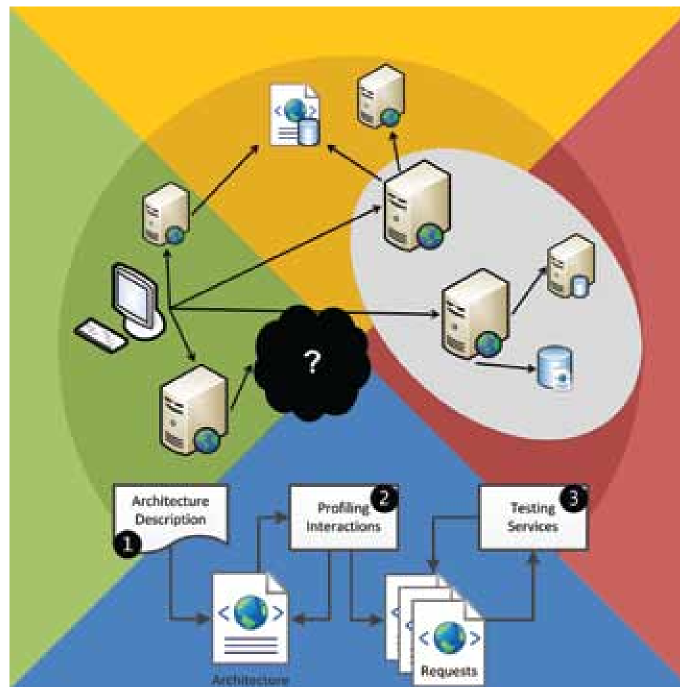


Nuno Antunes

Software Vulnerability Detection in Service-Based Infrastructures
Techniques and Tools

PhD Thesis in Information Science and Technology
supervised by Professor Marco Vieira and presented to the
Faculty of Sciences and Technology of the University of Coimbra

September 2013

Universidade de Coimbra

Software Vulnerability Detection in Service-Based Infrastructures

Techniques and Tools

Nuno Manuel dos Santos Antunes

Thesis submitted to the University of Coimbra
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
September 2013

Department of Informatics Engineering
Faculty of Sciences and Technology
University of Coimbra

This research has been developed as part of the requirements of the Doctoral Program in Information Science and Technology of the Faculty of Sciences and Technology of the University of Coimbra. This work is within the Dependable and Secure Systems specialization domain and was carried out in the Software and Systems Engineering Group of the Center for Informatics and Systems of the University of Coimbra (CISUC).

Funding for this work was partially provided by the Portuguese Research Agency *Fundação para a Ciência e Tecnologia* (FCT) through the scholarship SFRH/BD/65117/2009.

This work has been supervised by **Professor Marco Paulo Amorim Vieira**, Assistant Professor at the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

Ao Zé

Abstract

Service-based infrastructures consist of several software resources that interact to support (critical) business services of organizations. These resources are packaged as **services**, which are well-defined, self-contained, standard-based and protocol-independent modules providing business functionalities that are independent from the state or context of other services. These infrastructures typically support the implementation of Service Oriented Architectures (SOAs) and can be supported by different types of services and technologies, although Web Services are usually the implementation of choice.

Although software services should behave in a secure manner, they are often deployed with bugs that can be maliciously exploited. In fact, several studies show that, in general, web applications and services present dangerous flaws. Furthermore, the characteristics of service-based environments open the door to security challenges that must be handled properly, including services under the control of multiple providers and the dynamism of interactions and compositions.

To prevent security vulnerabilities, developers should apply best coding practices, perform security inspections, execute penetration tests, etc. However, many times, developers focus on the satisfying user's functional requirements and time-to-market constraints, disregarding security aspects. The problem is that software services are so exposed that hackers will most probably uncover any existing security vulnerability. Under this scenario, automated vulnerability detection techniques and tools play an extremely important role on helping deploying more secure service-based infrastructures, as they provide an easy and low cost way to detect software vulnerabilities.

This thesis addresses the problem of automated detection of software vulnerabilities in services and service-based infrastructures. First, the thesis proposes a framework defining the assumptions, the concepts, and the generic approaches that lay the basis for the development of innovative vulnerability detection techniques and tools. In practice, the framework defines a reference service-based infrastructure and proposes generic approaches for designing vulnerability detection tools for web services and for service-based environments.

The thesis also presents different techniques and tools to detect software vulnerabilities, designed following the approaches in the proposed framework. These include three new techniques to detect vulnerabilities in individual web services, each one addressing a different testing scenario and based on a different detection approach, namely: improved penetration testing, attack signatures and interface monitoring, and runtime anomaly detection. Built on top of such techniques, it is also proposed an integrated approach for security testing of service-based infrastructures, which is based on continuous monitoring to automatically

discover and test the existing services, resources and interactions, coping with the specificities of these dynamic and complex environments.

Finally, **the thesis proposes a generic approach for designing benchmarks that allow assessing and comparing vulnerability detection tools for service environments**. The approach specifies the components and the steps needed to implement concrete benchmarks, while focusing on two key metrics: precision and recall. It has been used to define two benchmarks, one supported by a predefined set of workload services and the other based on a set of services provided by the benchmark user. These benchmarks have been used to run several case studies to assess the vulnerability detection techniques proposed in the thesis, and to compare them to other existing tools, which at the same time allowed validating the benchmarking approach.

Keywords

Software services; service-based infrastructures; security vulnerabilities; vulnerability detection; testing; security assessment; benchmarking.

Resumo

As infraestruturas baseadas em serviços consistem em conjuntos de componentes de *software* que interagem entre si, utilizados de forma a suportar os processos de negócio (críticos) das organizações. Estes componentes, designados como **serviços**, são módulos bem definidos, auto contidos, baseados em standards e independentes de protocolos, que disponibilizam funcionalidades que são independentes do contexto ou estado de outros serviços. Estas infra-estruturas suportam a implementação de Arquiteturas Orientadas a Serviços (SOAs), e podem usar diferentes tipos de serviços e tecnologias, embora os serviços web sejam a forma de implementação mais comum.

Apesar de estes serviços possuírem requisitos de confiabilidade, vários estudos demonstram precisamente o contrário. De facto, estes estudos mostram que muitos dos serviços e aplicações web são disponibilizados com falhas graves que podem ser exploradas de forma maliciosa. Para além disso, as características inerentes aos ambientes baseados em serviços, incluindo serviços sob controlo de vários fornecedores de serviço assim como dinamismo na interação e composição de serviços, abrem a porta a novos desafios de segurança.

De forma a evitar vulnerabilidades de segurança, as equipas de desenvolvimento devem aplicar boas práticas de programação, levar a cabo inspeções de segurança, executar testes de penetração, entre outros. No entanto, muitas vezes estas equipas focam nos aspectos funcionais e na apresentação do produto dentro do tempo especificado, desprezando as preocupações de segurança. O problema é que estes serviços estão de tal modo expostos que qualquer brecha existentes acaba, muito provavelmente, por ser descoberta por potenciais atacantes. Nesse sentido, as técnicas e ferramentas de detecção automática de vulnerabilidades são extremamente importantes para facilitar a disponibilização de infraestruturas baseadas em serviços mais seguras, uma vez que proporcionam uma forma fácil e de baixo custo para detectar potenciais vulnerabilidades.

Esta tese ataca o problema da detecção automática de vulnerabilidades em serviços e infraestruturas baseadas em serviços. Primeiro, a tese define um enquadramento que inclui os pressupostos, conceitos e abordagens genéricas para o desenvolvimento de técnicas inovadoras para detecção de vulnerabilidades. Na prática, este enquadramento define uma infraestrutura baseada em serviços de referência e propõe abordagens genéricas para desenhar ferramentas de detecção de vulnerabilidades em serviços web e em ambientes baseados em serviços.

Para além disso, **a tese apresenta diferentes técnicas e ferramentas que permitem detetar vulnerabilidades de software,** desenhadas seguindo as abordagens mencionadas acima. Assim, foram desenvolvidas três novas técnicas de detecção de vulnerabilidades em serviços web, cada uma delas baseada numa abordagem

diferente e dirigida a um cenário de teste diferente, nomeadamente: testes de penetração, assinaturas de ataques e monitorização de interligações, e por fim detecção de anomalias em tempo de execução. Tirando partido destas técnicas, é ainda proposta uma abordagem integrada para testes de segurança em infraestruturas baseadas em serviços, a qual é baseada em monitorização contínua para descobrir e testar os serviços, recursos e interacções existentes, lidando com as especificidades destes ambientes dinâmicos e complexos.

Por fim, **esta tese propõe uma abordagem genérica para desenhar testes padronizados que permitam avaliar e comparar ferramentas de detecção de vulnerabilidades para ambientes baseados em serviços.** Esta abordagem especifica os componentes e os passos necessários para a implementação de testes concretos, tendo por base duas métricas chave: precisão e recuperação. Esta abordagem foi utilizada para definir dois testes padronizados: um baseado num conjunto pré-definido de serviços e outro baseado em serviços fornecidos pelo utilizador. Estes testes foram aplicados em vários casos de estudos, permitindo desse modo não só avaliar as técnicas de detecção de vulnerabilidades propostas na tese (estabelecendo uma comparação com outras ferramentas disponíveis no mercado), mas também validar a abordagem genérica para desenho de testes padronizados.

Palavras-chave:

Serviços de Software; infraestruturas baseadas em serviços; vulnerabilidades de segurança; detecção de vulnerabilidades; testes; avaliação de segurança; testes padronizados.

Acknowledgements

I would like to start by thanking to Professor Marco Vieira. Without his orientation, motivation and patience this work would never be possible. God knows how patient he was at times. The high level of quality and productivity that he demands from his students may turn the work into a very hard challenge, but it is also equally rewarding.

A word of appreciation to the members of the Software and Systems Engineering Group of the Centre of Informatics and Systems of the University of Coimbra for the excellent quality of the work they develop. Among these, I would like to distinguish the members of the SIGDep that regularly meet to discuss dependability related topics.

I also thank to all the anonymous reviewers for their comments that helped to improve the quality of the work developed.

I must also thank the help, the good humor, and the distractions of my laboratory colleagues. In fact, I think that my work would not be possible without the constant interruptions and distractions provided by my dear friends.

To all my friends, I thank for their support.

Special thanks go to Cristiana. In a not-so-long period of time, she became one of the cornerstones of my life. Let it last for a long, long time…

I would like to express my gratitude to my family. Especially to my parents, to whom I own more than what I can express for so many reasons, but above all, for the examples of hard work they gave to me.

Last but certainly not least, I must thank to my brother José Carlos that always believed in my capabilities and always encouraged me to pursuit a life of success and joy.

List of Publications

This thesis relies on the published scientific research presented in the following peer reviewed papers:

P 1. Nuno Antunes and Marco Vieira, *"SOA-Scanner: An Integrated Tool to Detect Vulnerabilities in Service-Based Infrastructures"*, *10th International Conference on Services Computing (SCC 2013)*, Santa Clara, CA, USA, 27 June - 2 July 2013, DOI:10.1109/SCC.2013.28.

P 2. Nuno Antunes and Marco Vieira, "Security Testing in SOAs: Techniques and Tools", in *Innovative technologies for dependable OTS-based critical systems*, Springer Milan, (Eds. Domenico Cotroneo), 2013, pp. 159-174, DOI:10.1007/978-88-470-2772-5_12.

P 3. Marco Vieira and Nuno Antunes, "Introduction to Software Security Concepts", in *Innovative technologies for dependable OTS-based critical systems*, Springer Milan, (Eds. Domenico Cotroneo), 2013, pp. 29-38, DOI:10.1007/978-88-470-2772-5_3.

P 4. Nuno Antunes and Marco Vieira, *"Defending against Web Application Vulnerabilities"*, IEEE Computer, vol. 45, no. 2, ISSN: 0018-9162, IEEE, pp. 66-72, February 2012, doi:10.1109/MC.2011.259.

P 5. Nuno Antunes and Marco Vieira, "*Evaluating and Improving Penetration Testing in Web Services*", 23rd IEEE International Symposium on Software Reliability Engineering (ISSRE 2012), Dallas, TX, USA, 27-30 November 2012, DOI:10.1109/ISSRE.2012.26.

P 6. Zoltán Micskei (**Budapest University of Technology and Economics,** Hungria) and Henrique Madeira and István Majzik (Budapest University of Technology and Economics, Hungria) and Alberto Avritzer (**Siemens Corporate Research,** EUA) Marco Vieira and Nuno Antunes, "Robustness Testing Techniques and Tools", in *Resilience Assessment and Evaluation: Past, Current and Future Trends*, Springer-Verlag Berlin Heidelberg, (Eds. Katinka Wolter, Alberto Avritzer, Marco Vieira, Aad van Moorsel), 2012, pp. 323–339, DOI:10.1007/978-3-642-29032-9_16.

P 7. Nuno Antunes and Marco Vieira, *"Enhancing Penetration Testing with Attack Signatures and Interface Monitoring for the Detection of Injection Vulnerabilities in Web Services"*, IEEE 8th International Conference on Services Computing (SCC 2011), Washington, D.C., USA: IEEE Press, ISBN: 978-1-4577-0863-3, 4-9 July 2011, DOI:10.1109/SCC.2011.67.

P 8. Nuno Antunes and Marco Vieira, "Detecting Vulnerabilities in Web Services: Can Developers Rely on Existing Tools?", in *Performance and Dependability in Service Computing: Concepts, Techniques and Research Directions*, ISBN: 978-1-609-60794-4, (Eds. Valeria Cardellini, Emiliano Casalicchio, Kalinka Castelo Branco, Julio Cezar Estrella, and Francisco José Monaco), IGI Global, June 2011, pp. 402–426, DOI:10.4018/978-1-60960-794-4.ch018.

P 9. Nuno Antunes and Marco Vieira, *"Benchmarking Vulnerability Detection Tools for Web Services"*, IEEE 8th International Conference on Web Services (ICWS 2010), Miami, Florida, USA: IEEE Computer Society, ISBN: 978-1-4244-8146-0, 5-10 July 2010,DOI:10.1109/ICWS.2010.76. **Received the Best Paper Award**

P 10. Nuno Antunes and Marco Vieira, *"Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services"*, IEEE 15th Pacific Rim International Symposium on Dependable Computing (PRDC'09), Shanghai, China: IEEE Press, ISBN: 978-0-7695-3849-5, 16-18 November 2009, DOI:10.1109/PRDC.2009.54.

P 11. Nuno Antunes, Nuno Laranjeiro, Marco Vieira, and Henrique Madeira, *"Effective Detection of SQL/XPath Injection Vulnerabilities in Web Services,"* IEEE International Conference on Services Computing (SCC 2009), 260-267, Bangalore, India: IEEE Computer Society, ISBN: 978-0-7695-3811-2, 21-25 September 2009, DOI:10.1109/SCC.2009.23.

P 12. Nuno Antunes and Marco Vieira, *"Detecting SQL Injection Vulnerabilities in Web Services",* Fourth Latin-American Symposium on Dependable Computing (LADC 2009), João Pessoa, Paraíba, Brazil: IEEE Press, ISBN: 978-1-4244-4678-0, 1-4 September 2009, DOI:10.1109/LADC.2009.21.

P 13. Marco Vieira, Nuno Antunes and Henrique Madeira, *"Using Web Security Scanners to Detect Vulnerabilities in Web Services",* 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009), Estoril, Lisbon, Portugal: IEEE Press, ISBN: 978-1-4244-4422-9, 29 June – 2 July 2009, DOI:10.1109/DSN.2009.5270294.

The following papers are related to this thesis but were not included:

P 14. Tânia Basso (**State University of Campinas (Unicamp)**, Brazil), Nuno Antunes, Regina Moraes (State University of Campinas (Unicamp), Brazil), and Marco Vieira, *"An XML-based Policy Model for Access Control in Web Applications"*, 24th International Conference on Database and Expert Systems Applications, Prague, Czech Republic, 26-29 August 2013, DOI:10.1007/978-3-642-40173-2_23.

P 15. Aniello Napolitano (**SESM S.c.a.r.l.**, Italy) and Gabriella Carrozza (SESM S.c.a.r.l., Italy) and Nuno Antunes and João Durães, "Survey on software faults injection in Java applications", in *Innovative technologies for dependable OTS-based critical systems*, Springer Milan, (Eds. Domenico Cotroneo), 2013, pp. 101–114, DOI:10.1007/978-88-470-2772-5_8.

P 16. Douglas Rodrigues (**University of São Paulo** - São Carlos, Brasil), Júlio Estrella (University of São Paulo - São Carlos, Brasil), Nuno Antunes, Francisco Mónaco (University of São Paulo - São Carlos, Brasil), Kalinka Branco (University of São Paulo - São Carlos, Brasil), Marco Vieira, "Engineering Secure Web Services", in *Performance and Dependability in Service Computing: Concepts, Techniques and Research Directions*, ISBN: 978-1-609-60794-4, (Eds. Valeria Cardellini, Emiliano Casalicchio, Kalinka Castelo Branco, Julio Cezar Estrella, and Francisco José Monaco), IGI Global, June 2011, pp. 360-380, DOI: 10.4018/978-1-60960-794-4.ch016.

Table of Contents

Chapter 1 Introduction 1
1.1 Detecting Vulnerabilities in Software Services 4
1.2 Main Contributions of the Thesis 7
1.3 Thesis Structure 11

Chapter 2 Background and Related Work 13
2.1 Software Services and Service-Based Infrastructures 14
2.2 Software Security 15
2.2.1 Threats and Vulnerabilities 17
2.2.2 Secure Coding 19
2.2.3 Security in the Development Process 21
2.3 Vulnerability Detection 23
2.3.1 Black-box Testing 23
2.3.2 White-box Analysis 28
2.3.3 Gray-box Testing 32
2.4 Assessment and Benchmarking 33
2.4.1 Workload Generation 34
2.4.2 Vulnerability and Attack Injection 34
2.4.3 Security Benchmarking 35
2.4.4 Runtime Monitoring and Testing 36
2.4.5 Assessment of Vulnerability Detection Tools 38
2.5 Conclusion 40

Chapter 3 Framework for the Detection of Vulnerabilities in Service-Based Infrastructures 41
3.1 Reference Service-Based Infrastructure 42
3.1.1 Specific Characteristics of Web Services 45
3.1.2 Challenges in Service-Based Infrastructures 46
3.1.3 Web Services Testing Scenarios 47
3.2 Designing Vulnerability Detection Tools for Web Services 48
3.2.1 Workload Emulator 51
3.2.2 Attack Emulator 52
3.2.3 Service Monitor 54
3.2.4 Vulnerability Detector 56
3.3 Integrated Approach for Vulnerability Detection 57
3.4 Conclusion 60

Chapter 4 Techniques for Detecting Injection Vulnerabilities in Web Services 63
4.1 Improved Penetration Testing [IPT-WS] 65
4.1.1 Workload Emulation 65

4.1.3 Attack Emulation 68
4.1.4 Vulnerability Detection 69
4.2 Attack Signatures and Interface Monitoring [Sign-WS] 72
4.2.1 Attack Emulation 73
4.2.2 Service Monitoring 80
4.2.3 Vulnerability Detection 81
4.3 Runtime Anomaly Detection [RAD-WS] 82
4.3.1 Workload Emulation 84
4.3.2 Service Monitoring 85
4.3.3 Vulnerability Detection 86
4.4 Conclusion 87

Chapter 5 Integrated Tool for Detecting Vulnerabilities in Service-Based Infrastructures 89
5.1 Architecture of the SOA-Scanner 90
5.2 Centralized Controller 93
5.2.1 Profiling Controller 94
5.2.2 Monitoring Controller 95
5.2.3 Testing Controller 96
5.3 Monitoring System 97
5.4 Testing Tools 99
5.5 Conclusion 101

Chapter 6 Benchmarking Vulnerability Detection Tools for Services 103
6.1 Generic Benchmarking Approach 104
6.1.1 Metrics 106
6.1.2 Workload 108
6.1.3 Procedure 109
6.2 Benchmark with a Predefined Workload [VDBenchWS-pd] 110
6.3 Benchmark with a User-Defined Workload [PTBenchWS-ud] 115
6.4 Benchmarking Properties 118
6.5 Conclusion 119

Chapter 7 Case Studies 121
7.1 Case Study #1: Assessing Public Web Services 122
7.1.1 Overall Results 123
7.1.2 False Positive Analysis 125
7.1.3 Coverage Analysis 129
7.1.4 Lessons Learned 130
7.2 Case Study #2: Using VDBenchWS-pd to Benchmark Vulnerability Detection Tools 132
7.2.1 Overall benchmarking results 133
7.2.2 Results for Tools that Report Vulnerable Inputs 135
7.2.3 Results for Tools that Report Vulnerable SQL Queries 138

7.2.4 Properties Discussion 140
7.3 Case Study #3: Using PTBenchWS-ud to Benchmark Penetration Testing Tools 144
7.3.1 Characterization of the workload 145
7.3.2 Benchmarking results 146
7.3.3 Comparison with the VDBenchWS-pd benchmark 147
7.3.4 Properties Discussion 148
7.4 Case Study #4: Detecting Vulnerabilities in a Service-Based Infrastructure 149
7.5 Conclusion 152
Chapter 8 Conclusion and Future Work 155
References 161

List of Figures

Figure 2.1 – SOAP Web Services Typical Structure. 15
Figure 2.2 – Simplified version of a Software Product lifecycle........ 22
Figure 3.1 – Reference Service-Based Infrastructure. 44
Figure 3.2 – Design of a web service vulnerability detection tool. 50
Figure 3.3 – Generic process for generating the attacks. 54
Figure 3.4 – Proposed generic testing procedure. 56
Figure 3.5 – Generic steps of the vulnerability detection approach. 58
Figure 4.1 – Overall design the improved penetration testing technique. 66
Figure 4.2 – Web service response analysis. 70
Figure 4.3 – Overall design of the Sign-WS technique. 74
Figure 4.4 – Examples of queries with signatures. 75
Figure 4.5 – Signature token used. (a) regular token; (b) reversed token. 76
Figure 4.6 – Mechanism to intercept and sign external requests. 79
Figure 4.7 – Examples of command processing steps with SQL queries. 81
Figure 4.8 – Overall design of the RAD-WS technique. 83
Figure 4.9 – Process to remove the variant parts of an SQL query. 84
Figure 4.10 – Example of SQL commands execution. 85
Figure 4.11 – The AOP Based Service Monitoring system. 86
Figure 5.1 – Design of SOA-Scanner. 92
Figure 5.2 – SOA-Scanner components in the infrastructure. 93
Figure 5.3 – Schema of the XML format used to store the architecture. 96
Figure 5.4 – Example of an Aspect in Java. 98
Figure 5.5 – Process to Integrate the Reports of multiple tools. 101
Figure 7.1 – Interception of SQL Injection vulnerabilities reported. 125
Figure 7.2 – False positives for SQL Injection in the public services. 127
Figure 7.3 – Interception of SQL Injection vulnerabilities without False Positives... 129
Figure 7.4 – Vulnerabilities detected distributed per type. 131
Figure 7.5 – VDBenchWS-pd results for pen. testing and Sign-WS. 135
Figure 7.6 – Intersection of vulnerabilities detected. 137
Figure 7.7 – VDBenchWS-pd results for static analysis and RAD-WS. 139
Figure 7.8 – Intersections for static analysis and RAD-WS. 140
Figure 7.9 – PTBenchWS-ud results for the penetration testing. 147
Figure 7.10 – Architecture for the Case study #4. 150
Figure 7.11 – GUI for classifying newly found services. 151

List of Tables

Table 3.1 – Examples of SQL/XPath Injection attack types. 53
Table 4.1 – Example of the domain specification for each parameter. 67
Table 4.2 – Rules for the analysis of opposite responses. 72
Table 4.3 – Examples of signed SQL Injection attack types. 77
Table 5.1 – Correlation between testing techniques and scenarios. 100
Table 6.1 – Vulnerabilities found in the workload services. 113
Table 6.2 – Final numbers of vulnerabilities in the workload servicers. 114
Table 6.3 – Example of coverage and false positive rates estimation. 117
Table 7.1 – Overall results obtained. 124
Table 7.2 – False positives for other vulnerability types. 128
Table 7.3 – Coverage for SQL Injection vulnerabilities. 130
Table 7.4 – Tools under benchmarking. 132
Table 7.5 – VDBenchWS-pd Benchmarking results. 134
Table 7.6 – VDBenchWS-pd tools ranking. 134
Table 7.7 – Third-party web services characterization. 141
Table 7.8 – Results for third-party web services. 142
Table 7.9 – Ranking based on third-party services. 142
Table 7.10 – VDBenchWS-pd repeatability results. 143
Table 7.11 – Tools under benchmarking. 144
Table 7.12 – Workload vulnerabilities as reported by Sign-WS. 145
Table 7.13 – PTBenchWS-ud benchmarking results. 146
Table 7.14 – PTBenchWS-ud tools ranking. 146
Table 7.15 – Results for both benchmarks. 148
Table 7.16 – Results for SOA-Scanner vulnerability detection. 152

Chapter 1
Introduction

A **Service-Based Software Infrastructure**[1] consists of several software resources working together to support the information infrastructure of one or more organizations (Bennett et al. 2000). These software resources allow the interaction between consumers and providers and are packaged as services. **Software services**[2] must be autonomous and self-contained, coarse-grained and loosely coupled, thus independent from the state or context of other services (Papazoglou and Heuvel 2007). This way they can be reused to implement different business processes, while the reduced dependencies allow replacing and/or modifying a service without the need for changing other components of the infrastructure.

