

[SLZD04] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *ACM Sigarch Computer Architecture News*, 32:85–96, 2004.

[Sou11] Spring Source. Spring web mvc, 2011.

[SPGK$^+$03] P. Sandoz, S. Pericas-Geertsen, K. Kawaguchi, M. Hadley, and E. Pelegri-Llopart. Fast web services. *Sun Developer Network*, 2003.

[SRBK12] Theodoor Scholte, William K. Robertson, Davide Balzarotti, and Engin Kirda. Preventing input validation vulnerabilities in web applications through automated type analysis. In Xiaoying Bai, Fevzi Belli, Elisa Bertino, Carl K. Chang, Atilla Elçi, Cristina Cerschi Seceleanu, Haihua Xie, and Mohammad Zulkernine, editors, *COMPSAC*, pages 233–243. IEEE Computer Society, 2012.

[SS02] David Scott and Richard Sharp. Abstracting application-level web security. In *Proceedings of the 11th international conference on World Wide Web*, WWW '02, pages 396–407, New York, NY, USA, 2002. ACM.

[SSdOMR12] Gabriel Serme, Anderson Santana de Oliveira, Julien Massiera, and Yves Roudier. Enabling message security for restful services. In *19th International Conference on Web Services*, jun. 2012.

[SSO13] Gabriel Serme, Theodoor Scholte, and Anderson Santana De Oliveira. Enforcing input validation through aspect-oriented programming. In *SETOP 2013, 6th International Workshop on Autonomous and Spontaneous Security, 12-13 September 2013, Rhul, Egham, UK*, Rhul, UNITED KINGDOM, 09 2013.

[STT05] Toyotaro Suzumura, Toshiro Takase, and Michiaki Tatsubori. Optimizing web services performance by differential deserialization. In *ICWS*, pages 185–192. IEEE Computer Society, 2005.

[SW13] Bojan Simic and James Walden. Eliminating sql injection and cross site scripting using aspect oriented programming. In *International Symposium on Engineering Secure Software and System (ESSoS 13)*, Paris, France, February 2013.

[Swe02] L. Sweeney. k-anonymity: A model for protecting privacy. *International Journal on Uncertainty Fuzziness and Knowledgebased Systems*, 10(5):557–570, 2002.

[SZ03] Pawel Slowikowski and Krzysztof Zielinski. Comparison study of aspect-oriented and container managed security. In Jan Hannemann, Ruzanna Chitchyan, and Awais Rashid, editors, *Analysis of Aspect-Oriented Software (ECOOP 2003)*, July 2003.

[Tru11] Trustwave. Trustwave webdefend - web application firewall, 2011.

[TS12] Slim Trabelsi and Jakub Sendor. Sticky policies for data control in the cloud. In Nora Cuppens-Boulahia, Philip Fong, Joaquín García-Alfaro, Stephen Marsh, and Jan-Philipp Steghöfer, editors, *PST*, pages 75–80. IEEE, 2012.

[TSGW09] Amin Tootoonchian, Stefan Saroiu, Yashar Ganjali, and Alec Wolman. Lockr: better privacy for social networks. In Jörg Liebeherr, Giorgio Ventre, Ernst W. Biersack, and S. Keshav, editors, *CoNEXT*, pages 169–180. ACM, 2009.

[Twi11] Twitter. Security Best Practices. `https://dev.twitter.com/docs/security-best-practices`, 2011.

[Uni12] University of Maryland. Findbugs. `http://findbugs.sourceforge.net`, July 2012.

[USB09] Cedric Ulmer, Gabriel Serme, and Yohann Bonillo. Enabling web object orientation with mobile devices. In *Mobility Conference*. ACM, 2009.

[VBC01] John Viega, J. T. Bloch, and Pravir Ch. Applying aspect-oriented programming to security. *Cutter IT Journal*, 14:31–39, 2001.

[VBKM00] John Viega, J. T. Bloch, Y. Kohno, and Gary McGraw. Its4: A static vulnerability scanner for c and c++ code. In *ACSAC*, pages 257–. IEEE Computer Society, 2000.

[Vir12] Virtual Forge. Codeprofilers. `http://www.codeprofilers.com/`, June 2012.

[VNJ$^+$07] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. February 2007.

[Win04] Bart De Win. Engineering application-level security through aspect-oriented software development. March 2004.

[WJD03] Eric Wohlstadter, Stoney Jackson, and Premkumar T. Devanbu. DADO: enhancing middleware to support crosscutting features in distributed, heterogeneous systems. In *International Conference on Software Engineering*, pages 174–186, 2003.

[WS02] Ian S. Welch and Robert J. Stroud. Security and aspects: A metaobject protocol viewpoint. In Yvonne Coady, Eric Eide, David H. Lorenz, Mira Mezini, Klaus Ostermann, and Roman Pichler, editors, *First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-2002)*, March 2002.