In a dynamic and competitive business world, organizations need to adapt quickly and efficiently to new challenges and opportunities. Hence, it is necessary to simplify the Information Technology (IT) infrastructure and improve the interoperability and business agility of the organization. Service-based infrastructures provide the ground for satisfying such demands, being typically used to support the well-known **Service Oriented Architectures (SOAs)**[3] (Erl 2005). In a SOA context, a service is a function offered by a provider that allows consumers to

---

[1] A **Service-Based Software Infrastructure** is an infrastructure based on the interaction of several software services with the objective of supporting the information infrastructure of one or more organizations (Bennett et al. 2000).

[2] A **Software Service** is a well-defined, self-contained and reusable component that implements a business functionality that is delivered to other applications through a standard-based interface in a protocol-independent environment (Papazoglou and Heuvel 2007).

[3] A **Service Oriented Architecture (SOA)** is a paradigm for organizing and utilizing distributed functionalities that may be under the control of different ownerships, following an architectural style whose main emphasis is on the loose coupling among interacting software agents or components (Perrey and Lycett 2003; MacKenzie et al. 2006).

achieve some desired outcome, with both of these roles being played by software agents on behalf of their owners. Service-orientation provides means for separation of concerns, taking advantage of the characteristics of software services to allow multiple functionalities without adding design complexity or increasing communication principles (Papazoglou and Heuvel 2007), and is nowadays used in a wide range of organizations and scenarios, including business-critical systems. Frequently, services are connected through a Enterprise Service Bus (ESB) (Keen et al. 2004), whose purpose is to connect and coordinate, in a reliable manner, the interaction among services across extended enterprises (David A. Chappell 2009). Although SOAs can be implemented using different types of software services and technologies, Web Services are usually the implementation of choice (Singhal, Winograd, and Scarfone 2007).

Web Services (WS) [4] are thus a strategic mean for data exchange, content distribution, and systems integration. Web services are supported by a complex software infrastructure, which typically includes an application server, the operating system, and a set of external systems (e.g. other services, databases, and payment gateways). In practice, a web service provides a simple interface between the consumers and the provider, based on the exchange of standardized messages over the network using the HTTP or HTTPS protocols (D. A Chappell and Jewell 2002a). The web service and the format of the messages to be exchanged are typically described in a definitions file.

Nowadays, there are two main classes of web services: SOAP and RESTful. SOAP web services (D. A Chappell and Jewell 2002a) are session-less and follow the XML-based Simple Object Access Protocol, after which the technology was originally named, and use a WSDL (Web Services Definition Language) file to describe the interface and the format of the messages to be exchanged. On the other hand, RESTful (REpresentational State Transfer) web services reuse the HTTP methods (e.g. *GET*, *POST*, *DELETE*) together with a simplified message format, and a WADL (Web Application Description Language) file may optionally be used to describe the interface (Richardson and Ruby 2007). While SOAP web services are interoperability oriented thus more platform independent, RESTful web services rely on a much lighter infrastructure, being much more suited for simpler and ad hoc integration scenarios (Pautasso, Zimmermann, and Leymann 2008).

For supporting **Business-Critical Systems**, SOAs and software services must be dependable and secure. However, several studies show that, in general, web-based applications present dangerous faults (OWASP Foundation 2013; Acunetix 2007) and services are no exception. In fact, previous works demonstrate that web services are

[4] A **Web Service (WS)** is a self-describing software component designed to support machine-to-machine interaction based on messages exchange, over a network and using the HTTP protocol together with Web-related standards (W3C 2004).

frequently deployed with security vulnerabilities (Vieira, Antunes, and Madeira 2009; Lowis and Accorsi 2009). The same is true for other types of software services, including messaging middleware, for which several robustness and critical security related problems have been disclosed in the past (Laranjeiro, Vieira, and Madeira 2008)

Security Vulnerabilities are a particular type of software fault that open the door for attackers to unduly access a system or network (Christey and Martin 2006), leaving space for the system/data exploitation and/or corruption. Existing studies show that injection vulnerabilities are among the most common and dangerous vulnerabilities in the Web (OWASP Foundation 2013). Injection attacks try to take advantage of improperly coded applications to execute the commands specified by the attacker, enabling, for instance, access to critical data and resources (OWASP Foundation 2013). These types of vulnerabilities are particularly relevant in software services (Vieira, Antunes, and Madeira 2009), as these frequently use a data persistence solution over a relational (Ramakrishnan and Gehrke 2003) or a XML database (Meier 2003).

Due to hard time-to-market requirements and other limitations, the security of web applications is frequently disregarded. In particular during the evolution of such applications (in terms of features and complexity), this problem tends to increase, as security is not one of the main concerns in the development lifecycle. Even when security is a concern, it is typically addressed from the network and operating system points of view. In such cases, assessment teams rely on automated tools to help finding breaches in the operating system and/or the network, ignoring that today's priorities are different (Curphey et al. 2002) as, being the publicly exposed face of an organization, web applications and services became the preferred targets for hackers. In fact, hackers moved their focus from the network to applications' code, searching for vulnerabilities by exploiting the inputs of applications with specially tampered values. These application level attacks are performed through network ports that are used for regular web traffic and thus cannot be mitigated by traditional security mechanisms such as firewalls and network intrusion detection systems (Singhal, Winograd, and Scarfone 2007).

To deploy software services without security vulnerabilities developers must follow a **defense-in-depth** approach (Howard and Leblanc 2002; Curphey et al. 2002). This approach assumes that any security precaution can fail and so, security depends on several layers of mechanisms that cover the failures of each other. Software engineering teams are expected to apply the effort needed to introduce adequate security precautions in a way that minimizes the probability of successful attacks. In practice, this means that they must address security in all the phases of the software product's development lifecycle (ranging from requirements elicitation to testing and deployment), which includes applying best coding practices, perform security inspections, execute penetration tests, deploy runtime attack detection systems, etc. (Antunes and Vieira 2012a). However, it is in the testing phase that the software

should be tested and verified in an effort not only to assure that the system fulfills the intended functional requirements, but also to detect and remove any existing security vulnerabilities. In fact, testing represents the last opportunity to prevent applications from being deployed with security flaws.

This work focuses on the **testing phase** of the development lifecycle (in particular from the automated security testing perspective) and *proposes techniques and tools for the detection of injection vulnerabilities in service-based software infrastructures*. In short, the motivation is threefold: 1) many times, developers focus on the satisfying user's functional requirements and time-to-market constraints, disregarding security aspects; 2) similarly to other web applications, web services are so exposed that hackers will most probably uncover any existing security vulnerability; and 3) in a service-based scenario, injection attacks are the most dangerous and common ones. **Automated vulnerability detection techniques and tools** thus play an extremely important role on helping the developers to produce non-vulnerable code while improving productivity, as they provide an easy and less expensive way for testing applications, without requiring the availability of human resources specifically specialized in computer security.

1.1 Detecting Vulnerabilities in Software Services

Many different techniques for the detection of software vulnerabilities have been proposed in the past (Stuttard and Pinto 2007). These are usually divided into white-box analysis, which consists in the analysis of the code of the application without executing it, and black-box testing that analyzes the execution of the application without accessing its internals (i.e. based on the outputs of the application). Exceptionally, other techniques, normally referred as gray-box, may combine black-box and white-box characteristics. All these approaches can be applied manually or automatically with the help of tools.

White-box approaches analyze of the program from an internal point-of-view and include code inspection, reviews, walkthroughs, etc. In a security code inspection the programmer delivers the code to his peers and they systematically examine it in a formal meeting, searching for security vulnerabilities. This is regarded as the most effective way for assuring that a piece of software has a minimum number of vulnerabilities (Curphey et al. 2002), but it is usually very time consuming and expensive. Less expensive alternatives are code reviews and code walkthroughs (Freedman and Weinberg 2000). Code reviews are a simplified version of code inspections that are still formal, but do not require a formal review meeting. Code walkthroughs are an informal approach that consists of manually analyzing the code by following the code paths as determined by predefined input conditions. However, these techniques still bring the cost of having more than one expert manually analyzing the code.

The alternative to reduce the cost of white-box analysis is to rely on automated tools, such as static code analyzers. In fact, the use of these tools is seen as an easier and faster way to find bugs and vulnerabilities. Static code analysis tools vet the code in an attempt to identify common implementation-level faults (Stuttard and Pinto 2007). The analysis performed varies depending on the tool sophistication. The main problem is that exhaustive source code analysis may be difficult and may not find many security flaws due to the complexity of the code and the lack of a dynamic (runtime) view.

Black-box approaches analyze the execution of the program from an external point-of-view. Testing is the most used technique for verification and validation of software and consists in executing the software and comparing the outcome with the expected result (Myers, Sandler, and Badgett 2011). There are several levels for applying black-box testing, ranging from unit testing to integration testing and system testing. The tests specification defines the coverage criteria and should be elaborated before development. The idea is that the test specification should help developers during the coding process. Furthermore, by designing tests a priori, it is possible to avoid biasing the tests due to knowledge about the code developed.

Robustness testing is a specific form of black-box testing. The goal is to characterize the behavior of a system in the presence of erroneous input conditions (Koopman and DeVale 2000; Vieira, Laranjeiro, and Madeira 2007). **Penetration testing**, by its turn, is a specialization of robustness testing and consists of the analysis of the program execution in the presence of malicious inputs, searching for potential vulnerabilities (Stuttard and Pinto 2007). In practice, the tester needs no knowledge about the implementation details and uses fuzzing techniques to test the inputs of the application from the malicious user's point of view (Stuttard and Pinto 2007). The problem is that the number of tests can reach hundreds or even thousands for each vulnerability type, representing a very repetitive, tedious and, essentially, expensive task if performed manually. Penetration testing tools allow reducing this cost by providing the required support for searching for vulnerabilities in an automatic way. The most common penetration testing tools used in web applications are generally referred to as web security scanners (or web vulnerability scanners).

Previous research and practice show that both white-box and black-box state of the art **vulnerability detection tools have a very limited effectiveness** (Fonseca, Vieira, and Madeira 2007; Wagner et al. 2005; Teixeira, Antunes, and Neves 2007). In the particular context of web services, studies show that static code analysis and penetration testing tools present very high false positive rates, which reduces the confidence on the precision of the vulnerabilities detected (Vieira, Antunes, and Madeira 2009; Antunes and Vieira 2009a). Those studies also demonstrate that the coverage of penetration testing tools is very low, suggesting that many vulnerabilities may remain undetected. Another key observation is that, even when implementing the same approach, different tools frequently report distinct

vulnerabilities for the same piece of code, generating conflicting results that again, reduce the confidence in the output of those tools.

The problem is that penetration testing is the technique most used by web developers to detect vulnerabilities in their applications (Stuttard and Pinto 2007). In the context of software services and infrastructures, where services are deployed, interconnected and updated at anytime, being often composed by third-party components, penetration testing assumes an even bigger importance as many times it is necessary to assess the security of services that are under the control of external entities. In this context, the effectiveness problems highlighted before clearly create the need for new and more efficient tools that allow testing services in different scenarios.

As a black-box technique, penetration testing has no visibility on the internal behavior of the tested services. In fact, it can only observe the application from the point of view of an external user, which results directly into two major restrictions:

- The vulnerability detection process must be based only on the analysis of the output of the web service (leading to a lack of information for decision making). This limits the capabilities of the tools, as most times the information that is released to the client is not enough to effectively detect vulnerabilities. Also, many times the output of the application is preprocessed to avoid the leakage of information about the system, making it almost impossible to identify any vulnerability (although they may exist and the testing tool may effectively exploit them).
- It is impossible for the tool to know what inputs to use in order to maximize the number of web service's code paths that are tested (resulting in inadequate code coverage). Obviously, if some paths of code are not executed during the testing process, obviously the vulnerabilities located in these pieces of code will not be detected.

In addition to these restrictions, the specific characteristics of service-based infrastructures raise **new challenges** for security testing. First of all, these infrastructures are dynamic in nature, facing (runtime) changes in the services used and in the way they interact. Second, they usually include services that are under the control of multiple providers, creating the need for diverse vulnerability detection tools that can cope with different kinds of information available (e.g. the source code may be available or not). Finally, the security (or lack of it) of a given service can impact the services and resources of the infrastructure with which it interacts, creating the need for considering the interactions with resources and other services.

Another major challenge is the ability to **select the most effective vulnerability detection tools and configurations** from a set of alternatives available. This is particularly relevant as different tools may generate different (or even conflicting) results. However, existing evaluations are limited by the small number of tools

assessed and by the representativeness of the experiments (Vieira, Antunes, and Madeira 2009; Antunes and Vieira 2009a; Fonseca, Vieira, and Madeira 2007; Wagner et al. 2005), thus not allowing the generalization of the results. This way, web services developers frequently select the tools based on common sense, which may lead to wrong decisions and ultimately to code deployed with vulnerabilities, thus calling for techniques that allows assessing and comparing such tools under realistic conditions.

In summary, taking into account the intrinsic characteristics of service-based infrastructures, the importance and widely usage of automated vulnerability detection, and the low effectiveness of existing tools, this thesis focus on addressing the following needs:

1. An integrated approach able to continuously monitor, discover and testing the services in the service-based infrastructure, coping with the dynamicity of these environments;
2. Effective vulnerability detection techniques and tools that can be used to test different services, under different scenarios, and with different levels of access and information available;
3. Adequate benchmarking approaches that allow evaluating and comparing vulnerability detection tools, thus helping providers and consumers to select the most effective ones and guiding the design and development of new tools.

1.2 Main Contributions of the Thesis

The main contribution of this thesis is an **integrated approach that allows continuously discovering the services and resources of a service-based infrastructure, and supports the process of testing those services for injection vulnerabilities** with improved effectiveness by using the testing technique most adequate to each scenario, depending on the type of service to be tested and on the information that is available. In detail, the contributions of this thesis are:

- The definition of a **framework for the detection the vulnerabilities in service-based infrastructures**, defining the assumptions, the concepts, and the generic approaches that lay the basis for the development of innovative techniques and tools. This framework includes a reference service-based infrastructure and generic approaches for designing vulnerability detection tools for web services and service-based environments.
- The proposal of **a generic approach for designing vulnerability detection tools** for web services, which includes the definition of the testing procedure

and of the tool components. Tools designed based on this approach should be able to detect a broad range of vulnerabilities, with priority to injection vulnerabilities, the most dangerous and common in the service-based context. Based on this generic approach, three new techniques to detect vulnerabilities in web services are proposed:

- An **improved penetration testing approach to detect SQL Injection vulnerabilities [IPT-WS]** (Antunes and Vieira 2009b). The approach uses representative workloads to exercise the web services, implements effective attackloads, and applies well-defined rules to analyze the web services responses, thus improving detection coverage while reducing false positives. The implemented prototype has shown to be, in several cases, more effective than existing commercial security scanners.
- An **approach based on attack signatures and interface monitoring to detect injection vulnerabilities [Sign-WS]** (Antunes and Vieira 2011). The approach overcomes the visibility limitations of penetration testing by introducing special tokens inside the injection attacks (signatures) and then monitoring the interfaces of the service under testing looking for these tokens to detect vulnerabilities. The implemented prototype is able to largely outperform existing penetration testing tools in terms of vulnerability detection coverage and false positives. Also, comparing to IPT-WS, the tool is able to detect more vulnerabilities, while reducing false positives to zero.
- A **runtime anomaly detection approach able to detect SQL Injection and XPath Injection vulnerabilities [RAD-WS]** (Antunes et al. 2009a). The approach exercises the service for profiling its regular internal behavior (learning phase) and then attacks the service (attacking phase), reporting a vulnerability when some deviation is detected. The implemented prototype has shown to be able to consistently outperform the aforementioned approaches and other existing tools, being able to achieve a higher detection coverage, while avoiding false positives.

• The proposal of **an integrated approach for security testing of service-based infrastructures [SOA-Scanner]** (Antunes and Vieira 2013). This approach is based on continuous monitoring to automatically discover and test the existing services, resources and interactions, coping with the dynamicity of these environments. This includes:

- The **design of a generic architecture** based on three key generic steps: architecture description, profiling interactions, and testing services. It includes the design of the main components of the approach, namely: an integrated controller, a testing service, and a set of probes to be deployed in the infrastructure. It also includes the definition of guidelines on how to integrate different vulnerability detection techniques and tools in the testing service.
- The implementation of a **prototype focused on web services and injection vulnerabilities** (Antunes and Vieira 2013). The prototype uses probes that are deployed in the infrastructure for monitoring the interactions among the services and resources of the infrastructure and then applies the vulnerabilities detection technique (or techniques) most adequate considering the level of access and information available about each web service. The goal is to achieve maximum effectiveness, both in terms of vulnerability coverage and false positives.

- The proposal of a generic **approach for designing benchmarks for vulnerability detection tools for services** (Antunes and Vieira 2010). The approach specifies the requirements for the benchmark components (i.e. workload, metrics and procedure) and the steps needed to implement concrete benchmarks. This approach focuses on two key metrics: precision (the ratio of correctly detected vulnerabilities to the number of all reported vulnerabilities) and recall (the ratio of correctly detected vulnerabilities to the number of known vulnerabilities). It has been used to define two concrete benchmarks:

  - A **benchmark based on a predefined workload targeting tools able to detect SQL Injection vulnerabilities in web services [VDBenchWS-pd]** (Antunes and Vieira 2010). This benchmark is based on a well defined and large set of web services adapted from standard performance benchmarks, and includes both vulnerable and non-vulnerable versions of the services. The main limitation of this benchmark is that, although based on a well-defined set of rules, it is not fully protected against "*gaming*" (i.e. adaptations/tuning that allow producing optimistic or biased results). In fact, as the set of web services is well known, vendors can easily tune their tools to maximum effectiveness in the context of the benchmark, while failing in different scenarios.
  - A **benchmark based on a user-provided workload (any set of services) targeting penetration testing tools for the detection of**

injection vulnerabilities in web services [PTBenchWS-ud] (Antunes and Vieira 2012b). This benchmark follows an alternative approach, solving the "*gaming*" problem by allowing the benchmark user to specify the workload (i.e. the target set of web services is not predefined and is unknown to the tools' providers) that best represents his specific development conditions, providing at the same time more realistic results. To support the (user) task of defining the workload, the benchmark includes a procedure and a tool to help characterizing the injection vulnerabilities that exist in the web services (and that serve as reference for the metrics calculation), thus avoiding the need for conducting such analysis manually.

- The execution of **multiple campaigns to experimentally evaluate and compare vulnerability detection tools**, including the ones proposed in this thesis and other well-known and widely used tools. This includes:

  - The evaluation of existing penetration testing tools in public web services (Vieira, Antunes, and Madeira 2009). This allows understanding the most frequent vulnerabilities and also the effectiveness and limitations of existing tools (and what needs to be improved). The results highlight the limitations of penetration testing as different tools reported different vulnerabilities and presented low detection coverage (less than 20% for two of the scanners), and high false positives rates (35% and 40% in two cases).
  - The **use of VDBenchWS-pd to evaluate different techniques and tools**, including the tools defined in this work (Antunes and Vieira 2010). Besides helping on improving the proposed methodologies, the goal is to validate the benchmarking approach. In practice, the benchmark has been used to compare several tools, including commercial and open-source penetration testers and static code analyzers. The results showed that the benchmark allows ranking the different tools according to the different measures, providing the support needed for the users to select the most effective tool under diverse requirements. Also, results demonstrated that the vulnerability detection approaches proposed in this work consistently provide better results than other state-of-the-art tools.

- The **use of PTBenchWS-ud to evaluate different penetration testing tools** (Antunes and Vieira 2012b). The goal is to show that the proposed benchmarking approach can be used to assess and compare this type of tools in the specific context of the services of an organization (i.e. under specific

workload conditions). Comparing to VDBenchWS-pd benchmark, similar results were obtained as, although some of the detailed measures differed, both benchmarks leaded to the same ranking of the tools.

- The **application of the SOA-Scanner tool in a case study based on a subset of the jSeduite SOA** (Antunes and Vieira 2013). The case study consists of a simplified service based infrastructure that uses the code of the **jSeduite SOA** (Delerce-Mauris et al. 2009). This allows demonstrating all the different testing scenarios and the functionalities of the integrated tool in a simple infrastructure, validating this way its capabilities.

Although it is possible to use some of the techniques proposed in this thesis beyond the testing phase (i.e. at runtime), such usage raises problems related to service degradation and failure propagation due to the execution of testing activities on production services. To tackle these problems, there are other works on techniques that provide partial solutions, such as sandboxing, virtualization, etc. (Michelsen and English 2012). Although this is a very important challenge, these concerns are **out of the scope** of this work and so, they will **not to be addressed** in this thesis.

1.3 Thesis Structure

This first chapter introduced the problem addressed and the main contributions of the thesis.

Chapter 2 presents background on web services, service based infrastructures, and SOAs. It also discusses related work on vulnerability detection techniques, with focus on techniques that apply to web applications and services, as well as on tools for monitoring and testing service based infrastructures. Finally, it discusses existing works on assessment and comparison of vulnerability detection tools, focusing particularly on benchmarking.

Chapter 3 presents the framework for the detection of vulnerabilities in Service-Based Infrastructures. The chapter starts by establishing an infrastructure that is used as reference throughout the work and the challenges and requirements of such kind of infrastructure. Furthermore, it presents a generic approach for designing vulnerability detection tools and the generic integrated approach for testing Service-Based Infrastructures for vulnerabilities.

Chapter 4 presents the vulnerability detection techniques developed based on the generic approach introduced in Chapter 3. It presents IPT-WS: a penetration testing technique able to detect SQL Injection vulnerabilities in web services, Sign-WS: a technique based on attack signatures and interface monitoring for the detection of

injection vulnerabilities, and RAD-WS: a runtime anomaly detection technique able to detect SQL Injection and XPath Injection vulnerabilities in web services.

Chapter 5 presents the integrated tool for security testing for service-based infrastructures. The tool implements the approach proposed in Chapter 3 and uses interface monitoring to continuously monitor and discover the services and resources of the infrastructure. For testing the services, the approach makes use of the vulnerability detection techniques presented in Chapter 4.

Chapter 6 presents a generic methodology for designing benchmarking approaches to evaluate vulnerability detection tools for web services. It also presents two instantiations of this methodology: VDBenchWS-pd, a benchmark based on a predefined workload targeting tools able to detect SQL Injection vulnerabilities in web services; and PTBenchWS-ud, a benchmark based on the use of any workload and targeting penetration testing tools able to detect injection vulnerabilities in web services.

Chapter 7 presents the case studies developed to show the practical application and to evaluate the proposed techniques and tools. The first case study presents an evaluation of commercial web security scanners using public web services. The second demonstrates the use of the VDBenchWS-pd benchmark to evaluate and compare a large set of vulnerability detection tools, including tools implementing the techniques presented in Chapter 4. The third case study demonstrates the VDBenchWS-pd benchmark by evaluating and comparing four penetration testing tools. The final case study demonstrates the capabilities of SOA-Scanner in the detection of vulnerabilities in a simple service-based infrastructure.

The last chapter concludes the thesis and proposes topics for future research.

Chapter 2
Background and Related Work

The research on software services has been a hot topic in the last decade, as shown by the increasing number of publications in this area. On the other hand, research related to security concerns dates way back before the concept of software services even existed, with exploratory research on defenses against malicious faults, i.e. security threats, starting in the mid-80s (Dobson and Randell 1986). The conjunction of both concerns is a key topic and, although research in web security and threats has also been a major topic in the last few years, in most cases the existing works do not deal with the specificities of services environment (Curphey et al. 2002; W. G. Halfond, Viegas, and Orso 2006; Bau et al. 2010; Fonseca, Vieira, and Madeira 2009).

This chapter presents background on software services and service-based infrastructures, software security threats, and concepts and the best practices regarding the software development process. It also includes an overview of the most relevant related work on vulnerability detection and on the assessment and benchmarking of existing tools. It is important to emphasize that some of the works presented are relative to web applications in general, and not to services. However, although few works have focused the problem of security and vulnerability detection in the web services environment, the works presented here contain important ideas that should be taken into account when researching new techniques.

The structure of this chapter is as follows. The next section introduces the basic concepts on software services and service-based infrastructures. Section 2.2 presents the background on software security, including security threats and how to deal with them. Section 2.3 reviews the related work on vulnerability detection techniques, while Section 2.4 reviews the related work on assessment and benchmarking approaches. Section 2.5 concludes the chapter.

2.1 Software Services and Service-Based Infrastructures

Service-based infrastructures consist of several software resources that interact to support (critical) business services of organizations. These resources are packaged as **software services**, which allow the interaction between consumers and providers (via the exchange of messages based on standards). Essentially, services are reusable components that represent business functionalities delivered in an efficient way within a protocol-independent distributed environment and through a standardized interface. Also, services are coarse-grained and loosely coupled and are designed to interact without the need for dependencies between services, supporting multiple functionalities without adding design complexity or increasing communication.

Service-based infrastructures typically support the implementation of Service Oriented Architectures (SOAs), which are no more than a paradigm for taking advantage of distributed functionalities that may be under the control of different owners, following an loosely coupled architectural style (Perrey and Lycett 2003; MacKenzie et al. 2006).

*"**'You don't need Web services to build SOA!'** These are words you'll hear many say prior to explaining service-oriented architecture. However, this statement is typically followed by something equivalent to **'...but using Web services to build SOA is a darn good idea...'**"* (Erl 2005).

A web service is a piece of business logic available on a network (usually Internet) and accessible through an Internet protocol (such as HTTP or HTTPS) by the exchange of messages according to the definition of the interface of the service, that is usually described using a machine readable format (D. A Chappell and Jewell 2002b; Sandoval, Roussev, and Wallace 2009; Christensen et al. 2001). **The service provides a set of operations** and, in practice, each operation is a method with several input parameters. In each interaction the consumer (client) sends a request message to the provider (server). After processing the request, the server sends a response message to the client with the results. To facilitate the discovery of Web services, brokers are usually used.

A web service is supported by a complex software infrastructure, which typically includes an application server, the operating system and external systems such as data sources (DBMS, XML databases, etc.) or other web services. Figure 2.1 presents a simplified view of the typical structure of a web services system. Web services can be implemented in a wide variety of architectures and technologies and can interoperate with other technologies and software design approaches, allowing an evolutionary adoption that does not require major transformations to legacy applications (Singhal, Winograd, and Scarfone 2007).

Actually, this interoperability is one of the main causes for web services adoption, which was also fostered by the advent of Service Oriented Architectures (SOA). As with any new technology, this success comes with a level of increased risk

(Lindstrom 2004). In fact, "*web services are a technology that can be used to implement Service Oriented Architectures (SOA) and are increasingly becoming the SOA implementation of choice. For a SOA to truly meet its goals, applications developed must be secure and reliable.*" (Singhal, Winograd, and Scarfone 2007).

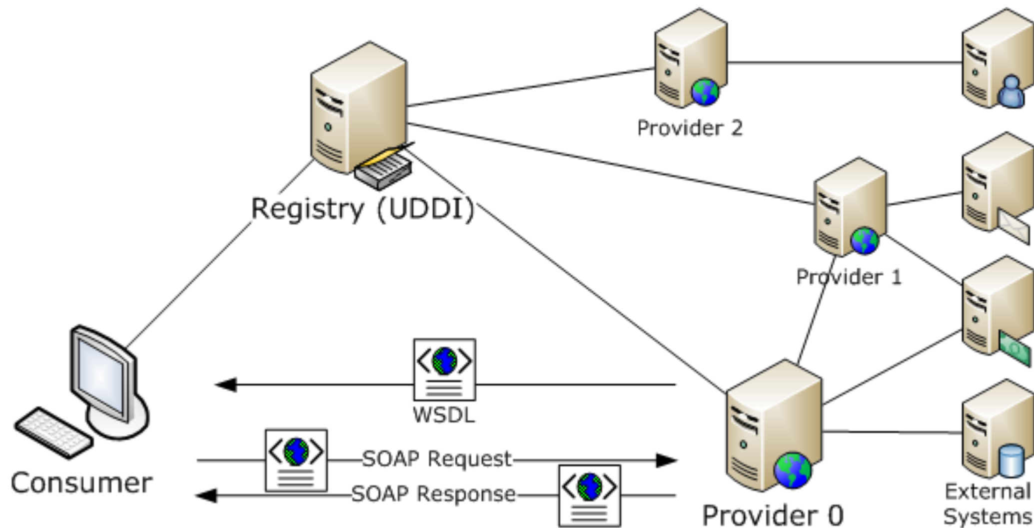


Figure 2.1 – SOAP Web Services Typical Structure.
A registry is used to discover the services and then the interactions are performed using SOAP messages that follow the specification in the WSDL file.

2.2 Software Security

Security, "*the practice of building software to be secure and function properly under intentional malicious attacks*" (G. McGraw 2006), is an integrative concept that includes four key properties (Cachin et al. 2000): confidentiality (absence of unauthorized disclosure of a service or piece of information), authenticity (guarantees that a service or piece of information is authentic), integrity (protection of a service or piece of information against illicit and/or undetected modification), and availability (protection against possible denials of service caused maliciously). To achieve these properties, several security mechanisms have been developed in the past, targeting especially subsystems such as operating systems, database management systems, and web servers. These mechanisms can be classified as follows (Cachin et al. 2000):

- **Secure channels and envelops**: mechanisms that provide communication in a secure way (e.g. encryption, TLS, SSL). The information is transmitted thought the network using secure channels or encapsulated in envelops.

- **Authentication**: mechanisms that assure that the data accessed by the users is authentic. (e.g. HTTP authentication, IPSec)
- **Protection and authorization**: mechanisms that protect resources and data from unauthorized access and guarantee that users only do what they are authorized to do (e.g. login/password checking, privilege management).
- **Auditing and intrusion detection**: these mechanisms allow a posteriori analysis of the accesses to resources and data, allowing the detection of unauthorized accesses or anomalous usage (e.g. auditing systems, intrusion detection systems, web applications firewalls).

In practice, the goal of security is to protect systems and data from intrusion. The risk of intrusion is related to the system vulnerabilities and the potential security attacks. The **system vulnerabilities** are an internal factor related to the set of security mechanisms available (or not available) in the system, the correct configuration of those mechanisms, and the hidden flaws on the system implementation. Many types of vulnerabilities and taxonomies to classify them currently exist (Stuttard and Pinto 2007). Vulnerability prevention consists on guarantying that the software has the minimum number of vulnerabilities possible. On the other hand, as the effectiveness of the security mechanisms depend on their correct configuration, the system administrator must correctly configure the security mechanisms by following administration best practices. Vulnerability removal consists on reducing the vulnerabilities found in the system. For example, the system administrator must pay attention to the new security patches release by software vendors and install those patches as soon as possible. Furthermore, any configuration problems detected on the security mechanisms must be immediately corrected.

Security attacks are an external factor that mainly depends on the intentionality and capability of humans to maliciously break into the system tacking advantage of vulnerabilities. In fact, the success of a security attack depends on the vulnerabilities of the system and attacks are harmless in a system without vulnerabilities. On the other hand, vulnerabilities are harmless if the system is not subject of security attacks. The prevention against security attacks includes all the measures needed to minimize or eliminate the potential attacks against the system. Attack removal is related to the adoption of measures to stop attacks that have occurred before (e.g. firewalls, security patches).

Secure Software behaves correctly in the presence of a malicious utilization (attacks), even though software failures may also happen when the software is used correctly (Gary McGraw and Potter 2004). Thus, many times software development and testing are only concerned with what happens when software fails and not with the intentions. This is where the difference between software safety and software security lies: in the presence of an intelligent adversary with the intention of damaging the system.

2.2.1 Threats and Vulnerabilities

In the last two decades, the World Wide Web radically changed the way people communicate and do business. Even critical infrastructures like water supply, power supply, banking, insurance, stock market, retail, communications, defense, etc., nowadays rely on networks, on the web and on the applications that run on top of these distributed environments. The problem is that, as the importance of the assets stored and managed by web applications increases, so does the natural interest of malicious minds in exploiting this new streak.

Web applications are so widely exposed that any existing security vulnerability will most probably be uncovered and exploited by hackers. Hence, the security of web applications is a major concern and is receiving more and more attention from the research community. However, in spite of this growing awareness of security aspects at web application level, there is an increase in the number of reported attacks that exploit web application vulnerabilities (Christey and Martin 2006; Stock, Williams, and Wichers 2007).

Hackers are nowadays moving their focus from network attacks to the exploitation of vulnerabilities in the code of web applications. This poorly programmed code represents a major risk and this is why we leave vulnerabilities related to security standards and protocols outside of the scope of this work.

Attacks that target vulnerabilities related to the code of the web applications or services take advantage of improperly implemented code, searching for vulnerabilities by exployting applications' inputs with specially tampered values. These values try to take advantage of existing vulnerabilities, representing a considerable danger to the application's owner (e.g. by giving to an attacker privileges to read, modify or destroy reserved resources). This is also very dangerous because traditional security mechanisms like network firewalls, intrusion detection systems (IDS), and encryption, cannot mitigate attacks targeting web applications, even assuming that the network and key infrastructure components such as web servers and database management systems (DBMS) are fully secure. The reason is that these attacks are performed through ports that are used for regular web traffic (Singhal, Winograd, and Scarfone 2007) and even application layer firewalls cannot protect the applications as that requires a deep understanding of the business context (OWASP Foundation 2010).

Published studies show that, in general, web applications present dangerous security flaws. In February 2007, Acunetix presented the results produced from the scanning of 3,200 websites during one year (Acunetix 2007). According to the results, 70% of the web sites scanned presented high or medium risk vulnerabilities. Another interesting point is that, in the websites having high-risk vulnerabilities, the two most common vulnerabilities were SQL Injection (50% of the websites with high vulnerabilities) and Cross Site Scripting (42% of the websites with high

vulnerabilities). The NTA's Annual Security Report 2008 (NTA Monitor 2008a) states that 25% of companies tested contain one or more high-risk vulnerabilities. This number is lower than the 32% reported in 2007 (NTA Monitor 2007). Nevertheless, in some sectors (finance, government, legal, retail and utilities) the overall number of vulnerabilities found has increased. The NTA's Annual Web Application Security Report 2008 (NTA Monitor 2008b), focused in web applications, states that 17% of the applications tested contained, at least, one high-risk vulnerability and that 78% of the applications contained medium risk vulnerabilities. Although these results cannot be generalized to web services environment, they clearly show that web software is being deployed without proper security cautions.

Web services, as web applications in general, are so exposed that any existent security vulnerability will probably be uncovered and exploited, becoming the entry point for attacks. Several studies (e.g. (Vieira, Antunes, and Madeira 2009), (Fogie et al. 2007), (Jensen et al. 2007)) show that a large number of Web services are deployed with security flaws that range from code vulnerabilities (e.g. code injection vulnerabilities) to the incorrect use of security standards and protocols.

While the Open Web Application Security Project (OWASP Foundation) (OWASP Foundation 2001) presented in 2007 the ten most critical web application security vulnerabilities (Stock, Williams, and Wichers 2007), featuring at top two vulnerabilities the same as in the Acunetix’s mentioned above (although in inverse order being XSS the most critical), in the most recent report (OWASP Foundation 2013) XSS is ranked only in third place, being overtaken by injection vulnerabilities in the top of the list. Additionally, although there is no large enough study to draw definitive conclusions, the results of the study presented in Section 7.1 suggest that the specificities of web services environments lead to a slightly different set of most relevant vulnerabilities. In this context, the following bullets present examples of relevant types of vulnerabilities[5] (see (OWASP Foundation 2001) for a survey on types of security vulnerabilities):

- **SQL Injection**: allow user-supplied data to “*alter the construction of backend SQL statements*” (WASC 2008). An attacker can read or modify database data and, in some cases, execute database administration operations or commands in the system (Stuttard and Pinto 2007). Example SQL Injection A tautology-based attack is a specific type of SQL Injection attack that tries to modify one or more conditional clauses of a backend SQL query in such way that it will always evaluate to true. More details on tautologies and other types of SQL Injection attacks can be found in (W. G. Halfond, Viegas, and Orso 2006).

[5] As services work without a direct attachment to web sites, XSS is not a priority in web services environment, thus it is not included in this list.

- **XPath Injection**: allow user-supplied data to modify an XPath query to "*be parsed in a way differing from the programmer's intention*" (WASC 2008). Attackers may gain access to information in XML documents (Stuttard and Pinto 2007).
- **Code Execution**: allow manipulating the application inputs to trigger server-side code execution (Stuttard and Pinto 2007). An attacker can exploit this vulnerability to execute malicious code in the server machine.
- **Buffer Overflow**: makes it possible to manipulate inputs in such a way that causes buffer allocation problems, including overwriting of parts of the memory (Stuttard and Pinto 2007). An attacker can exploit this causing Denial of Service or, in worst cases, "*alter application flow and force unintended actions*" (WASC 2008).
- **Username/Password Disclosure**: the web service response contains information related to usernames and/or passwords. An attacker can use this information to get access to private data (Stuttard and Pinto 2007).
- **Server Path Disclosure**: the response contains a fully qualified path name to the root of the server storage system. An attacker can use this information to discover the server file system structure and devise other security attacks (Stuttard and Pinto 2007).

As mentioned before, Injection vulnerabilities are now at the top of the most critical web application security risks (OWASP Foundation 2013). These vulnerabilities *"occur when untrusted data is sent to an interpreter as part of a command or query"* (OWASP Foundation 2013). This represents a large group of vulnerabilities that includes SQL Injection, XPath Injection and Code Execution (listed above), LDAP injection and OS Command Injection, among others. In practice, the malicious inputs of the attacker can cause the interpreter to execute unintended commands or accessing/destroying forbidden data.

2.2.2 Secure Coding

To mitigate the threats referred in the previous section it is necessary to apply the best coding practices and to perform specialized security testing in order to develop non-vulnerable code (Stuttard and Pinto 2007). Before applying vulnerability detection techniques, developers should follow the coding practices that are widely accepted as suitable to produce web applications' code without vulnerabilities.

To develop web applications without security vulnerabilities a defense-in-depth approach is necessary (Howard and Leblanc 2002). This approach assumes that each security precaution can fail and so, security depends on several layers of mechanisms that cover the failures of each other. Developers are expected to apply

the effort needed to put in place adequate security precautions that minimize the probability of successful attacks.

The web applications characteristics suggest two distinct lines of defense that can be used against threats. The **first line of defense** consists in reducing the input domain of the application as a whole, acting directly on the values provided by the users. This is frequently called **input validation** (OWASP Foundation 2001) and consists in forcing the input parameters of an application to be within the correspondent valid domain or to interrupt the execution when a value outside of the domain is provided. In the case of web applications, this starts with the normalization of the inputs to a baseline character set and encoding. Then, filtering strategies must be applied over the normalized inputs, rejecting the ones that contain values outside the valid domain. This is considered to be a good practice that may avoid many problems in web applications' code. Input Validation can also be performed using **positive pattern matching** or **positive validation.** In this case, the developers establish input validation routines that identify the inputs to be accepted, contrarily to the previous case. This technique may be advantageous in some cases as developers might not be able to predict every type of attack that could be launched against their application, but should be able to specify all the forms of legal input.

A key issue is that input validation is frequently not enough as the data domain of an input parameter may allow the existence of vulnerabilities, independently of the validation performed. For instance, in the case of SQL Injection vulnerabilities, a quote is the character used as a string delimiter in most SQL statements and so, it can be used to perform a SQL Injection attack. But, in some cases the domain of a string input must allow the presence of quotes. This way, we cannot exclude all the values that contain quotes. This means that, in this case, additional security must be delegated to the database statement execution.

This additional security represents the **second line of defense** and it is necessary to complement the limitations of a general input validation strategy. In practice, each type of attack to an application targets a specific set of statements of code of the application that are prone to specific types of vulnerabilities. The second line of defense focuses on protecting these lines, for instance by guaranteeing that the values actually used lie within their input domain. Let's take the specific case of SQL Injection, in which single and double quote characters exist in the majority of attacks. Thus, some programming languages provide mechanisms for **escaping** (Shema 2010) this type of characters in such way that they can be used within an SQL expression rather than delimiting values in the statement. However, this kind of techniques has two main problems. First, it can be circumvented in some situations by using more elaborated injection techniques like combining quotes (') and slashes (\). Second, the introduction of characters for escaping increases the length of the string and thus can cause data truncation when the resulting string is higher than allowed by database.

Correctly using **prepared statements** (also named parameterized queries) is the most efficient way to avoid SQL Injection vulnerabilities (Shema 2010). When a prepared statement is created (or prepared) its structure is sent to the database. The variable parts of the query are marked using question marks (?) or labels. Afterwards, each time that the query needs to be executed, the values must be binded to the corresponding variable part. No matter what is the content of the data, the expression will always be used as a value (and not as SQL code). Consequently, it is impossible to modify the structure of the query. To help ensuring the correct usage of the data, many languages allow typed bindings.

It is important to emphasize that prepared statements, by themselves, cannot fix insecure statements. It is necessary to assure that prepared statements are used knowing how they improve security. Otherwise, using prepared statements in the same way that regular statements are used (i.e. building the SQL queries using string concatenation), and not using correctly the placeholders for the variable part of the query, will result in similar vulnerabilities (Shema 2010).

Another important concept is **output validation** (OWASP Foundation 2001), which refers to the process of validating the output of a process before it is sent to some recipient, preventing the end user from receiving information that should not be received, like information about exception inside the application that can help conducting other attacks. Another example is searching the output of an application for critical data (e.g. credit card numbers) and replacing them with asterisks (*) before sending to the recipient. Output encoding is a type of output validation and it is a mandatory precaution to avoid XSS vulnerabilities (Shema 2010). If the data sent to the browser are to be echoed in a web page, then that data should be correctly encoded (depending on the destination in the page, either in HTML encoding or percent encoding). This way, even the malicious characters used in XSS attacks become innocuous while preserving its meaning.

2.2.3 Security in the Development Process

A software development process is composed of multiple phases (Ghezzi, Jazayeri, and Mandrioli 2002). To improve the situation regarding software security it is important not only to focus on secure coding, but also to take a broader view by integrating existing approaches and tools in the development process, i.e. to use such approaches and tools in the points of the process where they can make the difference. Different authors divide the software process in different ways, but usually software development includes the following phases (which can be repeated in an iterative manner): initialization, design, implementation, testing, deployment and decommissioning. Figure 2.2 shows a simplified representation.

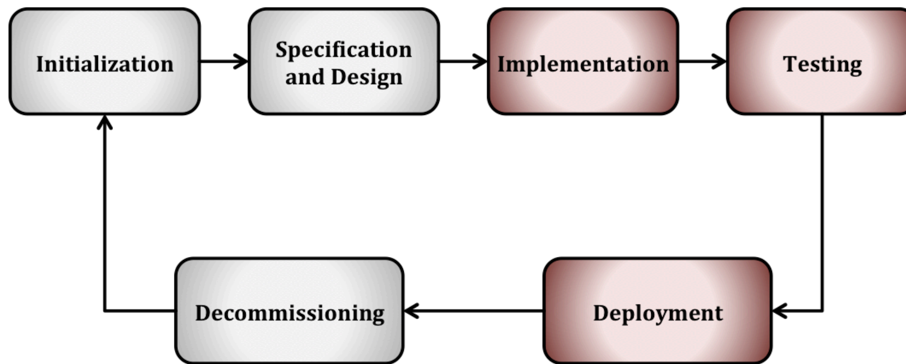


Figure 2.2 – Simplified version of a Software Product lifecycle.
The security concerns must be present during the complete cycle, but with special focus on implementation, testing and deployment.

The process starts with requirements gathering (including security requirements), followed by specification and design, implementation (coding), testing and deployment. Decommission takes place when the product is not useful/used anymore. Although code security concerns should be addressed during the entire software product development lifecycle, as highlighted by (G. McGraw 2006) especial focus should be put in three key phases (Howard and Leblanc 2002): implementation, testing, and deployment. The next points summarize the main challenges and put in the context of these three phases the concepts, techniques and tools introduced in the previous section:

- **Implementation:** during coding we must use best practices that avoid the most critical vulnerabilities in the specific application domain. Examples of practices include input and output validation, the escaping of malicious characters, and the use of parameterized commands (Stuttard and Pinto 2007). Vulnerability and attack injection techniques (Fonseca, Vieira, and Madeira 2009) have in this phase a very important job in the evaluation of the best security testing tools to use. Also, for the success of this phase, it is essential to adequately train the development teams. For instance, experience shows that the main reason for the vulnerabilities in web application's code is related to training and education. First, there is a lack of courses/topics regarding secure design, secure coding, and security testing, in most computer science degrees (Howard and Leblanc 2002). Second, security is not usually among the developers' main skills as it is considered a boring and uninteresting topic (from the development point-of-view), and not as a way to develop new and exciting functionalities.
- **Testing**: as introduced in Section 2.3.1, there are many security testing techniques available for the identification of vulnerabilities during the testing phase (Stuttard and Pinto 2007). To mitigate vulnerabilities, it is necessary to

have well-trained teams that adequately apply those techniques during the development of the application. The problem is that software quality assurance teams typically lack the knowledge required to effectively detect security problems. This way, it is necessary to devise approaches to quickly and effectively train security assurance teams in the context of web applications development, by combining vulnerability injection with relevant guidance information about the most common security vulnerabilities. Also, benchmarking techniques should be applied to assess, compare, and select the most adequate security testing tools for each concrete scenario.

- **Deployment**: at runtime, it is possible to include in the environment different attack detection mechanisms, such as Intrusion Detection Systems (IDS) and Web Application Firewalls (WAF), among others. These mechanisms can operate at different levels and use different detection approaches. The main problems preventing their use are related to the performance overheads and to the false positives that disrupt the normal behavior of the system. In this phase, security benchmarking plays a fundamental role in helping to select the best alternatives (in terms of servers, security mechanisms, etc.) to use, according to specific security requirements. Also, vulnerability and attack injection techniques represent in this phase an efficient way to evaluate the effectiveness of attack detections mechanism to be installed.

This thesis focuses on the testing phase and advances the state of the art in this area with two main contributions. First, it proposes techniques that present higher effectiveness than the existing ones. Second, it proposes an innovative integrated technique for testing service-based infrastructures for security vulnerabilities.

2.3 Vulnerability Detection

To minimize security issues, developers should search web applications and services for vulnerabilities, activity for which there are two main approaches: white-box analysis and black-box testing. Other techniques, generically named as gray-box, combine black-box testing and white-box testing to achieve better results. The following sections introduce these approaches and present some of the existing techniques and tools.

2.3.1 Black-box Testing

Black-box testing is based on the analysis of the program execution from an external point-of-view (Myers, Sandler, and Badgett 2011). In short, it consists of exercising the software and comparing the execution outcome with the expected result. Testing

is the most used technique for verification and validation of software. There are several levels for applying black-box testing, ranging from unit testing to integration testing and system testing. The testing approach can also be more formalized (based on models and well defined tests specifications) or less formalized (e.g. when considering informal "smoke testing"). The tests specification should define the coverage criteria (i.e. the criteria that guides the definition of the tests in terms of what is expected to be covered) and should be elaborated before development. The idea is that the test specification can help developers during the coding process (e.g. tests can be executed during development) and that, by designing tests a priori, it is possible to avoid biasing the tests due to knowledge about the code developed.

Test-driven development (TDD) (Beck 2003) is an agile software development technique based on predefined test cases that define desired improvements or new functions (i.e. automated unit tests that specify code requirements and that are implemented before writing the code itself). TDD begun in 1999, but is nowadays getting a lot of attention from software engineers (Newkirk and Vorontsov 2004). Development is conducted in short iterations in which the code necessary to pass the tests is developed. Code refactoring is performed to accommodate changes and improve code quality. Test-driven development is particularly suitable for web services as these are based in well-defined interfaces that are quite appropriate for unit testing. The tests specify the requirements and contain assertions that can be true or false. Running the tests allows developers to quickly validate the expected behavior as code development evolves. A large number of unit testing frameworks are available for developers to create and automatically run sets of test cases, e.g., JUnit (http://junit.org/), CppUnit (http://sourceforge.net/projects/cppunit/), and JUnitEE (http://www.junitee.org/).

Robustness testing is a specific form of black-box testing (Myers, Sandler, and Badgett 2011). The goal is to characterize the behavior of a system in presence of erroneous input conditions. Although it is not directly related to benchmarking (as there is no standard procedure meant to compare different systems/components concerning robustness), authors usually refer to robustness testing as robustness benchmarking. This way, as proposed by (Mukherjee and Siewiorek 1997), a robustness benchmark is essentially a suite of robustness tests or stimuli. A robustness benchmark stimulates the system in a way that triggers internal errors, and in that way exposes both programming and design errors in the error detection or recovery mechanisms (systems can be differentiated according to the number of errors uncovered). Web services robustness testing is based on erroneous call parameters (Vieira, Laranjeiro, and Madeira 2007). The robustness tests consist of combinations of exceptional and acceptable input values of parameters of web services operations that can be generated by applying a set of predefined rules according to the data type of each parameter.

Penetration testing, a specialization of robustness testing, consists of the analysis of the program execution in the presence of malicious inputs, searching for potential

vulnerabilities (Stuttard and Pinto 2007). In this approach the tester does not know the internals of the web application and it uses fuzzing techniques over the web HTTP requests (Stuttard and Pinto 2007). The tester needs no knowledge of the implementation details and tests the inputs of the application from the user's point of view. The number of tests can reach hundreds or even thousands for each vulnerability type. Penetration testing tools provide an automatic way to search for vulnerabilities avoiding the repetitive and tedious task of doing hundreds or even thousands of tests by hand for each vulnerability type.

Despite the use of automated tools, in many situations it is not possible to test all possible input streams, as that would take too much time. So, as soon as software specifications are complete, test cases should be designed to have the biggest coverage and representativeness possible. The most common automated security testing tools used in web applications are generally referred to as **web security scanners** (or web vulnerability scanners). Web security scanners are often regarded as an easy way to test applications against vulnerabilities. These scanners have a predefined set of tests cases that are adapted to the application to be tested, saving the user from defining all the tests to be done. In practice, the user only needs to configure the scanner and let it test the application. Once the test is completed the scanner reports existing vulnerabilities (if any detected). Most of these scanners are commercial tools, but there are also some free application scanners often with limited use, since they lack most of the functionalities of their commercial counterparts.

The process of using such scanners to test a web application or service differs in some points but shares three main stages (Acunetix 2008a) configuration, crawling, and scanning. The configuration stage includes the definition of the Uniform Resource Locator (URL) of the target web resource and the setup of the scanning parameters.