[WS04] Gary Wassermann and Zhendong Su. An analysis framework for security in web applications. In *Proc. FSE Workshop on Specification and Verification of Component-Based Systems*, SAVCBS'04, pages 70–78, 2004.

[WS07] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 32–41, New York, NY, USA, 2007. ACM.

[WS08] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 171–180, New York, NY, USA, 2008. ACM.

[WTD02] Eric Wohlstadter, Brian Toone, and Prem Devanbu. A framework for flexible evolution in distributed heterogeneous systems. In *Proceedings of the Workshop on Principles of Software Evolution*, pages 39–42. ACM Press, 2002.

[XA06] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, Berkeley, CA, USA, 2006. USENIX Association.

[XCLM11] Jing Xie, Bill Chu, Heather Richter Lipford, and John T. Melton. Aside: Ide support for web application security. In Robert H'obbes' Zakon, John P. McDermott, and Michael E. Locasto, editors, *ACSAC*, pages 267–276. ACM, 2011.

[Yah] Yahoo. OAuth Authorization Model. `http://developer.yahoo.com/oauth/`.

[YAM$^+$11] Fan Yang, Tomoyuki Aotani, Hidehiko Masuhara, Flemming Nielson, and Hanne Riis Nielson. Combining static analysis and runtime checking in security aspects for distributed tuple spaces. In Wolfgang De Meuter and Gruia-Catalin Roman, editors, *COORDINATION*, volume 6721 of *Lecture Notes in Computer Science*, pages 202–218. Springer, 2011.

[YSSSdO12] Peng Yu, Jakub Sendor, Gabriel Serme, and Anderson Santana de Oliveira. Automating privacy enforcement in cloud platforms. In Javier Herranz Roberto Di Pietro, editor, *7th International Workshop on Data Privacy Management*. Springer, sep. 2012.

La modularisation de la sécurité informatique dans les systèmes distribués

Gabriel SERME

RESUME : Intégrer les problématiques de sécurité au cycle de développement logiciel représente encore un défi à l'heure actuelle, notamment dans les logiciels distribués. La sécurité informatique requiert des connaissances et un savoir-faire particulier, ce qui implique une collaboration étroite entre les experts en sécurité et les autres acteurs impliqués. La programmation à objets ou à base de composants est communément employée pour permettre de telles collaborations et améliorer la mise à l'échelle et la maintenance de briques logicielles. Malheureusement, ces styles de programmation s'appliquent mal à la sécurité, qui est un problème transverse brisant la modularité des objets ou des composants. Nous présentons dans cette thèse plusieurs techniques de modularisation pour résoudre ce problème.

Nous proposons tout d'abord l'utilisation de la programmation par aspect pour appliquer de manière automatique et systématique des techniques de programmation sécurisée et ainsi réduire le nombre de vulnérabilités d'une application. Notre approche se focalise sur l'introduction de vérifications de sécurité dans le code pour se protéger d'attaques comme les manipulations de données en entrée. Nous nous intéressons ensuite à l'automatisation de la mise en application de politiques de sécurité par des techniques de programmation. Nous avons par exemple automatisé l'application de règles de contrôle d'accès fines et distribuées dans des *web services* par l'instrumentation des mécanismes d'orchestration de la plate-forme. Nous avons aussi proposé des mécanismes permettant l'introduction d'un filtrage des données à caractère privée par le tissage d'aspects assisté par un expert en sécurité.

MOTS-CLEFS : securité, programmation orientée aspect, modularisation, architecture orientée service

ABSTRACT : Addressing security in the software development lifecycle still is an open issue today, especially in distributed software. Addressing security concerns requires a specific know-how, which means that security experts must collaborate with application programmers to develop secure software. Object-oriented and component-based development is commonly used to support collaborative development and to improve scalability and maintenance in software engineering. Unfortunately, those programming styles do not lend well to support collaborative development activities in this context, as security is a cross-cutting problem that breaks object or component modules. We investigated in this thesis several modularization techniques that address these issues.

We first introduce the use of aspect-oriented programming in order to support secure programming in a more automated fashion and to minimize the number of vulnerabilities in applications introduced at the development phase. Our approach especially focuses on the injection of security checks to protect from vulnerabilities like input manipulation. We then discuss how to automate the enforcement of security policies programmatically and modularly. We first focus on access control policies in web services, whose enforcement is achieved through the instrumentation of the orchestration mechanism. We then address the enforcement of privacy protection policies through the expert-assisted weaving of privacy filters into software. We finally propose a new type of aspect-oriented pointcut capturing the information flow in distributed software to unify the implementation of our different security modularization techniques.

KEY-WORDS : security, aspect-oriented programming, modularization, service-oriented architecture