The crawling stage differs from web applications to web services. In the case of web applications, the vulnerability scanner produces a map of the internal structure of the target web application. This stage is of utmost importance as failing to discover some pages of the application prevents their testing (in the subsequent scanning stage). The scanner calls the first web page and then examines its code searching for links. Each link found is requested and this procedure is executed over and over again, until no more links or pages can be found. In the case of web services, the scanner analyses the file that specifies the interface of the service (e.g. WSDL or WADL) in order to discover the operations of the service, their input/output parameters, as well as the types of the parameters.

The scanning stage is where the automated penetration tests are performed. In the case of web applications, it is done by simulating a user clicking on links and filling in form fields (using a web browser). During this stage thousands of tests are executed. Malformed requests are also sent in order to learn the error responses. The

requests and the responses are recorded and analyzed using vulnerability policies. The responses are also validated using data collected during the crawling stage. During this stage new links are frequently discovered in web applications and when this happens they are added to the result of the crawler in order to be also scanned for vulnerabilities.

In the case of web services, the operations of the service are sequentially tested. For each operation, the process starts by executing some random non-malicious interactions (workload) to exercise the service. Afterwards, the scanner issues sequentially all the configured attacks to the operation parameters one by one. The requests and the responses are recorded and analyzed using algorithms to disclose vulnerabilities. The responses are also validated using data collected during the crawling stage. In the more advanced scanners, multiple attacks can be coordinated to find other vulnerabilities. A simple example is the use of one attack with the string “`or 1=1 --`” and another with “`or 1=0 --`”. The differences in the answers may lead to disclose a tautology vulnerability in (W. G. Halfond, Viegas, and Orso 2006).

After the scanning stage the results are shown to the user and they may be saved for later analysis. Most scanners also show some generic information about the vulnerabilities discovered, including how to avoid or correct them. Besides the graphical user interface, most scanners also have a command line application with several parameters aimed for automation by using batch jobs.

Two very popular free security scanners that support web services testing are Foundstone WSDigger (Foundstone, Inc. 2005) and WSFuzzer (OWASP Foundation 2008). **WSDigger** is a free open source tool developed by Foundstone that executes automated penetration testing in web services. Only one version of this software was released up to now (in December 2005). The tool contains sample attack plug-ins for SQL Injection, cross-site scripting (XSS), and XPath Injection, but it was released as open-source to encourage users to develop and share their own plug-ins and its test files are simple to edit to add new test cases. **WSFuzzer** is a free open source program that mainly targets HTTP based SOAP services. This tool was created based on real-world manual SOAP penetration tests, but automating them. Nevertheless, the tool is not meant to replace a solid manual human analysis. One issue with this tool is that its configuration is very complex. A problem of both WSDigger and WSFuzzer is that, in fact, they do not detect vulnerabilities: they attack the web service under testing and log the responses leaving to the user the task of examining those logs and identify the vulnerabilities. This requires the user to be an “expert” in security and to spend a huge amount of time to examine all the results.

As for commercial scanners, three brands lead the market: HP WebInspect (HP 2008), IBM Rational AppScan (IBM 2008), and Acunetix Web Vulnerability Scanner (Acunetix 2008a).

HP WebInspect is a tool that "*performs web application security testing and assessment for today's complex web applications, built on emerging Web 2.0 technologies. HP WebInspect delivers fast scanning capabilities, broad security assessment coverage and accurate web application security scanning results*" (HP 2008). This tool includes pioneering assessment technology, including simultaneous crawl and audit (SCA) and concurrent application scanning. It is a broad application that can be applied for penetration testing in web-based applications.

IBM Rational AppScan "*is a leading suite of automated Web application security and compliance assessment tools that scan for common application vulnerabilities*" (IBM 2008). This tool is suitable for users ranging from non-security experts to advanced users that can develop extensions for customized scanning environments. IBM Rational AppScan can be used for penetration testing in web applications, including web services.

Acunetix Web Vulnerability Scanner "*is an automated web application security testing tool that audits a web applications by checking for exploitable hacking vulnerabilities*" (Acunetix 2008a). Acunetix WVS can be used to execute penetration testing in web applications or web services and is quite simple to use and configure. The tool includes numerous innovative features, for instance the "AcuSensor Technology" (Acunetix 2008b).

Many other black-box tools were proposed in the past. Although those works target web applications, and not web services, we introduce some here due to the relevant innovations they bring.

WAVES (Y.-W. Huang et al. 2003) is a black-box technique for testing web applications for SQL Injection vulnerabilities. The technique is based in a reverse engineering process that identifies the data entry points of the application and attacks them using malicious patterns. An algorithm is proposed to allow "deep injection" and to eliminate false negatives. During the attack phase, the responses of the application to the attacks are monitored and machine learning techniques are used to improve the attack methodology. The problem is that the technique can only be applied to web applications (not to web services, where the interface is well defined) and ignores the user knowledge about the application being tested.

SecuBat (Kals et al. 2006) is an open-source web vulnerability scanner that uses a black-box approach to crawl and scan web sites for the presence of exploitable SQL injection and XSS vulnerabilities. SecuBat does not rely on a database of known bugs. Instead, it tries to exploit the distinctive properties of application-level vulnerabilities. To increase the confidence in the correctness of the results, the tool also attempts to automatically generate proof-of-concept exploits in certain cases.

A black-box taint-inference technique for the detection of injection attacks is proposed in (Sekar 2009). The technique does not require any intrusive source-code or binary instrumentation of the application to be protected; instead, it intercepts the

inputs and outputs of the application. Then, the technique infers tainted data in the intercepted SQL statements, and employs syntax and taint-aware policies to detect the unintended use of tainted data. However, false positives and false negatives are possible due to the accuracy limitations of the taint-inference algorithm and taint-awareness policies.

In (McAllister, Kirda, and Kruegel 2008) it is presented an automated penetration testing tool that can find reflected and stored cross-site scripting (XSS) vulnerabilities in web applications. The presented technique improves the effectiveness of web vulnerability scanners by leveraging input from real users as a starting point for its testing activity. The technique follows an entire user's session and uses recorded real user inputs to generate test cases to launch fuzzing attacks. This way, the technique increases the code coverage by exploring pages that are not reachable for other tools. The experiments show that the approach is able to test more thoroughly the web applications and identify more bugs than a number of open-source and commercial web vulnerability scanners.

2.3.2 White-box Analysis

White-box analysis consists in the examination of the code of the web service without executing it (Stuttard and Pinto 2007). This can be done in one of two ways: manually during inspections and reviews or automatically by using automated analysis tools.

Inspections, initially proposed by Michael Fagan in the mid 1970's (Fagan 1976), are a technique that consists on the manual analysis of documents, including source code, searching for problems. It is a formal technique based on a well-defined set of steps that have to be carefully undertaken. The main advantage of inspections is that they allow uncovering problems in the early phases of development (where the cost of fixing the problem is lower).

An inspection requires several experts, each one having a well-defined role, namely: author (author of the document under inspection), moderator (in charge of coordinating the inspection process), reader (responsible for reading and presenting his interpretation of the document during the inspection meeting), note keeper (in charge of taking notes during the inspection meeting), and inspectors (all the members of the team, including the ones mentioned before). During the inspection process, this team has to perform the following generic steps:

1) **Planning:** starts when the author delivers the artifact to be inspected to the moderator. The moderator analyses that artifact and decides if it is ready to undergo the inspection process. If not, then the artifact is immediately returned to the author for improvement. This step includes also selecting the remaining members of the inspection team.

2) **Overview:** after delivering the artifact (and other artifacts needed to understand it) to the experts, the author presents in detail the goal and structure of the artifact to be inspected. By the end of this meeting the inspection team must be familiar with the job to be performed.

3) **Preparation:** the inspectors analyze individually the artifact in order to prepare themselves for the inspection meeting.

4) **Inspection meeting:** in this meeting the reader reads and explains his interpretation of the artifact to the remaining inspectors. Discussion is allowed to clarify the interpretation and to disclose existing issues. The outcome of the inspection meeting is a list of issues that need to be fixed and one of three verdicts: accept (the document does not present any problem), minor corrections (the document presents minor issues that need to be fixed), and re-inspection (the document presents major issues that need to be fixed and the resulting artifact must be inspected).

5) **Revision:** the author modifies the artifact following the recommendations of the inspection team.

6) **Follow-up:** the moderator checks if all the problems detected by the inspectors were adequately fixed. The moderator may decide to conduct another inspection meeting if there were considerable changes in the artifact or if the changes made by the author differ from the recommendations of the experts.

A **code inspection** is the process by which a programmer delivers the code to his peers and they systematically examine it, searching for programming mistakes that can introduce bugs. A security inspection is an inspection that is specially targeted to find security vulnerabilities. Inspections are the most effective way of making sure that a service has a minimum number of vulnerabilities (Curphey et al. 2002) and are a crucial procedure when developing software to critical systems. Nevertheless, they are usually very long, expensive and require inspectors to have a deep knowledge on web security.

A less expensive alternative to code inspections is **code reviews** (Freedman and Weinberg 2000). Code reviews are a simplified version of code inspections that can be considered when analyzing less critical services. Reviews are also a manual approach, but they do not include the formal inspection meeting. The reviewers perform the code review individually and the moderator is in charge of filtering and merging the outcomes from the several experts. In what concerns the roles and the remaining steps reviews are very similar to inspections. Although also a very effective approach, it is still quite expensive.

Code walkthroughs are an informal approach that consists of manually analyzing the code by following the code paths as determined by predefined input conditions

(Freedman and Weinberg 2000). In practice, the developer, in conjunction with other experts, simulate the code execution, in a way similar to debugging. Although less formal, walkthroughs are also effective on detecting security issues, as far as the input conditions are adequately chosen. However, they still impose the cost of having more than one expert manually analyzing the code.

The solution to reduce the cost of white-box analysis is to rely on automated tools, such as static code analyzers. In fact, the use of automated code analysis tools is seen as an easy and fast way for finding bugs and vulnerabilities in web applications.

Static code analysis tools vet software code, either in source or binary form, in an attempt to identify common implementation-level bugs (Stuttard and Pinto 2007). The analysis performed by existing tools varies depending on their sophistication, ranging from tools that consider only individual statements and declarations to others that consider dependencies between lines of code. Among other usages (e.g. model checking and data flow analysis), these tools provide an automatic way for highlighting possible coding errors. The main problem of this approach is that exhaustive source code analysis may be difficult and cannot find many security flaws due to the complexity of the code and the lack of a dynamic (runtime) view. The following paragraphs briefly introduce some of the most used and well-known static code analyzers, including both commercial and free tools.

FindBugs is an open source tool that "*uses static analysis to look for bugs in Java code*" (University of Maryland 2009). Findbugs is composed of various detectors each one specialized in a specific pattern of bugs. The detectors use heuristics to search in the bytecode of Java applications for these patterns and classify it according to categories and priorities. Some of the highest levels of priorities are usually, among other problems, security issues.

Yasca (Yet Another Source Code Analyzer) is a "*framework for conducting source code analyses*" (Scovetta 2008) in a wide range of programming languages, including Java. Yasca is a free tool that includes two components: the first is a framework for conducting source code analyses and the second is an implementation of that framework that allows integration with other static code analyzers (e.g. FindBugs, PMD, and Jlint).

Fortify 360 is a suite of tools for vulnerability detection commercialized by Fortify Software (Fortify Software 2008). The module Fortify Source Code Analyzer performs static code analysis. According to Fortify, it is able to identify the root-cause of the potentially exploitable security vulnerabilities in source code. It supports scanning of a wide variety of programming languages, platforms, and integrated development environments.

IntelliJ IDEA is a commercial and powerful IDE for Java development that includes "*inspection gadgets*" plug-ins with automated code inspection functionalities (JetBrains 2009). IntelliJ IDEA is able to detect security issues in java source code.

These functionalities are available also in a community edition that is distributed free and open source since 2009.

Pixy is a free and open source program that performs automatic static code analysis of PHP 4 source code, aimed at the detection of XSS and SQL injection vulnerabilities (Jovanovic, Kruegel, and Kirda 2006). As referred in Pixy's webpage, "*Pixy takes a PHP program as input, and creates a report that lists possible vulnerable points in the program, together with additional information for understanding the vulnerability*".

Other approaches proposed by researchers target the detection of security vulnerabilities in web applications using static analysis. The following paragraphs present the most relevant works on this topic.

A static code analysis tool for checking type correctness of SQL queries generated dynamically is proposed in (G. Wassermann et al. 2007). This approach does not target the detection of SQL Injection vulnerabilities, but can be used to prevent attacks that take advantage of type mismatches in a dynamically generated query string to crash the underlying database. This technique is able to detect one of the root causes of many vulnerabilities in code, which is improper type checking of input. Nevertheless, this technique leaves undetected the SQL injection vulnerabilities that lead to syntactically and type correct queries.

In (Y. W. Huang et al. 2004) is presented an approach for the detection of vulnerabilities related to input validation. This approach relies on developer-provided annotations, which limits the practical applicability of the approach, and assumes that preconditions for all sensitive functions can be accurately expressed ahead of time, which is not always the case.

Wassermann and Su (G. Wassermann and Su 2004) proposed an approach that uses static analysis combined with automated reasoning to verify that the SQL queries generated in the application layer do not contain tautologies. The methodology is extremely limited because it only detects and prevents tautologies, which is only one of the many types of SQL Injection attacks that can be conducted.

Livshits and Lam (Livshits and Lam 2005) proposed a static analysis technique for detecting application vulnerabilities that stem from unchecked input, such as SQL injections and cross-site scripting. The proposed approach is based on a scalable points-to analysis and uses context sensitivity combined with improved object naming to detect vulnerabilities and keep the number of false positives low. In this approach, vulnerability signatures are described using PQL (Martin, Livshits, and Lam 2005), and a static analyzer is generated from the vulnerability description. The experimental evaluation showed that the analyzer detects instances of the specified vulnerability in the code but also showed that it produces a considerable number of false positives.

A technique to statically detect SQL injection vulnerabilities in PHP scripts is presented in (Xie and Aiken 2006). The analysis applies a custom three-tier

architecture to capture information at decreasing levels of granularity at the intrablock, intraprocedural, and interprocedural level. This architecture enables the technique to handle dynamic features of scripting languages that are not adequately addressed by other techniques. The tool was used on six popular open source PHP code bases, finding 105 previously unknown security vulnerabilities that, according to the authors, may be remotely exploitable. However, the technique presents multiple limitations, as it cannot correctly handle recursive function calls, alias and multi-dimensional arrays.

2.3.3 Gray-box Testing

The main limitation of black-box approaches is that the vulnerability detection is restricted by the output of the application. On the other hand, white-box analysis does not take into account the runtime view of the code. Gray-box approaches combine black-box and white-box techniques in order to overcome their limitations and can be used for both vulnerability and attack detection.

Dynamic program analysis consists of the analysis of the behavior of the software while executing it (Stuttard and Pinto 2007). The idea is that by analyzing the internal behavior of the code in the presence of realistic inputs it is possible to identify bugs and vulnerabilities. Obviously, the effectiveness of dynamic analysis depends strongly on the input values (similarly to testing), but it takes advantage of the observation of the source code (similarly to static analysis). For improving the effectiveness of dynamic program analysis, the program must be executed with sufficient test inputs. Code coverage analyzers help guaranteeing an adequate coverage of the source code (Doliner 2006)(Atlassian 2010).

"Acunetix AcuSensor Technology" (Acunetix 2008b) is a technique introduced by Acunetix that combines black-box scanning with feedback obtained during the test execution. This feedback is provided by sensors previous placed, using code instrumentation, inside the source code or bytecode. Acunetix states that by using this technique it is possible to find more vulnerabilities, to indicate in the code exactly where they are, and to report less false positives. This technology is only available to web applications, specifically .NET and PHP web applications. In case of .NET this technology can be injected in the bytecode.

Two techniques that combine static and dynamic analysis have been proposed to perform automated test generation to find SQL Injection vulnerabilities. SQLUnitGen, presented in (Shin, Williams, and Xie 2006), is a tool that combines static analysis with unit testing to detect SQL injection vulnerabilities. The tool uses a third-party test case generator and then modifies the test cases to introduce SQL injection attacks. These concrete attacks are obtained by using static analysis to trace the flow of user input values to the point of query generation. Sania, presented in (Kosuga et al. 2007), is a testing framework to detect SQL Injection vulnerabilities in

web applications during development and debugging phases. Sania intercepts the SQL queries between a web application and a database and constructs parse trees of these queries. Terminal leafs of parse trees typically represent vulnerable spots. The technique then generates attacks according to the syntax and semantics of these potentially vulnerable spots. Finally, Sania compares the parse trees of the original SQL query and with the ones resulting after an attack to assess the safety of these spots. The differences between the parse trees are considered vulnerabilities, originating a warning.

While other works focused on identifying vulnerabilities related to the use of external inputs without sanitizations, the work presented in (Balzarotti et al. 2008) introduces an approach that combines static and dynamic analysis to analyze the correctness of sanitization processes in web applications. First, a technique based on static analysis models the modifications that the inputs suffer along the code paths. This approach uses a conservative model of string operations, which might lead to false positives. Then, a second technique based on dynamic analysis works bottom-up from the sinks and reconstructs the code used by the application to modify the inputs. The code is then executed, using a large set of malicious input values to identify exploitable flaws in the sanitization process.

Runtime anomaly detection tools can also be used for vulnerability detection. One of those tools is AMNESIA (Analysis and Monitoring for NEutralizing SQL-Injection Attacks) (W. G. J. Halfond and Orso 2005) that combines static analysis and runtime monitoring to detect and avoid SQL injection attacks. Static analysis is used to analyze the source code of a given web application building a model of the legitimate queries that such application can generate. At runtime, AMNESIA monitors all dynamically generated queries and checks them for compliance with the statically generated model. When a query that violates the model is detected it is classified as an attack and is prevented from accessing the database. The problem is that the model built during the static code analysis may be incomplete and unrealistic because it lacks a dynamic view of the runtime behavior of the application.

2.4 Assessment and Benchmarking

A key aspect when applying automated approaches is to select the ones that are most effective from the (frequently) large set of alternatives available. This way, we need always to consider techniques to perform assessment and benchmarking. This section introduces the key concepts, techniques and tools related to assessing and benchmarking the security of computer systems, components, and tools. The topics discussed are: workload generation, vulnerability and attack injection, security benchmarking, runtime assessing and monitoring and assessment of vulnerability detection tools.

2.4.1 Workload Generation

One component stands out when testing a system: the workload. The workload or the test data represents the work that the system must perform during the experiments. In the specific case of services, the workload is no more than a set of non-malicious requests that are used to exercise the service under study to understand its behavior. Besides just generating requests, it is important to understand the proprieties of the workload, including its representativeness (how close are the characteristics of the workload to the real work the system will execute in the field) and its code coverage (how much of the code the workload is able to exercise). This way, executing the workload may also be useful to obtain feedback about the its own properties, for instance by gathering information about the code coverage of the tests using a tool like Cobertura (Doliner 2006).

Several approaches are available for workload generation. A survey on automatic test data generation techniques is presented in (Edvardsson 1999). Examples of possible approaches include the obvious random workload generation and the generation of the workload based in the automated analysis of the service source code (obviously, the source code is needed). In (Santiago et al. 2006) state charts are used for automated test case generation. Another approach is to generate the workload using the characterization of real load patterns through the application of Markov Chains (De Barros et al. 2007).

2.4.2 Vulnerability and Attack Injection

The use of fault injection techniques to assess security is a particular case of software fault injection, focused on the software faults that represent security vulnerabilities or may cause the system to fail in avoiding a security problem (Fonseca, Vieira, and Madeira 2007). Security vulnerabilities are in fact a particular case of software faults, which require adapted injection approaches. This way, in the same way fault injection techniques are essential to evaluate the effectiveness of fault tolerant mechanisms, also vulnerability and attack injection is highly valuable to evaluate security mechanisms such as vulnerability and attack detectors.

Vulnerability injection avoids the need for having vulnerable software, although there are vulnerability representativeness issues to be considered (i.e. artificially injected vulnerabilities may not be as representative as real vulnerabilities). The vulnerability injection mechanism is normally paired with an automated attack component, i.e. an attack injector that exploits injected vulnerabilities. This way, the process usually includes two major steps: first, an analysis of the target system's source is performed so that locations where vulnerabilities can be injected are identified; then, a vulnerability is injected by performing some code or configuration mutation.

In (De Barros et al. 2007) the vulnerabilities of six web applications were analyzed using field data based from 655 security fixes. Results show that only a small subset of 12 generic software faults is responsible for all the security problems. In fact, considerable differences are observed when comparing the distribution of the fault types related to security with studies on common software faults.

A procedure inspired on the fault injection technique (that has been used for decades in the dependability area) targeting security vulnerabilities is proposed in (Fonseca, Vieira, and Madeira 2009). In this work, the "security vulnerability" plus the "attack" represent the space of the "faults" that can be injected in a web application; and the "intrusion" is the "error" (Echtle and Leu 1992; Fonseca, Vieira, and Madeira 2009). To emulate real world web vulnerabilities with accuracy the work relies on the results obtained in the field study on real security vulnerabilities (Fonseca, Vieira, and Madeira 2007).

Conceptually, attack injection is based on the injection of realistic vulnerabilities that are automatically attacked, and finally the result of the attack is evaluated. As proposed in (Fonseca, Vieira, and Madeira 2009), a tool able to perform vulnerability and attack injection is a key instrument that can be used in several relevant scenarios, namely: evaluate security tools like vulnerability detectors, train security teams, evaluate security teams, and estimate the total number of vulnerabilities still present in the code, among others.

2.4.3 Security Benchmarking

Comparing different alternatives in terms of security is a difficult problem faced by many developers and system administrators. Security benchmarking allows assessing and comparing the security of systems and/or components, supporting informed decisions while designing, developing, and deploying complex software systems and tools.

Several security evaluation methods have been proposed in the past (Commission of the European Communities 1993; Infrastructure and Profile 2002; Qiu et al. 1985; Sandia National Laboratories 2012). The Orange Book (Qiu et al. 1985) and the Common Criteria for Information Technology Security Evaluation (Infrastructure and Profile 2002) define a set of generic rules that allow developers to specify the security attributes of their products and evaluators to verify if products actually meet their claims. Another example is the red team strategy (Sandia National Laboratories 2012), which consists of a group of experts trying to hack its own computer systems to evaluate security.

The work presented in (Maxion and Tan 2000) addresses the problem of determining, in a thorough and consistent way, the reliability and accuracy of anomaly detectors. This work addresses some key aspects that must be taken into

consideration when benchmarking the performance of anomaly detection in the cyber-domain.

The set of security configuration benchmarks created by the Center for Internet Security (CIS) is a very interesting initiative ("Center for Internet Security" 2012). CIS is a non-profit organization formed by several well-known academic, commercial, and governmental entities that has created a series of security configuration documents for several commercial and open source systems. These documents focus on the practical aspects of the configuration of these systems and state the concrete values each configuration option should have in order to enhance overall security of real installations. Although CIS refers to these documents as benchmarks they mainly reflect best practices and are not explicitly designed for systems assessment or comparison.

Vieira & Madeira proposed a practical way to characterize the security mechanisms in database systems (Vieira and Madeira 2005). In this approach database management systems (DBMS) are classified according to a set of security classes ranging from Class 0 to Class 5 (from the worst to the best). Systems are classified in a given class according to the security requirements satisfied.

In (A.A. Neto and Vieira 2008) the authors analyze the security best practices behind the many configuration options available in several well-known DBMS. These security best practices are then generalized and used to define a set of configuration tests that can be used to compare different database installations. A benchmark that allows database administrators to assess and compare database configurations is presented in (A.A. Neto and Vieira 2009). The benchmark provides a trust-based security metric, named minimum untrustworthiness, that expresses the minimum level of distrust the DBA should have in a given configuration regarding its ability to prevent attacks.

The use of trust-based metrics as an alternative to security measurement is discussed in (Afonso Araújo Neto and Vieira 2010). Araújo & Vieira also proposed a trustworthiness benchmark based on the systematic collection of evidences of the use (or lack of it) of secure coding practices (collected using static analysis techniques), which can be used to select one among several web applications, from a security point-of-view.

2.4.4 Runtime Monitoring and Testing

A monitoring system can be used for different purposes. For example, it can be used to check if certain service is working correctly, to check if the availability of the service fulfills the requirements defined in a Service Level Agreement (SLA), to check the compositions of the services, to get information about the architecture, among other objectives.

Several techniques based on monitoring and model checking at runtime are discussed in (Calinescu 2011). In this work it is advocated the need for using a collection of "@runtime" techniques for the development, operation and management of software capable of self adaptation and high integrity. Additionally, there is also an increasing interest in incremental model checking techniques, as shown in (Pistore et al. 2004) that presents techniques based on "*Planning as Model Checking*" to automatically compose web services and synthesize monitoring components.

Petri nets are also used together with a simple monitoring system to model the external behavior of the software in (Grosclaude 2004). The components are associated to a local controller that scrutinizes its messages and compares them with the specified behavior. As the components interact, information about errors and time constraints violations are collected and analyzed to infer indicators about the state of components.

In (B. Wassermann and Emmerich 2011), is described a monitoring system for Web Service compositions called Monere. It instruments some components across the layers of a service composition and exploits the structure of BPEL workflows to obtain structural cross-domain dependency graphs. In (Baresi, Ghezzi, and Guinea 2004), the authors propose approaches to monitor dynamic service compositions with respect to contracts expressed via assertions on services. In (Gao et al. 2000) is included a support for monitoring of software components in component-based programs. In (Zubin67 2010), it is presented a module for monitoring services in Service Oriented Architectures, focusing on the Enterprise Service Bus (ESB) level, allowing collecting data in a easy way even if the services are executed on different servers.

A tool for testing SOAs is proposed in (Ceccarelli, Vieira, and Bondavalli 2011a). It is supported by a discovery algorithm that is able to trace the SOA evolution by automatically discovering the services that compose the architecture and the connections among them. This approach is then used in the context of a testing service for SOA validation (Ceccarelli, Vieira, and Bondavalli 2011b) that is basically a composite service able to monitor SOA evolution and test the various services according to specific testing policies. The work proposes the use of copies of the services to avoid service degradation and error propagation caused by the testing activity. This algorithm is based on a collaborative approach where providers need to share information they have on their part of a SOA.

In (Bertolino et al. 2006; Bertolino et al. 2009) the authors present a way for online testing for Web Services, more specifically interoperability testing, where the service invocations are redirected to stubs, suggesting the correctness of a service against its specification. The problem with the proposed approach is that services that are already active in SOA do not take part in the testing process. For runtime testing, the

system under production should be protected from undesired side effects, which is also not considered in this work.

2.4.5 Assessment of Vulnerability Detection Tools

Regardless of the importance they have, automated approaches for vulnerabilities detection are frequently unable to produce accurate results. In consequence of this, many works have been published proposing methodologies to evaluate tools and presenting results of tools evaluations. Following are presented the most significant published studies.

In what concerns to web security scanners, previous research suggests that their effectiveness in the detection of vulnerabilities varies a lot, being often unsatisfactory. In (Fonseca, Vieira, and Madeira 2007) it is proposed a method to evaluate and benchmark automatic web vulnerability scanners in web application's environment using vulnerability injection techniques. Software faults are injected in the application code and the tool under evaluation is executed, showing its strengths and weaknesses concerning coverage of vulnerability detection and false positives. The study focused on the SQL Injection and Cross Site Scripting (XSS) types of vulnerabilities. Three leading commercial scanning tools were evaluated and the results showed that in general the coverage is low and the percentage of false positives is very high (ranging from 20% to 77%).

Another evaluation of web vulnerability scanners is presented in (Doupé, Cova, and Vigna 2010). Both commercial and open-source scanners were evaluated, in a total of 11 scanners. To test the tools the authors introduced different types of vulnerabilities in a realistic web application, challenging the crawling capabilities of the tools. The main findings of the study were that the crawling process is critical to the success of the scanning process and that many classes of vulnerabilities are completely overlooked by these tools.

In (Teixeira, Antunes, and Neves 2007) is proposed a preliminary version of a benchmark to compare the effectiveness of static analysis tools effectiveness. This is based on a study conducted using code analysis tools freely available on Internet. The benchmark uses an application developed by the authors containing vulnerabilities manually introduced. The tools are run over the application to find the existing vulnerabilities. The experiment has shown that each tool is able to find only few classes of vulnerabilities. Using these results work was developed to aggregate the results of the benchmarked tools originating a new tool with higher coverage and less false positives. However, the work targets types of vulnerabilities that are not the focus of this Ph.D. work and the fact that the application used to evaluate the tools is synthetic hurts the representativeness of the evaluation.

In (Wagner et al. 2005) authors evaluated three bug finding tools and compared their effectiveness with a review team inspection. The tools achieved higher efficiency

than the review team in detecting software bugs (the study did not consider security issues) in five industrial Java-based applications, but all the tools presented false positive rates higher than 30%. The work focuses on application code defects that are also outside the focus of this thesis.

In (Bau et al. 2010) the authors evaluated 8 automated black-box leading tools it terms of the class of vulnerabilities tested, their effectiveness, and the relation of the target vulnerabilities to vulnerabilities found in the field. The study was conducted using a vulnerable web application and previous versions of widely used web applications containing known vulnerabilities. The results showed that "stored" forms of Cross Site Scripting (XSS) and SQL Injection (SQLI) vulnerabilities are not currently found by many tools.

Another important technique to evaluate vulnerability detection tools is vulnerability injection. In the same way fault injection is an approach that can be used to validate specific fault handling and fault detection mechanisms (Carreira, Madeira, and Silva 1998), vulnerability injection is a powerful tool that can be used to assess the effectiveness of vulnerability detection and attack detection tools and methodologies. In (Fonseca, Vieira, and Madeira 2009) is proposed a methodology to inject vulnerabilities and attacks in web applications. The methodology can be used to evaluate both defensive mechanisms and vulnerability detection mechanisms. As mentioned in Section 2.4.2, to provide realistic vulnerabilities the methodology is based on a field study that included a large number of vulnerabilities in web applications. During the experimental evaluation the methodology was used to evaluate the coverage and false positives of an intrusion detection system for SQL injection and two web vulnerability scanners. The problem is that the methodology applies only to web applications and does not take into account the specificities of the web services environment. Additionally, the tool presented is specific for LAMP (Linux, Apache, MySQL, and PHP) web applications and requires access to the application source code to perform vulnerability injection.

Although the works presented until now tried to assess the effectiveness of vulnerability detection tools, none has proposed a standard approach that allows the comparison of tools and so developers urge the definition of practical approaches that help them comparing alternative tools concerning their ability to detect vulnerabilities. As performance benchmarks have contributed to improve the performance of systems, we believe that the use of a benchmarking approach on automated vulnerability detection tools is for improving the state of the art on vulnerability detection in services-based environments.

2.5 Conclusion

This chapter presented background on services and service-based infrastructure and discussed the state of the art in terms of vulnerability detection tools and assessment and benchmarking techniques.

The basic characteristics of service-based environment were introduced and the security threats faced in such environments clearly show the need for new means to help developers and researchers improving security. The framework proposed in Chapter 3 is a possible answer towards advancing the state of the art in this area.

A key aspect is that, although a large number of vulnerability detection tools were introduced, very few aim to tackle the specific problem of vulnerabilities in service-based environment. This shows the need for an integrated approach able to tackle these specific requirements as proposed in Chapter 5.

The findings regarding works on assessing and benchmarking the value of vulnerability detection tools show that in many cases the existing tools present very low effectiveness. The low effectiveness of the tools presented show the need for the development of more efficient tools, as the ones presented in Chapter 4.

Finally, most of these studies do not fit the model of a standard way to evaluate and compare vulnerability detection tools, as they mostly consist of ad-hoc experiments design for the specific tools under scrutiny. Additionally, most of studies presented above are focused specifically in web applications and the results obtained cannot be easily generalized, especially if we take into account the specificities of web services environments. This highlights clearly the need for benchmarking approaches focusing vulnerability detection tools for web services, as the one presented in Chapter 6.

Chapter 3
Framework for the Detection of Vulnerabilities in Service-Based Infrastructures

In order to support business-critical scenarios, Service-Based Infrastructures must be secure and reliable (Singhal, Winograd, and Scarfone 2007). However, studies and reports show that both web applications and services are many times deployed with security vulnerabilities (Vieira, Antunes, and Madeira 2009; NTA Monitor 2011a; OWASP Foundation 2013). Studies also show that, although penetration testing is considered to be the vulnerability detection technique most used by web developers (Stuttard and Pinto 2007), it has limited effectiveness, reporting very high numbers of false positives, while leaving many vulnerabilities undetected (Vieira, Antunes, and Madeira 2009; Fonseca, Vieira, and Madeira 2007; Doupé, Cova, and Vigna 2010). Furthermore, the specific characteristics of service-based infrastructures raise new security requirements, not addressed by existing tools. First, these infrastructures are dynamic in nature, facing changes in the services used and in the way they interact. Second, these infrastructures usually include services that are under the control of multiple providers. Finally, it is necessary to consider interactions with resources and other services.

Such requirements create the need for innovative techniques that help developers improving the current situation. In this chapter we define a framework[6] that provides the context for the work on **detecting vulnerabilities in service-based**

[6] In the context of this work, we define a *framework* as being a set of assumptions, concepts, and practices that represent a way of viewing and addressing a problem.

infrastructures presented in the thesis. The framework is composed of three key parts, as introduced in the next paragraphs.

To help understanding the research challenges and to clarify the types of infrastructures targeted, we defined a **reference service-based infrastructure**. This infrastructure, portraying the typical characteristics of web services, guided the work conducted and influenced the solutions proposed. In practice, the reference infrastructure defines the security challenges to be addressed, including both the traditional security requirements and the ones that are raised by service-based orientation.

The services considered in the reference infrastructure are divided in three testing scenarios with specific characteristics. Addressing these testing scenarios calls for techniques that take into account the different access conditions to the services under testing. Furthermore, a generic procedure that supports the design of *standardized* and modular tools and that guarantees exhaustive and effective detection of vulnerabilities is needed. The idea is that the different components should implement specific features of the tool, in a decoupled manner, allowing for easily designing and later improving the tool. This way, we propose a **generic approach for designing vulnerability detection tools for web services**. This approach includes the definition of the testing procedure and of the main components that should be implemented by the tool.

To orchestrate and integrate the tools developed following such design approach, in a way that allows detecting vulnerabilities in service-based infrastructures in general, we propose a **generic integrated approach** that encompasses aspects like how to create and manage a description of the underlying architecture, how to define what should be done gather information about the infrastructure, and how to run the tests using the tools that best suit each service.

The outline of this chapter is as follows. The next section presents the reference infrastructure, discussing the characteristics of web services, the main security challenges, and the testing scenarios considered. Section 3.2 presents the generic approach for designing vulnerability detection tools for web services, including the components and the testing procedure that a tool should implement. Section 3.3 presents the integrated approach for detecting vulnerabilities in service-based infrastructures. Finally, Section 3.4 concludes the chapter.

3.1 Reference Service-Based Infrastructure

Service-based Infrastructures and **SOAs** represent the response for the need to simplify the IT infrastructure of organization, improving at the same time interoperability and increasing the business agility, Although they can be implemented in multiple different ways, there is a consensus about their basic

design principles (Erl 2005; Papazoglou and Heuvel 2007), which are summarized next.

It is widely accepted that the functionalities implemented by the system should be available in the form of **services** that allow the interaction between consumers and providers. Services are reusable components that efficiently deliver business functionalities within a protocol-independent distributed environment through a standardized interface. Services must also be autonomous and self-contained, coarse-grained and loosely coupled. In order to help the implementation of these principles, the services are frequently connected through a service bus component, designated as Enterprise Service Bus (ESB) (Keen et al. 2004).

A detailed representation of the concepts and components around service-based infrastructures could become too complex to be analyzed, thus rendering such representation useless. This way, it is necessary to make some **representation simplifications** in order to focus on the key parts of these infrastructures that are the subject of this thesis. Also, as it is not possible to address all the existing technologies, we need to **reduce the diversity** of technologies addressed. Finally, it is necessary to make some **assumptions** in order to help directing the development of techniques and tools. To accomplish such needs, we established a reference infrastructure that includes the key concepts and components that should be kept in mind to understand the present work. Figure 3.1 portrays this reference infrastructure.

The figure shows a simplified example of a service-based environment, with a small number of Services *(Sx)* and Resources *(Rx)*. The example includes a Provider *(P0)*, with a gray ellipse that represents the parts of the system under his total control. This provider offers two services (*S0.5* and *S0.6*) to the exterior, while other resources (*R0.7* and *R0.8*) are for internal access only. While the service *S0.6* is fully under control, the service *S0.5* is not. In fact, the former uses only the services and resources owned by the provider to complete its business functionality, but the latter (*S0.5*) uses some resources that are outside the control of the provider (*R3* and *S4*).

The services and resources outside the gray ellipse are not controlled by *P0*, thus being distributed throughout the Internet. Although they are not under control of *P0*, they are known and within-reach, which means that *P0* is able to invoke them. The example also includes a consumer *(C)*, which represents the users of the infrastructure that can be persons, organizations or even other systems. In this specific case, *C* is a consumer that uses services *S1*, *S2*, *S0.5* and *S0.6*. The cloud *(?)* represents parts of the system about which *P0* has no information, but that may be used by some of the services not under his full control.

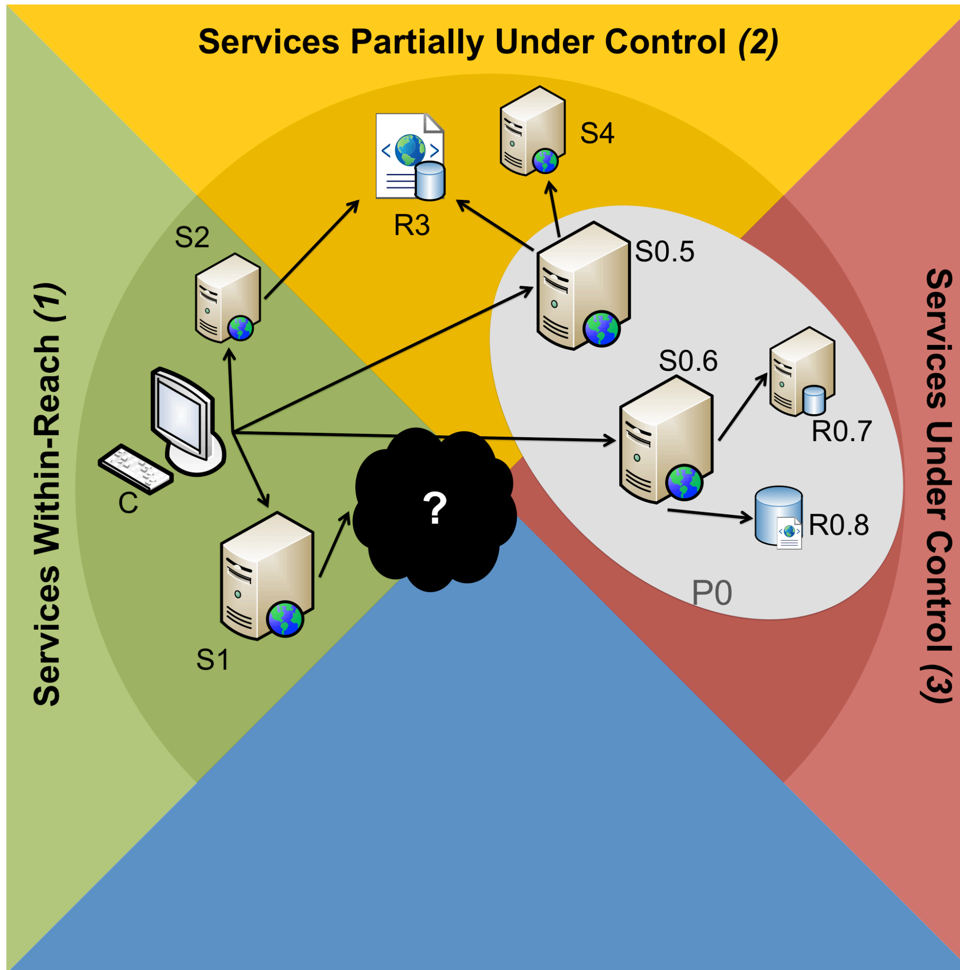


Figure 3.1 – Reference Service-Based Infrastructure.
Simplified representation of the most important concepts and components on service-based infrastructures needed to better understand the present work.

It is important to emphasize that this is a simplified representation of a service-based infrastructure, and thus there are some key observations that should to be made. First, service-based infrastructures are frequently much more complex than the represented one. However, in order to maintain the analysis feasible, we decided to keep the representation as simple and clear as possible. Second, no service bus is represented, which does not mean that the services are connected in a point-to-point basis. We opted by not representing a service bus because it is not an important point for our work (we just need to know the interfaces of services and that they may be inter-connected). Finally, service-based environments are dynamic, meaning that they can change over time, which we opted not to represent in the figure but that is important for our work, as it makes testing a continuous and difficult process, requiring tools to cope with such evolution.

3.1.1 Specific Characteristics of Web Services

A **Web Service** delivers some business functionality in a protocol-independent distributed environment through a standardized interface, which allows the service to be used independently from the implementation. There are several specificities that **distinguish web services from other web applications**. These should be kept in mind by researchers and developers when talking about detecting security vulnerabilities, and include:

- Web services are **reusable components** that deliver business functionalities in an efficient way within a protocol-independent distributed environment through a standardized interface. The reusability allows them to be applied in different business processes while their standardized interface enables access to be independent from implementation specificities. This way, any problem in a web service may affect the organization in multiple ways.
- Web services have a **well-defined interface**. This is mostly a positive aspect, as it avoids the need for a crawling/learning phase (required by some vulnerability detection approaches to learn the interface of a web application), but makes it easier to mask information about internal problems of the application (internal errors, exceptions raised, etc.). This can be a limiting factor, as it reduces the available information (e.g. compared to what testing tools can extract from the service' responses during crawling).
- Web services may include **several operations**, as defined in the service interface. Each operation includes several input and output parameters, that have a data type. The data types may be simple or complex. Besides the data type, each input parameter may have a domain that may be defined in the interface or not. When defined in the interface, the input domain allows improving the interoperability of the service.
- Web services must be autonomous and self-contained, coarse-grained and loosely coupled, allowing **multiple functionalities** without **adding design complexity or increasing communication**. In fact, web services should represent an effective way for reducing the complexity and overhead that comes with custom-coded interfaces allowing at the same time efficiently managing changes and evolution.
- The **interoperability and reduced dependency** among services not only facilitates their replacement or modification (without requiring changes in other parts of the system), but also requires vulnerability detection to be effective (i.e. conducted in reduced times in order to not delay the deployment process) and to take into account the potential interactions among services.

- In many situations, the user that needs to test the security of a web service is not its owner, thus **cannot access its internals** (a requisite for some vulnerability detection techniques). A common example of this scenario is when a consumer has to select a service from a multitude of alternatives provided by third parties.

Although the characteristics above are common to services implemented in different kinds of technology, there is one key technology-dependent difference: RESTful web services do not always have a well defined and machine readable description, while SOAP web services do. In fact, although WADL constitutes a clearly structured, detailed and extensible format to describe web applications, it does not seem to be widely adopted by developers, probably due to its perceived complexity (Kopecky, Gomadam, and Vitvar 2008). This way, in most cases RESTful service descriptions remain as unstructured text (Kopecky, Gomadam, and Vitvar 2008). On the other hand, SOAP web services are required to have a WSDL file describing its interface, increasing interoperability and fostering adoption.

3.1.2 Challenges in Service-Based Infrastructures

Although the problem of security testing of software services has been addressed in the past (see Section 2.3), most of the existing works focus on testing a single service at a time, disregarding key characteristics of service-based environments. In fact, several challenges are raised when considering security testing in service-based infrastructures and SOAs:

- It is necessary to **consider interactions between services and other resources or services**. Thus, besides testing each service offline and individually from each other, testing tool should take into account the overall architecture of the infrastructure. In practice it, is necessary to test all the interfaces of the services, including the ones between a service and the resources and other services used (contrarily to black-box techniques that focus only on the interface between the service and the external user).
- Service-based infrastructures are usually built using services that are **under the control of multiple providers**, creating the need for testing tools that can cope with different levels of available information and different levels of access (e.g. the source code may be available or not).
- The services under control of an organization may invoke services that were developed internally or **externally by a third party**. Service consumers are not necessarily end-user applications, but can also be portals, internal or external systems, or composite services, making use of other services. This creates the need for tools that cope with different levels of access and information to the services.

- SOAs are **dynamic in nature**, facing changes related to services being added, removed or updated (i.e. new versions are deployed), as well as related to the way services interact with each other. This brings the need for automated approaches able to continuously monitor and test the whole architecture in an automated way.
- Frequently, the third-party services to be invoked **are only known at runtime** using directory services (or brokers). These brokers allow the consumer to find the services that fulfill some kind of criteria or set of functionalities. However, it is the consumer's responsibility to check whether the offered service meets the desired security requirements. This creates a need for tools that detect changes at runtime and discover new services to be tested.

3.1.3 Web Services Testing Scenarios

Knowing both the specificities of web services and the challenges of detecting software vulnerabilities in service-based infrastructures in general, we can identify distinct testing scenarios representing the information and level of access that the user can have to each service. In the context of this work three scenarios are envisaged:

1) ***Services under control:*** a service is under control and also the resources that it uses are known, like in the case of S3 in Figure 3.1. This service uses only services and resources owned by the provider P0 to complete its business functionality. It is possible to use all kinds of vulnerability detection techniques, including the ones that require access to the source code (e.g. static analysis).
2) ***Services partially under control:*** a service is under control but some of the resources that it uses are not, like in the case of S2 in Figure 3.1 (it uses resource R2 that is not under control). In practice, this service requires services and/or resources that are not owned by the provider P0 to complete its business functionality. In this case, it may not be possible to access the source code (e.g. in the case of legacy systems or systems based in off-the-shelf components). However, all the interfaces between the service and the external environment are known, which allows one to obtain relevant information about input domains and about the use of external resources. This allows using techniques that take advantage of such information (e.g. techniques based on interface monitoring).
3) ***Services within reach:*** a service is within reach but not under control, like in the case of S1 in Figure 3.1. This means that provider P0 is able to invoke this service, but not to control it (e.g. has no access or detailed information about

the service). As it is not possible to access the internals of the service, only black-box testing techniques can be used. This scenario also represents the typical point-of-view of the consumer.

The cloud identified with a question mark (**?**) represents services and resources that are unknown to provider P0, but may be used by other within-reach services, like in the case of S1 in Figure 3.1. The provider has no access to the internals of within reach services and, consequently, he has no information about the unknown services.

As mentioned before, these scenarios call for a solution based on effective tools for vulnerability detection that are supported by innovative techniques that take into account the different access conditions to the services under testing. In order to tackle the characteristics of service-based infrastructure, these techniques may be integrated into an iterative testing process that monitors, discovers and tests services at runtime. However, applying this kind of approach after deployment raises new problems, including the impact of the testing in services that are running, and failure propagation, for which different works proposed and studied alternative supporting techniques such as sandboxing, virtualization, etc. For instance, past works tried to overcome this difficulty by testing copies of services, avoiding service degradation and error propagation caused by the testing activity (Ceccarelli, Vieira, and Bondavalli 2011b). Another approach is virtualizing the services under testing as in (Michelsen and English 2012). Although this is a very important challenge, these concerns are **out of the scope** of this work and thus, we rely on other works to provide the required support for the testing process.

3.2 Designing Vulnerability Detection Tools for Web Services

Research and practice shows that state-of-the-art scanners frequently present low effectiveness both in terms of vulnerability coverage and false positive rates (Vieira, Antunes, and Madeira 2009; Fonseca, Vieira, and Madeira 2007). The main problem is that most of these tools try to be as generic as possible (to detect many types of vulnerabilities), but are typically **very limited in terms of the detection approaches** they implement for each vulnerability type and do not take advantage of the specific access conditions to the target services.

Building effective tools demands for innovative techniques that take into account the different access conditions to the services under testing (i.e. that consider the testing scenarios presented in Section 3.1.3). For example, if one has access to the web service interfaces (e.g. Scenario 2), including interfaces with external resources like other services or databases, then an improved technique based on interface

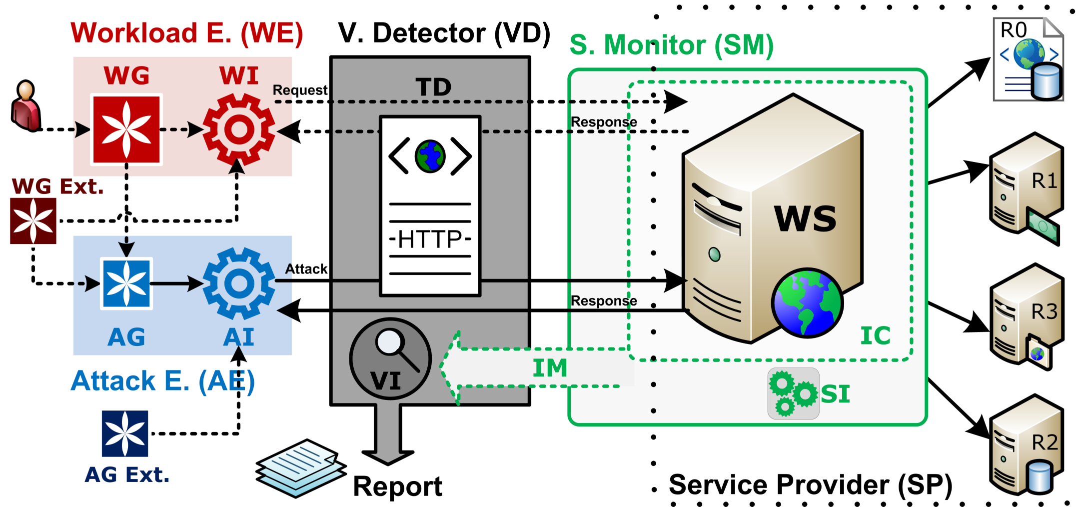
monitoring may be used in detriment of traditional penetration testing. Furthermore, we argue that vulnerability detection tools should implement a procedure that supports the **design of standardized and modular tools**, based on multiple components implementing specific features in a decoupled manner, thus allowing for easily designing and later improving the tool.

This section **proposes a generic approach for designing vulnerability detection tools for web services**. The approach defines the components and the testing procedure that a tool should implement. The components include a workload emulator (responsible for generating and executing a set of requests to exercise the web service), an attack emulator (in charge of generating and injecting requests that simulate attacks), a service monitor (in charge of instrumenting the service under testing, if needed, and collecting relevant information to support vulnerabilities identification), and a vulnerability detector (responsible for analyzing the collected information and identify vulnerabilities, and for running the testing procedure). Figure 3.2 depicts these components, and the relation among them and with a web service under testing.

As it is possible to observe, the workload emulator and the attack emulator work together to create and submit attacks, the vulnerability detector uses knowledge about the attacks and information collected from the web service to identify vulnerabilities, and the service monitor is in charge of instrumenting the web service and collecting information to feed the vulnerability detector (the type of information collected depends on the vulnerabilities being detected and on the detection technique used). The vulnerability detector is also in charge of implementing the testing procedure by coordinating the remaining components.

Due to the high diversity of web services technologies, types of vulnerabilities, and vulnerability detection approaches, designing an effective tool requires focusing on a well-defined domain. In fact, the division of the spectrum into well-defined areas is necessary to better support decisions during the definition of the components and procedure. In this context, the **definition of the tool domain** includes selecting:

- The **class of web services** (e.g. SOAP, REST), which allows understanding the characteristics of the services that will be tested.
- The **types of vulnerabilities** (e.g. SQL Injection, XPath Injection, file execution) that should be detected by the tool.
- The **vulnerability detection approach** (e.g. penetration testing, anomaly detection) that the tool will use to detect vulnerabilities.



Legend:

WE	Workload Emulator	
		User input
	WG	Workload generator
	WI	Workload injector
	WG Ext.	External workload generator
AE	Attack Emulator	
	AG	Attack generator
	AI	Attack injector
	AG Ext.	External attack generator
VD	Vulnerability detection module	
	TD	Testing driver
	VI	Vulnerability identifier
R	Vulnerabilities detected	
SM	Service monitor	
	IC	Information collector
	SI	Service instrumentation
	IM	Information manager
SP	Service Provider	
	WS	Web Service under test
R0	XML database	
	R1	Payment gateway
	R2	Database
	R3	FTP server
---	Optional elements or connections	

Figure 3.2 – Design of a web service vulnerability detection tool.
Generic interactions between the modules and also with the web service. Some of the presented modules are optional, as it will be possible to observe in other related figures.

The next subsections detail the proposed components. The testing procedure is discussed together with the vulnerability detector component. Note that the goal of this section is not to design a specific tool (that is done in Chapter 4), but to define an approach that can be used to design tools for detecting vulnerabilities in web services.

3.2.1 Workload Emulator

The **workload emulator** (WE) component is in charge of generating a set of valid requests. These requests will be used by the attack injector to generate attacks, but can also be executed in the absence of attacks to exercise each operation of the web service under testing and thus understanding its typical behavior. The WE includes two elements: a workload generator and a workload injector.

The **workload generator** (WG) starts by obtaining the required definitions about the web service under testing. As mentioned before, we assume that the web service interface is described in a descriptor file (e.g. WSDL, WADL) (D. A Chappell and Jewell 2002; Richardson and Ruby 2007), which should be processed to obtain the list of operations, parameters (including return values), and associated data types. However, as in most cases the valid values for each parameter (i.e. the domain restrictions of the parameter) are not available in that file and associated schemas, the user should be allowed to provide additional information about the valid domains for each parameter (including for parameters based on complex data types, which are composed by a set of individual parameters). Note that, for web service operations with several input parameters, the valid domain for a given parameter may be dependent on the value specified for another parameter (e.g. for a service that has the parameters "country" and "city", the domain of the city depends on the country, which must also be specified).

A workload (set of valid web service calls) should then be generated to exercise each operation of the web service under testing. As it is not possible to define a generic workload that fits all web services, a specific workload is needed for each service under testing. A vulnerability detector may provide more than one way to generate the workload, thus offering to the user the option of selecting the one that best fits his requirements. Usually, three alternatives are available for implementing the workload generation:

- **Use a user-defined workload generator**: in this case the user of the tool should implement a generation component based on the knowledge he has about the service being tested. The workload emulator should provide an easy way for integrating this generation component, which needs to interact with the workload injector (in charge of submitting the workload requests to the services under testing) and with the attack generator (that creates attacks based on the workload requests in an educated manner, as described in Section 3.2.2).
- **Use the functions of an existing vulnerability scanner:** consists of supporting the integration of external tools in a similar way to the user-defined workload. A key aspect is that in this case we are not interested in the attack generation and vulnerability detection capabilities of such tool (this will be addressed later in the design of the attack injector and of the

vulnerability detector), but only on interface identification and workload generation features. Also note that selecting an existing vulnerability scanner to provide this feature may not be an easy task as, depending on the type of vulnerabilities to detect and on the detection approach to implement, existing scanners may be limited in the support they provide (thus such selection process is of utmost importance).

- **Include a workload generator module in the tool:** several approaches can be used for generating web service requests, including (see Section 2.4.1 for details on these approaches): deterministic generation (e.g. based on constant values, step functions, values shuffling, etc.), stochastic methods (e.g. based on uniformly distributed random values, random values added by a step function, and Gaussian, Poisson, or exponential distributions), and hybrid approaches (a combination of multiple approaches). As the goal is to generate (valid) requests that adequately exercise the services under testing (i.e. allow achieving a high coverage of the code under testing), this process should take into account the web service definitions mentioned above. It is also of extreme importance to design a workload generator that satisfies the requirements in terms of the target vulnerabilities and of the detection approach being implemented.

The **workload injector** (WI) component takes the workload generated and submits it to the web service. This is an optional element, as some approaches may not require the execution of a workload. For example, classical penetration testing is based only on the execution of the penetration tests. On the other hand, anomaly detection approaches require a training phase, thus a workload execution is required. A feature that may be added to the workload injector is code coverage analysis (Doliner 2006; Atlassian 2010). The idea is that in the cases where source code or *bytecode* is available, code coverage can be used to drive the generation of the workload requests. To obtain such a code coverage value a tool that analyzes the execution profile of the web service during the execution of the workload may be used (e.g. Cobertura (Doliner 2006), Clover(Atlassian 2010), etc.). As the completeness of code coverage is always relative to a specific population of possible test cases, many metrics are available, including Line coverage, Branch coverage, Path coverage, Loop coverage, among others (Kaner 1996). The calculated coverage value should be used to decide if more requests are needed to increase coverage. In such case, the injector component should ask the generator component to create additional requests.

3.2.2 Attack Emulator

The **attack emulator** (AE) component is in charge of automatically generating attacks and of submitting them to the web services under testing. For this, it includes two

elements: an attack generator, in charge of creating attacks, and an attack injector, responsible for submitting those attacks.

Two alternatives can be considered for implementing the **attack generator** (AG):

- **Use the functions of an external vulnerability scanner**: this consists of supporting the integration of external generators in a similar way to what is done for the workload generation (see previous subsection). As before, selecting an adequate external attack generator may be difficult, because existing tools may not implement the testing strategies required to successfully implementing a given vulnerability detection approach.
- **Include an attack generator module in the tool:** this module takes the workload and replaces valid values by malicious ones following a set of mutation rules (see Table 3.1 for examples of typical mutation rules for SQL Injection and XPath Injection). Obviously, the mutation rules depend on the type of vulnerabilities to detect and should be as complete as possible in order to achieve high detection coverage. This way, defining the set of mutation rules is a complex task that should consider multiple sources of information, including information on how existing tools work, knowledge on previous successful attack attempts in the field, and scientific references.

Table 3.1 – Examples of SQL/XPath Injection attack types.
The attack generator module should use this type of rules to mutate the workload elements into attacks.

SQL/XPath Injection mutation rules
" or 1=1 --
" or 1=1 or ""="
' or (EXISTS)
' or uname like '%
' or userid like '%
' or username like '%
' UNION ALL SELECT
' UNION SELECT
char%2839%29%2b%28SELECT
" or 1=1 or ""="
' or ''='

Although the process of generating the attacks may depend on the vulnerability detection technique, we propose a generic procedure whose goal is to support the design of generation approaches capable of creating comprehensive sets of attacks. As shown in Figure 3.3, such procedure includes **several phases**, where each phase focuses on generating malicious calls that target a given operation of the web service and includes a **set of steps**. Each step targets a specific parameter of the operation,

and comprises several **attack sets**. An attack set includes the attacks to be performed over a given parameter, which are generated by applying the mutation rules mentioned before. Obviously, the same mutation rule may be applied one or more times over the same input parameter in order to increase the code coverage of the tests (as the requests of the workload may use different values for the remaining parameters).

The **attack injector** (AI) component is in charge of submitting the generated attacks to the web service under testing. If an external attack generator is used, then the injector should provide the required integration interfaces. Similarly to the workload injector, the attack injector may also support coverage analysis features, whose output can be used to drive the generation of additional attacks.

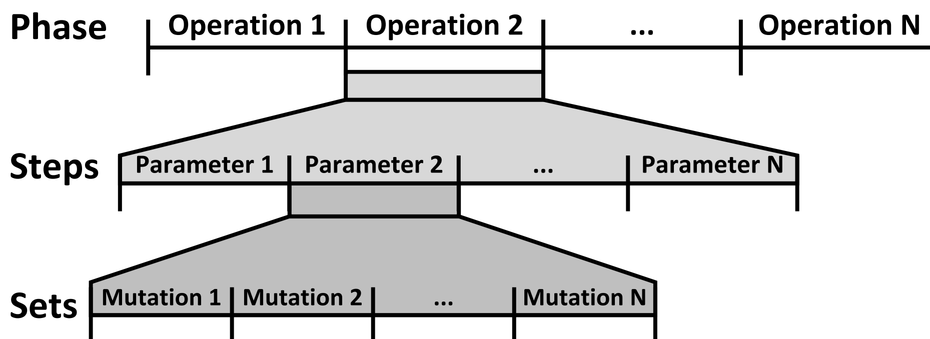


Figure 3.3 – Generic process for generating the attacks.
Workload elements are used to generate attacks based on attack mutation rules. Only one parameter is attacked at a time.

3.2.3 Service Monitor

To identify vulnerabilities we need to collect as much information as possible about the behavior of the web services under testing (this information is later used by the vulnerability detector component). Obviously, the information that can be collected depends on the access conditions to the target web services. In fact, as explained in Section 3.1, these environments are based on services that can be under the control of multiple providers, and the users of the testing tool may have different types of access to the services to be tested (i.e. services may be within reach, partially under control, or under control).

Each vulnerability detection technique has its specific requirements in terms of the information needed, but the most basic information are the web services requests and corresponding responses (to both workload requests and attacks). In the case of more advanced approaches, additional information may be related to the internal

functioning of the web service, to the web services interfaces (including interfaces with external resources like other services and databases), etc. This way, the service monitor should be able to instrument the target services in a way that allows collecting the required information, in the less intrusive way possible. Depending on the type of information needed and on the level of access to the internals of the web service, there are multiple options to monitor web services. Some examples are:

- **Network packet sniffing:** consists of reading each packet as it flows across the network. Packet sniffers usually work by setting the network interface into a mode in which it captures all traffic (Fuentes and Kar 2005). Multiple tools and libraries are available to perform packet sniffing, for instance Tcpdump & Libcap (Fuentes and Kar 2005);
- **Use a proxy:** a proxy is a relay for requests. The clients send the requests to the proxy, which then forwards them to the destination server, whose response is also sent to the proxy before being forward to the client. During this process, the proxy is able to read and modify the requests and responses. An example of a HTTP proxy implemented in Java is LittleProxy (LittleShoot 2010);
- **Driver instrumentation:** when the interface to be monitored is accessed through a driver (e.g. Java applications use JDBC drivers to access the database server), this driver can be instrumented to include monitoring facilities. In most cases, this can be achieved using Aspect-Oriented Programming (AOP) (Kiczales et al. 2002).

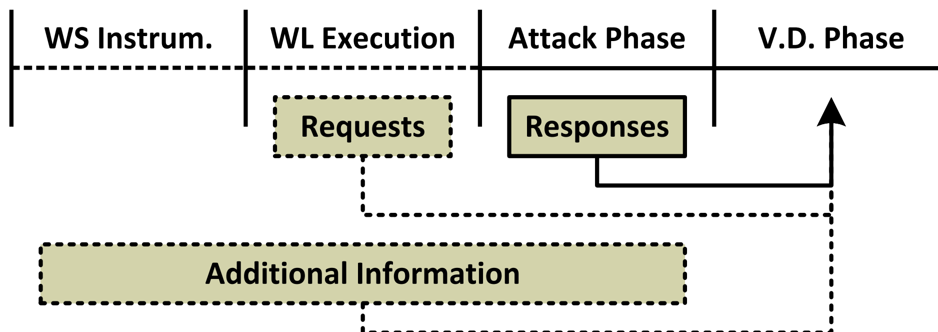
Obviously, the driver instrumentation technique is more intrusive than the other two, but the modifications can be easily done outside the core of the applications being tested. In fact, it is possible to create an instrumented version of a specific driver that can then be used by different applications. Nevertheless, one must be extremely careful in order not to introduce bugs in the instrumented driver during this process. On the other hand, although network packet sniffing is the least intrusive, it is the most difficult to implement: it needs to filter the packets and reconstruct the information that to be monitored. Finally, using proxies is less intrusive than applying driver instrumentation, and less complex to implement than network packet sniffing. However, it still requires the development of different proxies for different technologies.

The **service monitor** (SM) component may be composed of three components: **service instrumentation** (SI), an optional component in charge of instrumenting the service under testing, as needed; **information collector** (IC), responsible for collecting the information during the execution of the tests; and **information manager** (IM), responsible for storing the collected information in a database and for providing that information to the vulnerability detector component when required.

3.2.4 Vulnerability Detector

The **vulnerability detector** (VD) is in charge of processing and correlating the information collected to detect vulnerabilities. This component is probably the most critical one and should be able to identify as much vulnerabilities as possible (based on the available information), while minimizing the number of false positives reported. As mentioned before, in addition to the traditional analysis of requests and responses (applied in classical penetration testing), vulnerability detection can be based on more elaborated approaches such as interface monitoring (Antunes and Vieira 2011), anomaly detection (Antunes et al. 2009a), etc. (what is important is to apply techniques that adequately take advantage of the available information). The vulnerability detector is also the component responsible for managing all the tests by implementing the testing procedure, which is achieved by orchestrating the components presented before. This way, two elements should be included in the detector: the **vulnerability identifier** (VI) and the **testing driver** (TD). We propose to keep these two elements inside the same component as the testing procedure and the vulnerability detection approach greatly influence each other and are highly dependent on the vulnerability detection technique being implemented.

A key aspect is that the testing procedure should be as standard as possible in order to guarantee exhaustive testing and high vulnerability detection coverage. Although such procedure depends on the specificities of the detection technique, in Figure 3.4 we present an overview of the generic approach we propose. As shown, the procedure includes four phases: 1) web service instrumentation, 2) workload execution, 3) attack, and 4) vulnerability detection.



Legend:

Information used in vulnerability detection

---- Optional elements or connections

Figure 3.4 – Proposed generic testing procedure.

Procedure and flow of information used in the detection phase. Different information is used by different techniques.

In the **web service instrumentation phase** the service monitor component is asked to instrument the web service under testing in a way that allows gathering the required information. Obviously, as service instrumentation may not be required in some techniques, this phase is optional. For example, in the case of classical penetration testing, the only information needed is web service requests and responses, whose collection does not require any particular instrumentation (this information is automatically provided by the workload injector and by the attack injector).

The **workload execution phase** consists of generating and submitting the workload requests. This phase is also optional as running a workload may not be needed in some cases. For example, classical penetration testing does not require the execution of the workload, but only the injection of the attacks. On the other hand, more advanced techniques need to learn the behavior of the applications during the workload execution for later detecting vulnerabilities to attacks that otherwise would not be detectable (e.g. blind injection attacks (W. G. Halfond, Viegas, and Orso 2006)). In practice, the workload execution phase consists of using the workload generator component for generating requests and the workload injector component for submitting them. The goal is to allow the service monitor component to gather information on the behavior of the web service in the absence of attacks.

During the **attack phase** the attacks are generated and submitted to the web service. In practice, it consists of using the attacks generator component for generating attacks and the attacks injector component for submitting those attacks. During this process, the service monitor gathers information about the behavior of the web service. This information, combined with the one collected during the workload execution phase, should then be used during the **vulnerability detection phase** to identify vulnerabilities.

3.3 Integrated Approach for Vulnerability Detection

Although the problem of testing services for security has been addressed in the past (see Chapter 2), most of the existing works **disregard key characteristics of service-based environments**. In fact, as discussed in Section 3.1, several challenges are raised when considering security testing in this context (that are not addressed by traditional techniques): SOAs are dynamic in nature, facing changes in the services used and in the way they interact; services are usually under the control of multiple providers, thus the users of the testing tool may have different types of access to the services to be tested; and, the service under testing may interact with other services and resources, thus testing only the interface between the external users and the service is not sufficient.

To address these problems, this section proposes a generic integrated approach for testing service-based infrastructures for vulnerabilities (an instance of this approach

is presented in Chapter 5, including implementation details). Designed in a modular manner, the approach can be easily implemented to support multiple types of software services and security vulnerabilities. In practice, continuous interface monitoring copes with the dynamicity of these environments allowing automatically discovering the existing services, resources and interactions. This allows creating a map of the architecture, which is required for extensively testing the overall infrastructure. The approach considers the use of different testing approaches (that should be developed following the approach proposed in Section 3.2), depending on the level of access to each service (i.e. considering the scenarios defined in Section 3.1.3).

The proposed approach is depicted in Figure 3.5, being based in three key generic steps: 1) Architecture Description, 3) Profiling Interactions and 3) Testing Services. The following paragraphs detail each of these steps.

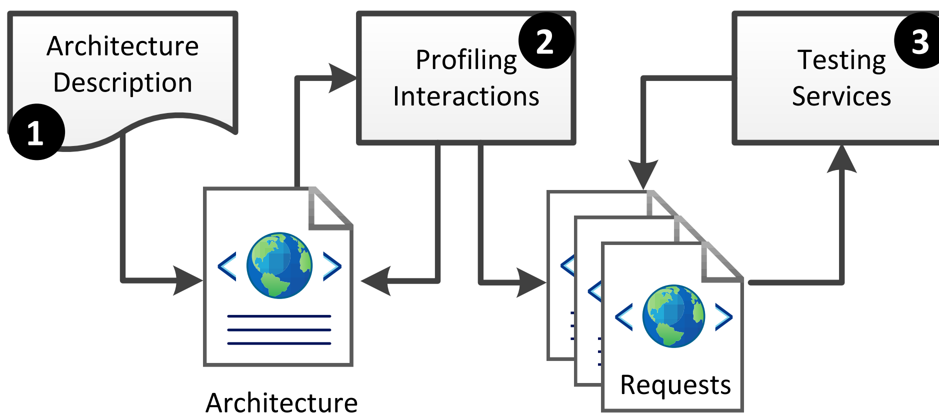


Figure 3.5 – Generic steps of the vulnerability detection approach.
Sequence of high level steps that may be implemented in different ways and using very different technologies.

The process starts with a preliminary architecture description (phase 1: **Architecture Description**). In this phase, the user must provide information about, at least, the services that act as entry points for accessing the system (these are typically under control services that are available to be used from an user point-of-view). However, when possible, he must also provide information about other known services and resources, as well as the relations between them (Service-to-Service and Service-to-Resource). Also, if viable, the user should provide additional information about the valid domains for each parameter, including domain dependencies among parameters and the definition of parameters based on complex data types (composed by a set of individual parameters). Note that, for service operations with several input parameters, the valid domain for a given parameter may be dependent on the value specified for another parameter. This information is important to

improve the quality of the generated workloads and if not provided may lead to workloads that are not able to exercise effectively the code of the services (i.e. have low code coverage).

After the input of base information about the services and resources and their locations, automated pre-processing should be performed to discover gather information. During the process, the user may be asked for complementary information for services (e.g. to define which services are under control, or not) and for resources (e.g. to specify the type of the existing resources). The *Architecture* is thus a key element to support the vulnerability testing process. In practice, it consists of a data structure capable of persistently store information mapping the complete infrastructure, and can be implemented in many different ways, such as metadata database, xml files, properties files, or some proprietary binary format (e.g. Java object serialization format (Gosling et al. 2005)). A key aspect is that it should be continuously updated as the service-based infrastructure evolves.

Using the definitions obtained in the previous phase, a set of profiling interactions should be generated and executed (phase 2: **Profiling Interactions**). The importance of these interactions is twofold. First, they allow discovering new resources and services. Second, they are used to gather information for the training phase of the vulnerability detection techniques that require such training. In practice, profiling interactions consist of running a set of workloads, one for each of the services already mapped in the architecture.

A workload is a set of invocations to the service that simulate valid and non-malicious usage. As mentioned in Section 0, these invocations must be generated for each service in the infrastructure, as it is not possible to define a generic workload that fits all services. Due to the automated nature of the process, a practical option is to use a random workload generator. Obviously, the main problem of the random workload generation approach is that the representativeness of the interactions is not guaranteed. However, defining correctly the input domains of each service allows generating more targeted workloads that may improve the effectiveness of this phase. Nevertheless, easily integrating components that implement other workload generation approaches, including real applications, should be possible.

The workloads are progressively submitted to the services, exercising the infrastructure. For each new service discovered the respective workload must be created. In the case of services that are totally or partially under control, probes should be deployed for interface monitoring before submitting the workload. These probes will monitor the interface activity to detect interactions with other (still not mapped) resources and services. Similarly to the case of the monitor component presented in Section 3.2.3, there are multiple alternatives to implement these probes, including: network packet sniffing, use of proxies and driver instrumentation.

The profiling process is finished when no more services are discovered. Afterwards, in the testing phase (phase 3: **Testing Services**), the process proceeds to test the

services in order to detect vulnerabilities. This should take advantage of multiple vulnerability detection techniques, implemented following the generic design proposed in Section 3.2. As different techniques require different types of information to implement the vulnerability detection process (i.e. several testing scenarios exist, as discussed in Section 3.1.3), during this phase the knowledge about the types of access and information provided by the user is used to rank the applicable detection techniques and selected the ones to apply to each service. Depending on the configuration, one or more of the highest ranked techniques may be used. In other words, each service should be tested using the most effective techniques according to its testing scenario. An important aspect is that when more than one technique is used to detect vulnerabilities, it is necessary to deal with contradictory results, which may appear as different tools frequently report distinct vulnerabilities for the same piece of code, including tools that implement the same detection approach (Vieira, Antunes, and Madeira 2009; Antunes and Vieira 2009a).

Although unlikely, new services may be found during the testing process (most of the services are discovered during profiling phase). These services should be included in the architecture to be also profiled and tested. Most probably these will be within-reach, as services under control have a high probability of being detected during the profiling phase. The testing process should finish when all the services are tested.

To **use this approach at runtime**, besides the aforementioned difficulties of dealing with tests in services that are running (e.g. due to failure propagation), it is necessary to return periodically to the profiling phase to deal with updates in the services and new services being used. It is also necessary to detect services that are decommissioned or are not used anymore in the infrastructure (the deployed probes should help identifying services that are not used anymore).

3.4 Conclusion

This chapter established the framework for the detection of software security vulnerabilities in service-based infrastructures. It presented the specificities of these environments and discussed the challenges that motivate us to propose such a vulnerability detection framework. This framework answers to the requirements of testing such infrastructures and is based on two key ideas.

The first is a **standardized and consistent approach to design vulnerability detection tools** targeting web services. This includes the architecture of such tool, the generic approach, and a set of well-defined components. The approach provides an integrated support for developing innovative and more effective tools whose modularity allows iterative improvements simply by upgrading each module by improved versions of themselves.

The second idea is an **integrated approach that is able to continuously monitor and test the infrastructure**, discovering services and resources in an automated way and testing them for security vulnerabilities. This integrated approach uses tools designed using the proposed approach for vulnerability detection (although it is possible to use other tools, by writing the required interface adapters). This way, has the tools are improved or new tools are added, the effectiveness of the integrated approach also increases.

The proposed framework is generic and concrete instantiations of the components are presented ahead in this thesis. Chapter 4 presents three different techniques for detecting injection vulnerabilities in SOAP web services, designed using the generic approach proposed in Section 3.2. These techniques are later used in Chapter 5 to develop an instantiation of the integrated approach proposed in Section 3.3, focusing on infrastructures supported by SOAP web services and targeting the detection of injection vulnerabilities.

An aspect that should be emphasized is that using different techniques for vulnerability detection raises several questions about how effective these techniques are and how can we select the best techniques for each scenario. This introduces the need for techniques to assess and compare vulnerability detection tools. This way, Chapter 6 addresses this problem by proposing benchmarking approaches for automated vulnerability detection tools. In the same way performance benchmarks have contributed to improve the performance of systems, we believe that this can help to foster the state of the art of vulnerability detection tools for services.

Chapter 4
Techniques for Detecting Injection Vulnerabilities in Web Services

Vulnerability detection tools are a key instrument for development teams to test their services, but research and practice show that state-of-the-art vulnerability detection tools frequently present low effectiveness both in terms of vulnerability coverage and false positive rates. This low effectiveness shows the need for **innovative techniques** that take into account the different access conditions to the services under testing. Furthermore, as presented in Chapter 3, vulnerability detection techniques should implement a generic procedure that supports the design of standardized and modular tools and that provide improved efficiency. By implementing a tool based on multiple components (each with a specific purpose) in a decoupled manner it is easier to later improve the tools.

With the high diversity of web services, types of vulnerabilities, and vulnerability detection approaches available, designing effective techniques requires focusing on well-defined domains. In fact, the division of the spectrum into well-defined areas allows making the right decisions regarding the definition and design of the components and procedure. As mentioned previously, in this context the **definition of the technique domain** includes selecting the class of web services, the, types of vulnerabilities, and the vulnerability detection approaches (see Section 3.2).

This chapter presents the design of **three vulnerability detection techniques that implement the generic approach and components** presented in the in Section 3.2. Regarding the domain of these techniques, the class of web services targeted is SOAP web services, while the type of vulnerabilities to be detected are in the Injection class. The vulnerability detection approach, however, varies from technique

to technique. It is important to highlight that each technique is presented individually and that the integrated approach will be discussed in Chapter 5. The experimental evaluation of tools that implement the proposed techniques is presented and discussed together with the case studies in Chapter 7.

Due to the importance of penetration testing in service-based environments, particularly for the case of services within reach (see testing scenario 3 in Section 3.1.3) and the clear limitations of existing tools (Vieira, Antunes, and Madeira 2009), we first present an improved penetration testing technique. The limitation of this technique is that, although the testing approach is based on the execution of the code and on extensive workloads and attacks, the vulnerability detection process still consists on the analysis of the web services responses, which limits the visibility on the internal behavior of the service.

To overcome the penetration testing limitations, we then present an alternative technique that implements a detection approach based on attack signatures and interface monitoring (Antunes and Vieira 2011). The approach goes further by monitoring the interfaces between the web service and the resources, which allows using additional information about the use of the resources related to the vulnerabilities, achieving higher effectiveness. This tool is particularly useful for the case of services partially under control (see testing scenario 2 in Section 3.1.3).

The last technique proposed aims at achieving better results by analyzing the internal behavior of the web service. Vulnerability detection is based on a runtime anomaly detection approach, which exercises the web service for profiling its regular internal behavior (learning phase) and then attacks the service (attacking phase), reporting a vulnerability when some deviation is detected. Comparing with the aforementioned approaches, it is able to achieve better results due to the added knowledge about the internal behavior of the application. This tool can only be used in the case of services under control (see testing scenario 1 in Section 3.1.3).

The outline of this chapter is as follows. The next section presents the improved penetration testing technique, in particular the generic components (some of them used in the other techniques) and procedure and the specific vulnerability detection mechanism. Section 4.2 presents the attack signatures and interface monitoring technique, emphasizing the new modules (compared to the technique presented in Section 4.1) and the vulnerability detection approach. Section 4.3 presents the technique based on runtime anomaly detection, focusing on the innovative components and on the specific process for vulnerability detection. Finally, Section 4.4 concludes the chapter.

4.1 Improved Penetration Testing [IPT-WS]

Penetration testing is nowadays the technique most used by web developers to detect vulnerabilities in their applications and services. It consists of stressing the application from the point of view of the attacker using a black-box approach, trying to penetrate it by issuing a huge amount of tampered interactions (Stuttard and Pinto 2007). This technique assumes particular relevance in the web services environment, as many times clients and providers need to test services without having access to the source code (e.g. when testing third-party services), which prevents the use of more effective techniques that require that access.

The technique proposed in this section targets the detection of injection vulnerabilities, particularly for services that are within reach but not under control of a provider (i.e. Scenario 1 presented in Section 3.1.3). Comparing to existing web vulnerability scanners based on penetration testing, our approach has three key improvements:

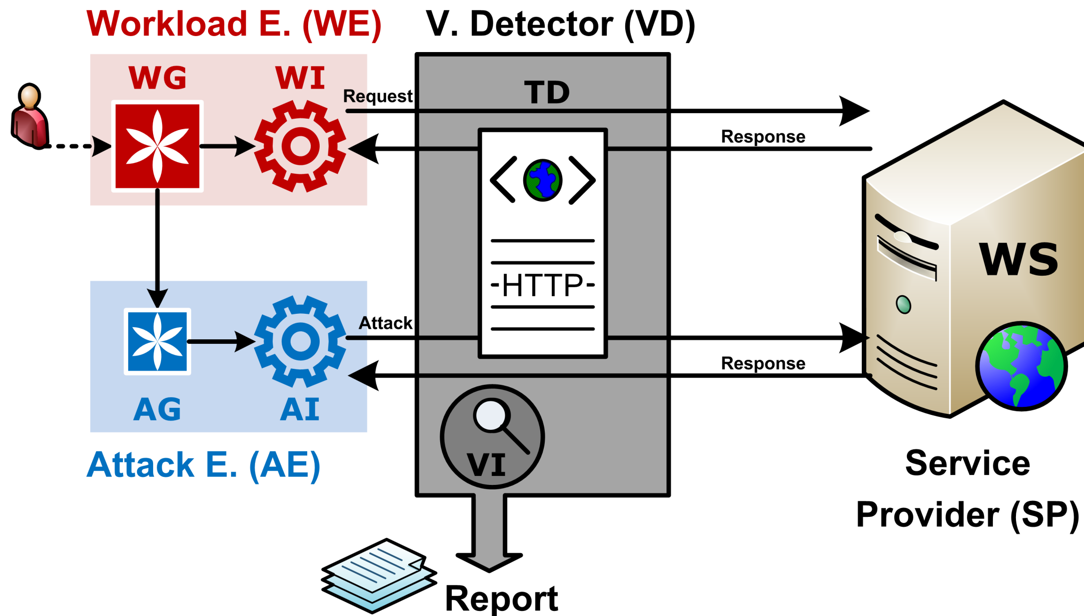
1. It uses a representative workload to exercise the services and understand the expected behavior (i.e. the typical responses in the presence of valid inputs);
2. It uses a more complete set of attacks. The attacks considered are a compilation of all the attacks performed by a large set of scanners plus many attack methods that can be found in the literature; and
3. It applies well-defined rules to analyze the web services responses in order to improve coverage and reduce false positives. These rules include comparing the responses obtained when using malicious inputs with the normal responses (i.e. responses in the presence of a valid workload) and with the responses from classical robustness tests (Vieira, Laranjeiro, and Madeira 2007).

Figure 4.1 presents the overall design of the technique. Obviously, a service monitor component is not represented as in penetration testing the monitoring consists simply in collecting web services requests and responses, which are directly provided by the workload emulator and by the attacks emulator. Also, information about the services and resources that used is not accessible, thus they are not represented. The represented modules are detailed in the following subsections.

4.1.1 Workload Emulation

For generating the workload the technique automatically reads the web service definitions (i.e. operations, return values, parameters, data types, and domains) from the WSDL file. As the valid values for each parameter (i.e. the domain restrictions of the parameter) may not be available, the user is allowed to provide additional information about the valid domains (including for parameters based on complex

data types, which are decomposed in a set of individual parameters). Table 4.1 shows an example of how the user should specify the domains for an example web service named *ValidateService* that provides the following operation to the clients: *ValidateObject (String name, String date, String trackingNumber, int number)*.



Legend:

WE	Workload Emulator	
		User input
	WG	Workload generator
	WI	Workload injector
AE	Attack Emulator	
	AG	Attack generator
	AI	Attack injector
VD	Vulnerability detection module	
	4.1.2	TD Testing driver
	VI	Vulnerability identifier
R	Vulnerabilities detected	
SP	Service Provider	
	WS	Web Service under test
---	Optional elements or connections	

Figure 4.1 – Overall design the improved penetration testing technique.
This technique targets Injection vulnerabilities in web services.
Being an approach based on penetration testing, some of the modules depicted in Figure 3.2 are not applicable, thus they are not presented.

Table 4.1 – Example of the domain specification for each parameter.
Example parameters that represent different data types and domains. For instance, the trackingNumber must respect the pattern shown in the third row in the table.

Service Operation and Parameter	Parameter Domain Definition

`ValidateObject.name`	`Type: String`
	`Min Length: 3`
	`Max Length: 15`

`ValidateObject.date`	`Type: Date` `Format: YYYY/MM/DD`
`ValidateObject.trackingNumber`	`Type: String` `Pattern: \u{2} \d{4} \d{4} \d \u{2}`

`ValidateObject.number`	`Type: Integer`
	`Min Value: 100000000`
	`Max Value: 999999999`

Two options are available for generating the workload. The first option is to use a **user-defined workload**. In this case, the user should implement a workload emulation component to be integrated in our testing technique (the tool allows to load the workload from *xml* files that respect a certain format; also, an API is provided to allow the user to program an adaptor). To simplify the implementation of the workload generator there are several easy to use client emulation tools like soapUI (eviware 2008) that can be used. The second option is to use the **random Workload Generator** (WG) provided, which is able to generate a workload automatically by performing the following steps:

1. **Generate test values for each input parameter**: using the web service definitions mentioned above, the technique generates randomly a set of valid input values (i.e. values in the parameter domain specified by the user). The number of test values to be generated is also defined by the user.

2. **Generate test calls for each operation**: the technique creates a large set of calls for each operation. This consists in the sum of all combinations of the test values generated for all the parameters. For example, take an operation with 5 parameters (p) and 10 test values (v) for each parameter. The total number of test calls is 100000 (i.e. v^p).

3. **Select test calls for each operation**: as it may be unfeasible to use a workload based on all the test calls generated (e.g. due to time constraints), the technique is able to randomly select a subset of the calls. Obviously, it is up to the user to specify the size of this subset, which determines the final size of the workload to be applied during the tests.

Note that, the main problem of the random workload generation approach is that the representativeness of the web service calls is not guaranteed (although our technique allows using workloads of different sizes and randomly generated values are

enough in most cases). Thus, this approach should be used only if the user-defined workload approach is not possible. Obviously, replacing the workload generator by more advanced workload generation approaches (as discussed in Section 3.2) allows easily improving the effectiveness of the technique. After the generation process, the workload injector (WI) executes the workload to gather information about the web service typical responses (i.e. responses obtained without injecting attacks).

4.1.3 Attack Emulation

The IPT-WS technique includes an **attack generator (AG) module** that takes the workload and replaces valid values by malicious values one parameter at a time. This replacement follows an extensive set of mutation rules that is based on the compilation of the attacks used by a large set of scanners (three commercial: Acunetix Web Vulnerability Scanner (Acunetix 2008a), IBM Rational AppScan (IBM 2008), HP WebInspect (HP 2008), and two open source: Foundstone WSDigger (Foundstone, Inc. 2005), and wsfuzzer (OWASP Foundation 2008)). This list was analyzed and complemented based on practical experience and using information on injection methods available in the literature (e.g. (Jensen et al. 2007; Stuttard and Pinto 2007; Shema 2010; W. G. Halfond, Viegas, and Orso 2006)). The final list includes 137 attack types (see Table 3.1 for examples and (Antunes 2013) for the complete list).

The number of attacks to be performed can be extremely huge and depends on the size of the workload considered and on the number of mutation rules to be applied. Take for example a web service with 3 operations with 5 parameters each and a workload with 25 test calls per operation. Applying all the attack types (137) over the entire set of test calls (25) for every parameter (5) of each operation (3) would end up representing 51375 (137 x 25 x 5 x 3) web service executions. Depending on the time available this may be unfeasible. This way, the technique allows the user to specify the number of test calls from the original workload that should be used for the attack load generation. For this, the original test calls are ranked based on their ability to help us detecting vulnerabilities and then a subset is selected. In practice, ranking is built using the following rules:

1. Test calls that during Phase 1 leaded to valid web service responses (i.e. no exception, no server error, and no SOAP error) are in the top of the list.
2. Test calls that during Phase 1 leaded to web service exceptions are in second place.
3. Test calls that during Phase 1 leaded to server errors (e.g. HTTP errors in the 400 and 500 intervals) are in third place.

4. Test calls that during Phase 1 leaded to client-side errors (e.g. SOAP exceptions) are in the bottom of the list (used only as last resource).

After the generation process, the attacks are submitted to the web service and the responses are collected. This (attacks and corresponding responses) together with the information gathered during the execution of the workload (valid requests and corresponding responses) provide the support for the vulnerability detection phase.

4.1.4 Vulnerability Detection

The **VD** module is in charge of processing and correlating the information collected to detect vulnerabilities. As proposed in the generic approach (see Section 3.2), two elements are included in the detector: the **vulnerability identifier** (VI) and the **testing driver** (TD). However, some of the (optional) elements represented in Figure 3.4 are not needed in the case of IPW-WS. In fact, as this technique is based on penetration testing, no instrumentation phase is required and no additional information is available besides the analysis of requests and responses. This way, the resulting procedure includes three phases: 1) workload execution phase, which consists of generating and submitting the workload requests as specified in 4.1.1; 2) attack phase, in which the attacks are generated and submitted to the web service as detailed in 4.1.3; and finally 3) vulnerability detection phase, where the information collected is used to identify vulnerabilities, as detailed in the next paragraphs.

By analyzing the responses obtained during the workload and attacks execution, together with the application of well-defined rules, makes it possible to identify vulnerabilities and exclude potential false positives. This is a crucial step to achieve high coverage low false positive rates. The proposed process is depicted in Figure 4.2.

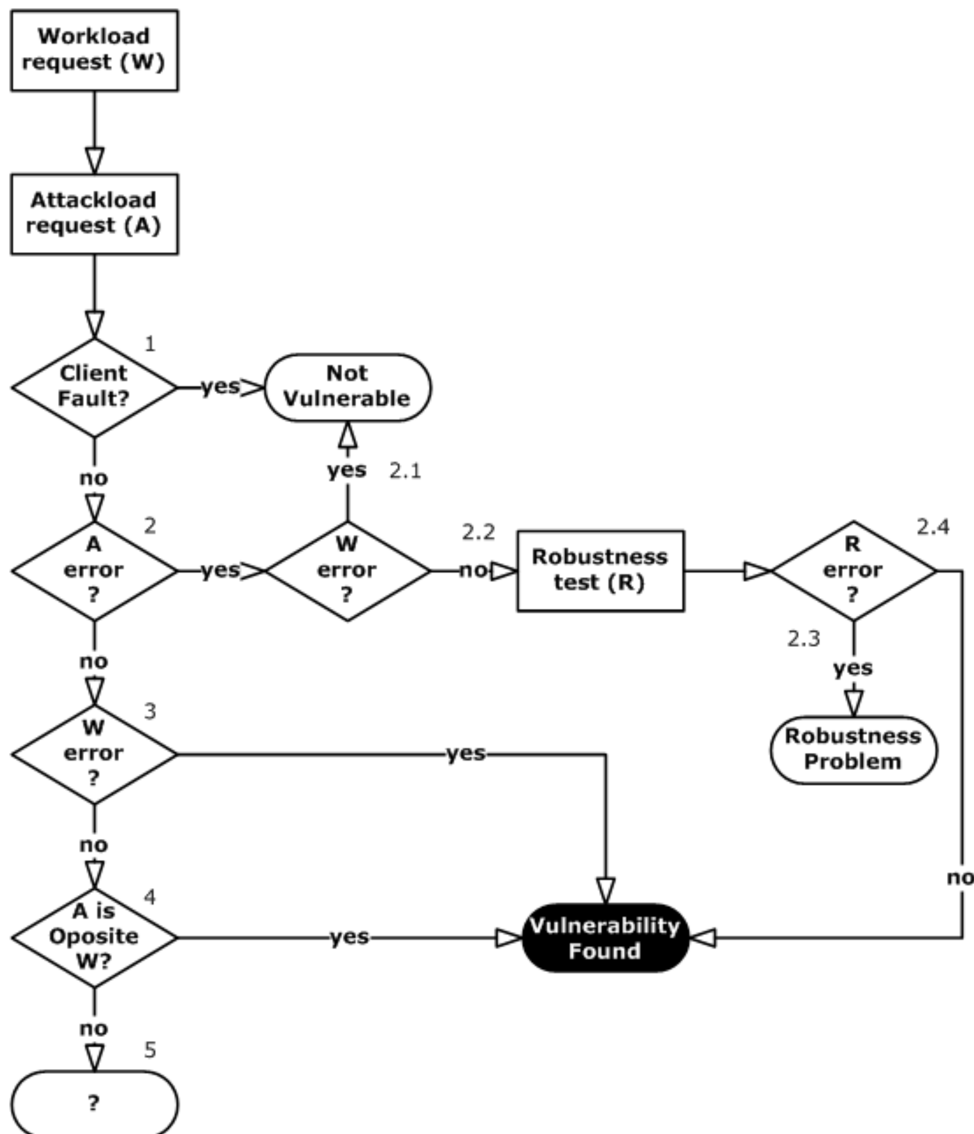


Figure 4.2 – Web service response analysis.
Specific vulnerability detection procedure for finding the maximum number of vulnerabilities while reducing the false positives reported.

The following points describe the figure and the steps performed to analyze the results for each of the attacks performed over each parameter:

1. If the response obtained is a client side error (e.g. SOAP stack error) then the attack was not successful and no vulnerability was explored, as the web service code was not even executed.
2. Otherwise, if the response is an error then
   2.1. ✓If the response for the original test call (before being mutated with malicious values) was the same then **the error is not due to the attack**, but to another characteristic of the service (e.g. a software bug or database server problem).

2.2. Otherwise, apply robustness testing over the parameter, as proposed in (Vieira, Laranjeiro, and Madeira 2007). Creating a robustness test is very similar to the creation of an attack. However, instead of a malicious input, the attack injector uses a non-malicious invalid input (i.e. values outside the expected input domain). The Testing Driver module is in charge of asking the Attack Injector to generate and execute this request. The technique includes a predefined set of values for each possible parameter domain (see details in (Vieira, Laranjeiro, and Madeira 2007)).

2.3. ✓If the responses obtained during robustness testing include the same error then ***the error obtained is due to a robustness problem and not to a vulnerability.***

2.4. ×Otherwise, *an* ***injection vulnerability exists*** as the attack leaded to invalid responses that could not be observed when using a valid workload or when applying robustness testing. This means that the invalid response is caused by the attack, and not by a value out of the target parameter's domain. This is a strong symptom of the existence of a vulnerability.

3. ×Otherwise (i.e. a valid response in the presence of the attack), if the execution of the original test call (before being mutated with malicious values) leaded to a database error, server error or web service exception then **an injection vulnerability has been detected**, as the attack was able to exercise parts of the service that were not possible to execute when using a valid workload. An example of this situation is when an attack is able to circumvent an authentication mechanism that was preventing valid test calls from proceeding.

4. ×Otherwise, if the response obtained in the presence of the attack is the opposite of the response obtained for the original workload call (before being mutated with malicious values) then **an injection vulnerability has been detected**, as the attack leaded to the successful execution of the operation, which was not the case when using the valid workload. Take for example an operation that performs a database modification and only returns a value indicating the success or nonsuccess of the modification. If and attack is able to circumvent an authentication mechanism that was preventing valid test calls from proceeding, then there is a security vulnerability.

5. ✓Otherwise, **no vulnerability** was found. The attack did not change the web service behavior in a visible manner.

The most difficult step of the algorithm is Step 4. In fact, it is quite difficult to establish if a response obtained in the presence of an attack is the opposite of the response obtained for the original test call. The rules must be simple and applied to data types that allow determining the opposition between the result of a valid request and an attack. This way we propose the set of rules presented in Table 4.2 to make these decisions. Obviously, these rules may be a source of false positives, but the user of the technique can easily integrate new rules.

As in classical penetration testing, the proposed IPT-WS is based on the effective execution of the code, and as the service is tested from the user point of view, there is no need to access or modify the source code (which many times is not even available), for instance when testing third party web services. The main problem is that, in practice, vulnerability identification can only rely on the analysis of the web services output to the client. This way, the **effectiveness of the proposed penetration testing approach is still limited by the lack of visibility** on the internal behavior of the service.

Table 4.2 – Rules for the analysis of opposite responses.
Simple and effective rules that allow the technique to easily find if one value is the opposite of another.

Original Response	Opposite Response
False	True
Fail	Success
0	1
Empty list or array	List or array with values
0	GUID data type
Error	No Error
Invalid	Valid

4.2 Attack Signatures and Interface Monitoring [Sign-WS]

To tackle the limitations of penetration testing, this section presents the design of a technique named Sign-WS that uses **attack signatures and interface monitoring for the detection of injection vulnerabilities**. This technique targets the detection of Injection vulnerabilities in services partially under control (Scenario 2 defined in Section 3.1.3).

The goal is to improve the testing process by providing enhanced visibility, yet without needing to access or modify the web services code. The **key assumption** is that most injection attacks manifest, in some way, in the **interfaces between the attacked web service and other systems** (e.g. database, operating system, gateways) and services (e.g. other web services in a service composition). For example, a successful SQL Injection attack leads the service to send malicious SQL queries to the database. Thus, these attacks can be observed in the SQL interface between the service and the database server. Similarly, XPath Injection attacks (on top of XPath or XQuery) manifest at the interface with XML files (OWASP Foundation 2013) and OS Command Injection are visible at the interface with the operation system (Stuttard and Pinto 2007), etc. In practice, the approach targets the vulnerabilities that lead web services to be used as front end for attacking backend resources.

Comparing with traditional penetration testing and with the IPT-WS technique, the Sign-WS technique is more effective as it uses additional information that allows increasing the number of vulnerabilities detected and reduce false positives. For

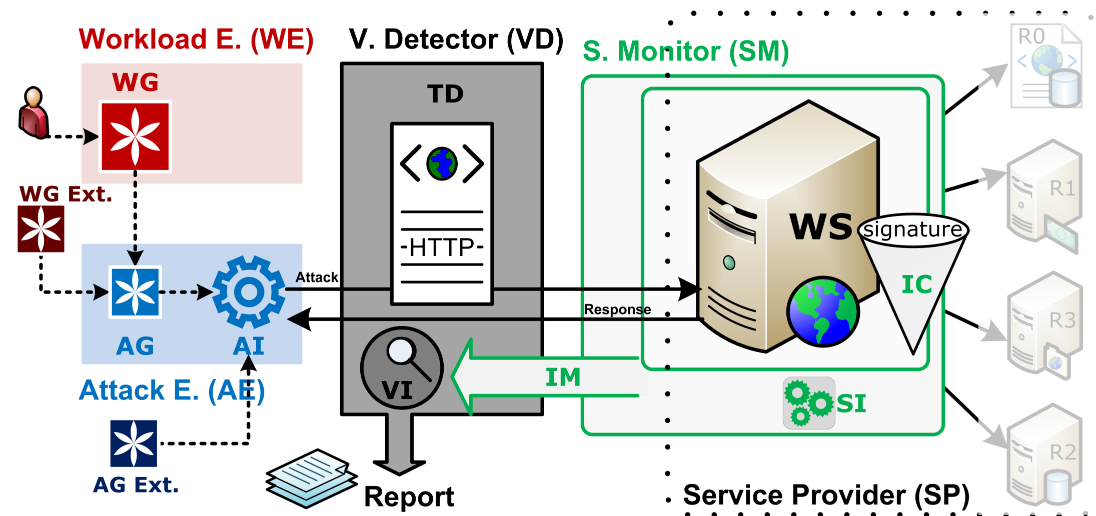
example, blind injection vulnerabilities (that exist when an application is vulnerable, but the results of an attack are not directly visible to the attacker) cannot be detected by traditional penetration testing (or by our IPT-WS technique); however, they can be detected by the Sign-WS approach as they can be observed in the interface between the web services and the resource targeted by the attack. Furthermore, detecting vulnerabilities based on the effects of attacks (e.g. changes in SQL queries), is much more precise than considering only the analysis of the web service output, allowing decreasing the rate of false positives. Figure 4.3 presents the overall design of the technique, which is detailed in the next subsections. Note that the technique does not require a Workload Injector module, as the technique has no need to know the regular behavior of the web service (i.e. how the service behaves under non-malicious requests). The *random workload generator* presented in Section 4.1.1 is used to support the generation of attacks when using the internal attack generator. This way, the workload emulation component is not discussed.

4.2.1 Attack Emulation

Sign-WS supports two options for generating attacks: it includes a specific generation module, similar to the one presented in Section 4.1.3, and also allows the integration of an external technique (e.g. another vulnerability scanner). A key aspect that is common to both cases is that signatures are added to the attacks to later support the detection of vulnerabilities. This way, the concept of attack signatures is transversal to both cases. Also common to both cases is the fact that after the submission of a signed attack, the tool continuously waits for receiving a notification of any signature detected. When it receives such notification, a match is established between the information received and the attacked input. This way, it is possible to precisely identify the attacked input and report it as vulnerable. The next paragraphs present the concept of attack signatures and describe how they are added to the attacks.

Defining Efficient Attack Signatures

In the literature, attack signatures are defined in multiple ways, depending on the type of system studied, but according to (Sabhnani and Serpen 2004) an attack signature is "*a distinctive complex pattern used to detect system penetration, which may involve comparison of audit and log data from a variety of sources within the computing platform or infrastructure*".



Legend:

WE	Workload Emulator	
	(user icon)	User input
	WG	Workload generator
	WG Ext.	External workload generator
AE	Attack Emulator	
	AG	Attack generator
	AI	Attack injector
	AG Ext.	External attack generator
VD	Vulnerability detection module	
	TD	Testing driver
	VI	Vulnerability identifier
R	Vulnerabilities detected	
SM	Service monitor	
	IC	Information collector
	SI	Service instrumentation
	IM	Information manager
SP	Service Provider	
	WS	Web Service under test
R	Resource	
	R0	XML database
	R1	Payment gateway
	R2	Database
	R3	FTP server
---	Optional elements or connections	

Figure 4.3 – Overall design of the Sign-WS technique.
The technique detects injection vulnerabilities in web services using attack signatures and interface monitoring. The resources are faded because they may be outside of the provider's control. However, their interface is known.

In the context of this work, an attack signature is a token that is introduced inside a malicious string (the injection attack) in such way that, if the attack is successful, the token is observable somewhere in the interfaces of the service. For example, in a successful SQL Injection attack, the signature should show up in the manipulated SQL command (the target of the attack), outside any literal string (i.e. as a part of the

actual command), revealing that it is possible for attackers to modify the structure of the command sent to the database server (OWASP Foundation 2013). In this case, the signature is considered **active** (see example in Figure 4.4 (a)). Otherwise, if the signature is placed inside a literal string, it is considered **inactive** and is inoffensive (Figure 4.4 (b)).

```
Select n from t where dsc LIKE '%input' SIGNATURE%';
```

(a)

```
Select n from t where dsc LIKE '%input SIGNATURE%';
```

(b)

Figure 4.4 – Examples of queries with signatures.
(a) the signature is active; (b) the signature is inside a literal string, it is inactive.
The red text represents the signature while the bold signals a literal string.

Defining attack signatures is not easy. On one hand, signing attacks with complex signatures may not be possible due to length limitations of the target commands or parameters, restrictions in terms of the characters that can be used, etc. On the other hand, very simple signatures may raise false positives, as there is the risk of using a signature that matches a valid keyword (the valid keyword would wrongly suggest the presence of the signature). This way, to maximize the success of the approach, the signature token must be:

1. **Unambiguous**: the signature must not be easily confused with the tokens/keywords regularly found in the context of the applications being tested;
2. **Inoffensive**: the signature must not include characters that may be filtered, escaped or refused. Although the goal is the attack to pass the protection mechanisms, the signature token must be harmless;
3. **Informative**: the signature must include information about what is being attacked to later allow the identification of the vulnerable input;
4. **Short:** the token must be as short as possible to avoid problems with limited length fields or protection mechanisms as length validators, which are extremely common in web services.

The proposed signature model is composed of a set of five elements, including two delimiters, two identifiers (that represent the information transported by the signature), and a qualifier (see example in Figure 4.5). The first delimiter (underscore character) marks the beginning of the signature, the first identifier represents the web service operation or resource being tested, the second identifier is the input parameter attacked, and the second delimiter (again an underscore character) marks

the end of the signature information. The qualifier, placed after the second delimiter, indicates whether the signature is applied in normal or reversed mode, as explained below.

This model allows short and informative enough signatures. To reinforce unambiguity, each time a signature is detected in an interface, a confirmation request is submitted, now containing a reversed version of the signature. This is important to decrease the probability of using a signature that, by coincidence, partially matches a part of the target command, also providing a second validation of the vulnerability. As signatures do not include any "special" characters, they do not suffer any transformation due to existing escaping routines, thus assuring inoffensiveness. Obviously, to guarantee portability and allow adapting to different types of web services, the user of the technique is allowed to configure the signature model he wants to apply (using regular expressions).

Figure 4.5 shows the signature model (including the reversed version) used with the default configuration of our technique. When building the signature, digit "`1`" is replaced by the identifier of the web service operation under testing, and digit "`2`" is replaced by the identifier of the input parameter attacked. Each identifier can be a number (10) or a letter (52). The attack injector maintains a dictionary that maps each identifier to their real meaning. Obviously, if the number of operations or input parameters is greater than 62 (it rarely is) then it is necessary to add digits to the signature model described in configuration files, increasing the size of the signature.

`_12_p`	`_21_o`
(a)	(b)

Figure 4.5 – Signature token used. (a) regular token; (b) reversed token.
The regular token respects characteristics that help it to be successful. The reversed token is used to confirm the vulnerabilities detected.

Generating Signed Attacks in the Internal Attack Generator

As presented in Section 4.1.3, the approach implemented by the internal attack generator (AG) consists of mutating the workload requests. In practice, valid values are replaced by the malicious values. Differently from IPT-WS, however, is the fact that Sign-WS adds signatures to the attacks for later supporting the vulnerability identification. In the default configuration, Sign-WS uses a set of attack types that is based on the compilation of the types used by a large set of scanners, complemented with practical experience and information on SQL Injection methods available in the literature (as shown in Section 4.1.3). The final list includes 102 attack types (this list slightly differs from the list used in Section 4.1.3, as in many cases the attacks are not

adequate to be used together with a signature). Some examples, including signatures placeholders, are presented in Table 4.3 and the full list available at (Antunes 2013).

For better understanding the concept, consider the following examples: attack 1) tries to invalidate the query by closing a literal string and exposing the signature. Attack 2) is designed to avoid naive escaping mechanisms using a slash ('\') as protection. Attacks 4) and 5) try to turn a where clause into a tautology (e.g. to circumvent an authentication mechanism), while exposing the signature. Attack 10) is similar to 1) but tries to also include the value of the valid workload to disguise the attack and signature. As shown, there are some tokens representing placeholders to be replaced at runtime. These tokens are useful to define more complex and efficient attacks. The meaning of each of the tokens is as follows:

Table 4.3 – Examples of signed SQL Injection attack types.
These include special placeholders that define where the technique will insert the signature and other important data.

SQL/XPath Injection mutation rules	
1)	`' %SIGNATURE%`
2)	`\' %SIGNATURE%`
3)	`' -- %SIGNATURE%`
4)	`' or 0=0 -- %SIGNATURE%`
5)	`' = '' or %SIGNATURE% = '`
6)	`hi') %SIGNATURE% ('a'='a`
7)	`= %SIGNATURE% || '`
8)	`1' || %SIGNATURE% || '`
9)	`' UNION %SIGNATURE%`
10)	`%WORKLOAD%' %SIGNATURE%`
11)	`%WORKL_%' or %SIGNATURE%=%SIGNATURE% --`

- **`%SIGNATURE%`** – is a placeholder to be replaced by the signature token dynamically generated. Using this, the user can control the specific location of the signature inside the malicious string;
- **`%WORKLOAD%`** – is a placeholder to be replaced at runtime by the value of the input in the original workload request. This is useful because it helps to disguise the attack with valid data, avoiding some weak validators that perform pattern matching. An example is a validator that only accepts a format that consists of a date with some more text ("DD-MM-YYYY (…)"). For an attack to be successful, it needs to contain the date token. If the workload was correctly generated to fit this kind of domain restrictions,

using the workload as a base to generate the attacks increases the chances of generating a successful attack;

- `%WORKL_%` – similar to %WORKLOAD%, but in this case only the initial characters are used in order to maintain the total length of the input. This helps avoiding length validators. For instance, considering a validator that accepts a string with a length between 100 and 150 characters (or even a more restrictive range), if the workload request is valid, then the attack generated using this token is also valid (as we cut from the string the characters needed to introduce the signature without exceeding the limits).

An important point is the use of the apostrophe character (`'`). This character is usually the one that delimits literal strings in commands or queries. However, sometimes other characters have similar functions, for instance the quote character (`"`). This way, the user can define in the configuration file the values to use in that position, and at runtime the technique performs the needed replacements. If multiple values are defined, the technique replicates the attack for each one. For example, during our tests (presented in Chapter 7), four values were used: "`'`", "`"`", "`'`" and "`"`" (the two string delimiters and the correspondent html entities).

The defined attacks cover the majority of the cases, using techniques that try, for instance, to avoid weak escaping mechanisms by combining multiple escaping characters together. However, the user can easily add more attacks using the rules defined above. A key aspect is that, to reveal a vulnerability, the attack does not need to be successful on accessing, modifying or destroying data. It is only required that the attack is able to modify the structure of the backend command. For example, in the case of SQL Injection, what is required is the attack to be able to change the structure of the SQL query in such way that the signature token can be identified in the service interface with the database server as being active. The same is valid for other types of injection vulnerabilities.

Signing attacks generated by an External Attack Generator

As an alternative to the internal attack generator, the technique allows to use a third party tool to generate the attacks that later will be signed. The concepts behind the signatures used are exactly the same as introduced before. Here we focus solely on the way externally generated attacks are intercepted and signed.

For supporting the integration of an external tool to generate attacks, the attack injector intercepts the requests performed by that tool. In practice, all the requests are intercepted, locally stored, and finally forwarded (without any change) to the target web service. The idea consists of later strategically placing a signature close to the attack. The overall architecture of the system is depicted in Figure 4.6.

Figure 4.6 – Mechanism to intercept and sign external requests.
The proxy intercepts the HTTP requests and stores a copy for future use.

As we can observe, an HTTP proxy is used to intercept the requests performed by the penetration testing tool. Regarding the external tool, it is only required that it is configured to use an HTTP proxy, which is allowed by most tools and consists in changing a simple parameter (e.g. add a proxy address and port in the tool settings). Sign-WS uses LittleProxy (LittleShoot 2010) that is an open-source "*high-performance HTTP proxy written in Java*", available online under Apache 2 license. The proxy was modified to intercept all the requests performed by the external testing tool and forward them without any change. In this phase, the requests are not signed, as the goal is to allow the penetration tester to perform its work without any interference. The only interference are the very small delays introduced by the proxy, but this is not considered an important problem as timing issues are not particularly important, especially during the type of security testing targeted by our tool.

After the external testing tool completes the scanning process, all the stored requests are analyzed by the attack injector, searching for attacks (i.e. requests containing malicious input strings). When an attack is found, a signature token is added to the attack string. The signature includes information about the component and input under attack (obtained from the original request) that is also inserted in the dictionary that maps the identifiers to the inputs they represent. The place in the attack string where the signature is inserted can be defined by the user of the technique, in the form of key characters. These **key characters** depend on the type of attack being conducted; for example, in our experiments with SQL Injection we considered the typical characters necessary to launch SQL Injection campaigns: literal string delimiters (`'` and `"`), the equal character (`=`) often used to manipulate

SQL conditions, and the parenthesis characters) and (, used to manipulate or add subqueries. After this, the new request is sent to the web service. If no key character is found in the original request, then it is discarded, as it is not considered an attack (e.g. the external tool is issuing a non malicious request).

4.2.2 Service Monitoring

Simultaneously to the submission of the attacks containing the attack signatures, it is necessary to monitor the interfaces of the application to capture the executed commands (IC component in Figure 4.3). To monitor the interface of the web services there are multiple options depending on the type of interface, including (as detailed in Section 3.2.3): use network packet sniffing, use a proxy, instrument the code, etc. In the particular case of Sign-WS we perform driver instrumentation. In practice, when the interface to be monitored is accessed through a driver (e.g. Java applications use JDBC drivers to access the database server), this driver can be instrumented to include monitoring facilities. Obviously, driver instrumentation is an intrusive technique, but the modifications can be done outside the core of the applications being tested. In fact, it is possible to create an instrumented version of a specific driver that can even be used in different applications.

For example, the Sign-WS implementation targeting SQL Injection attacks we instrumented a JDBC driver using Aspect-Oriented Programming (AOP) (Kiczales et al. 2002), in order to monitor the queries sent to the database. AOP is a well-known programming paradigm that allows injecting crosscutting concerns into any application in a non-intrusive way (Kiczales et al. 2002). The Java Database Connectivity (JDBC) API is designed to access any kind of tabular data, but it is mostly used to access relational databases in Java applications (Reese and Oram 2000). It is the responsibility of each database vendor to provide a library containing the JDBC driver and the implementation of the JDBC API for Java applications to interact with its Database Management System (DBMS).

In practice, AOP was used to transparently intercept the key points inside the JDBC library where the SQL commands are sent to the database server. The result of this process was a new driver library. To use this driver during the web application testing process, what is needed is to refer the modified version in the *classpath* of the application instead of the original one. This is the main reason for our option: it is very practical to use it in different Java based-systems during the experimental evaluation. A similar approach is used for other drivers.

When the signed attacks are submitted to the service, the information collector (IC) gathers information from the web service interfaces. This information is stored by the information manager (IM) in a database (together with the corresponding requests) and later used by the vulnerability detector (VD) to identify vulnerabilities.

4.2.3 Vulnerability Detection

After capturing the commands at the service interfaces, it is necessary to process and analyze them to detect potential vulnerabilities. In practice, when a signature token is found outside a literal string in a command sent to an external resource, this means that there is a vulnerability in the web service. Thus, before applying regular expressions to find signatures, it is necessary to process the data in order to remove the inoffensive parts (e.g. control characters, well-formed literal strings). Figure 4.7 shows an example of the transformations applied to SQL queries during the command processing steps (a similar approach is used for other types of commands).

- **Step 1:** all the correctly escaped slashes (\), apostrophes (`'`) and quotes (`"`) are removed from the string. Obviously, the definition of "correctly escaped" varies according to the type of commands that are being processed.
- **Step 2:** the remaining literal strings identified by the regular expression "`'[^']*'`" are removed. This regular expression represents a collection of characters delimited by two apostrophes that cannot contain apostrophes inside.
- **Step 3:** any attack signature that still remains in the command after this process is considered active, as can be observed in Figure 4.7 (a). In this case a vulnerability is identified, having the associated information: '2' (operation or resource under testing) and '8' (input parameter attacked). This is the information that allows linking the vulnerability to the input that can be used to exploit it. Obviously, the attack signatures should include the information needed to make a correspondence between this information and the inputs of the service under testing.

```
1: Select n from t where dsc LIKE '%input' _28_p%';
2: Select n from t where dsc LIKE '%input' _28_p%';
3: Select n from t where dsc LIKE _28_p%';
```

(a)

```
1: Select n from t where dsc LIKE '%input\' _28_p%';
2: Select n from t where dsc LIKE '%input _28_p%';
3: Select n from t where dsc LIKE ;
```

(b)

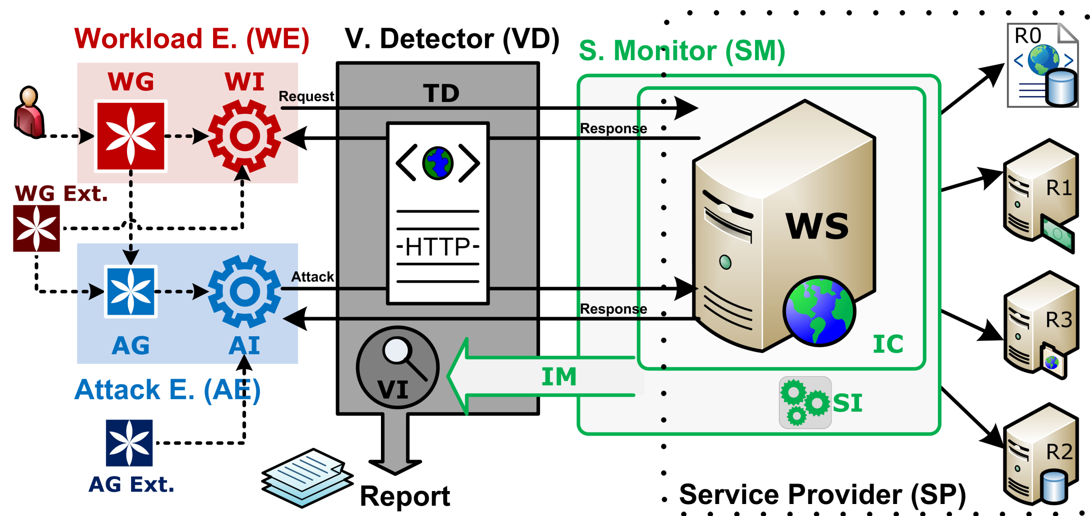
Figure 4.7 – Examples of command processing steps with SQL queries. (a) an active signature is found; (b) no active signature is found. *Transformations applied to each query during the processing. Tokens in `red` are removed in that step. Tokens in* **`bold`** *are active signatures.*

4.3 Runtime Anomaly Detection [RAD-WS]

Although the use of attack signatures and interface monitoring pushes the effectiveness of the Sign-WS technique beyond the limits of penetration testing, it still ignores important information that may be available to the providers of web services. This information is related to the internals of the application and can be effectively used to understand the internal behavior of the web services. To take advantage of this information, this section presents the design of a **runtime anomaly detection technique** for the detection of injection vulnerabilities. The technique targets the detection of Injection vulnerabilities in services under control (Scenario 3 defined in Section 3.1.3), as it is necessary to perform some changes to the services under testing.

The idea is to instrument the web service and then to generate and run a workload to exercise it, while learning the profile of the internal commands issued (e.g. SQL and XPath commands). Afterwards we generate and run a large set of injection attacks and observe the internal commands being executed. By matching the command issued during the attack phase with the profile that was learned during the previous phase, it is possible to detect anomalies that can be reported as vulnerabilities.

Comparing to the approaches proposed before, RAD-WS takes advantage of information about the internal behavior of the service under testing, which allows increasing detection coverage. Figure 4.8 presents the design of the technique components, which are detailed in the next subsections. The Attack Emulator component is not detailed as this technique uses the one implemented by IPT-WS (presented in Section 4.1.3).



Legend:

WE	Workload Emulator	
		User input
	WG	Workload generator
	WI	Workload injector
	WG Ext.	External workload generator
AE	Attack Emulator	
	AG	Attack generator
	AI	Attack injector
	AG Ext.	External attack generator
VD	Vulnerability detection module	
	TD	Testing driver
	VI	Vulnerability identifier
R	Vulnerabilities detected	
SM	Service monitor	
	IC	Information collector
	SI	Service instrumentation
	IM	Information manager
SP	Service Provider	
	WS	Web Service under test
R	Resource	
	R0	XML database
	R1	Payment gateway
	R2	Database
	R3	FTP server
---	Optional elements or connections	

Figure 4.8 – Overall design of the RAD-WS technique.
The technique learns the profile of the regular behavior of the service and then detects injection vulnerabilities in web services by detecting anomalies in this behavior during the attack phase.

4.3.1 Workload Emulation

The workload emulator (WE) module is similar to the one presented for the IPT-WS technique in the Section 4.1.1. In fact, the workload generator (WG) is based on the *random workload generator* presented. However, during the process of executing the workload, a responsibility of the workload injector (WI) module, there is a key part of the process that is substantially different: the Service Monitor must profile the commands executed to latter be used for vulnerability detection.

The workload injector exercises the service by executing the generated workload. During this phase, internal commands (e.g. SQL and XPath commands) are intercepted (using AOP, as presented in Section 4.3.2) and parsed in order to remove the data variant part (if any) and a hash code is generated to uniquely identify each command. In other words, the information used does not represent the exact command text, since commands may differ slightly in different executions, while keeping the same structure. For example, in the case of the SQL command presented in Figure 4.9, the job and the salary in the select criteria are dependent on the user's choices (see line 1 in Figure 4.9). Thus, instead of considering the full command text, we just represent the invariant part of it (see line 3 in Figure 4.9). As shown, the variant parts (i.e. numbers and literal strings) are progressively removed.

```
1: SELECT * from EMP where job like 'CLERK' and SAL > 1000;
2: SELECT * from EMP where job like Str? and SAL > 1000;
3: SELECT * from EMP where job like Str? and SAL > Nbr?;
```

Figure 4.9 – Process to remove the variant parts of an SQL query.
The lines show the transformations applied to each query during the processing. Tokens in red *(and italic) are removed in that step.*

Each hash signature is associated with a source code entry point (which is provided by the AOP) in a Map structure. This does not mean that we need the original application's source code; it just means that we need *bytecode* compiled with source code line information, which is generally the case, even in production applications, as it provides extra information on failure events. In the previously referred Map structure, each key corresponds to a code that includes the source code point and, when available, a part of the stack trace information. Each key is associated with a set of valid/expected hashed commands. Using the stack trace information allows to differentiate information that otherwise would not be available, since the application can use a single piece of specific code to execute all the interactions with database, thus using only the entry source point would be less informative. Note that, as represented in Figure 4.10, in a given point there might be several valid commands (this is why we need a set of valid commands for each source code point).

```
...
if (isInsert()) {
  sql = "INSERT INTO CLIENT VALUES (seq.nextval, Jack')";
} else {
  sql = "UPDATE CLIENT SET NAME='John' WHERE ID=1";
}
statement.execute(sql);
...
```

Figure 4.10 – Example of SQL commands execution.
The actual "sql" executed has two different valid possibilities, depending on the execution.

An important aspect is that the workload must guarantee a minimum level of code coverage (as discussed in Section 0). Although this does not assure a complete learning of internal commands, it allows us to have a high confidence degree. This way, the technique allows easily integrating an external code coverage analysis technique. Additional workload requests are generated if the coverage value is under a given threshold defined by the user.

4.3.2 Service Monitoring

For the web service instrumentation we again use the well-known Aspect-Oriented Programming (AOP) paradigm (Kiczales et al. 2002). In this case it was used to intercept key web service execution points and introduce the vulnerability detection mechanisms.

Vulnerability detection starts by automatically identifying all the locations in the web service code where commands are executed. This is achieved by using AOP to intercept all the calls to methods that belong to APIs used to execute SQL commands (e.g. Java's JDBC, the Spring JDBC, etc.), to evaluate XPath expressions (e.g. Java's JAXP, JaxenXPath, etc.), etc. Virtually any API can be added, as the only requirement is to know the signature of the method to be intercepted. Figure 4.11 represents the basic architecture of the interception mechanism.

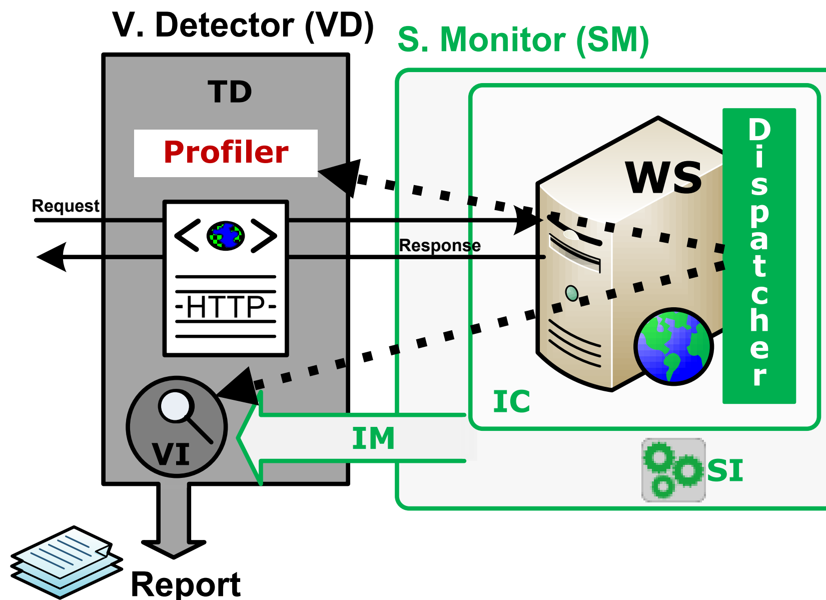


Figure 4.11 – The AOP Based Service Monitoring system.
The Information Collector intercepts the commands and sends them to the Profiler or to the Vulnerability Identifier depending on the phase of the process. The names of the modules are the same as in Figure 4.8.

As we can observe, at runtime the issued SQL commands, XPath queries, etc., are intercepted and delivered to the Information Collector that includes a dispatcher. The decision here is simply to check if the application is in profiling mode or in detection mode and to deliver the request to the appropriate module (i.e. the profiler or vulnerability identifier modules). It is important to emphasize that instrumentation does not change the web service behavior (the code logic is not modified) and that it is only meant for the RAD-WS technique (it is removed after testing).

4.3.3 Vulnerability Detection

To detect vulnerabilities we perform security checks for each data access executed during the attack phase. As mentioned before, all commands (SQL, XPath, etc.) are intercepted (using AOP), hashed, and stored during that phase. These are compared to the values of the learned valid commands for the code point at which the command was submitted. In practice, the matching process consists of looking up the current source code origin in the previously referred Map structure and getting the set of hash codes of the valid (learned) commands for that point. This set (generally quite small) is then searched for an element that exactly matches the hash

of the command that is being executed. If a match is not found, the occurrence (i.e. the potential vulnerability) is logged for future reference.

The log entry includes a reference to the code location where the vulnerability was detected, the query that was executed in the presence of the attack, and information about the operation input values, namely the attacked parameter and the attack value. If the source code origin is not found in the Map lookup, the log indicates that the line was not learned. This case indicates that the learning phase is incomplete (coverage was not good enough) and that a more exhaustive workload is required. Note that the lines that have not been learned provide indications on how to improve the workload to increase coverage.

4.4 Conclusion

This chapter presented the design of three vulnerability detection techniques that implement the generic approach and components presented in the in Chapter 3. The domain of these techniques is SOAP web services and Injection vulnerabilities. The vulnerability detection approaches are respectively: improved penetration testing, attack signatures and interface monitoring, and runtime anomaly detection. For each of these techniques the generic design was instantiated and the generic components and procedure were defined. These tools use different parts of the generic architecture, showing the versatility of the approach.

The **improved penetration testing** uses representative workloads to exercise the web services, implements effective attackloads, and applies well-defined rules to analyze the web services responses, thus improving detection coverage while reducing false positives (compared to traditional penetration testing). The **attack signatures and interface monitoring approach** overcomes the visibility limitations of penetration testing by introducing special tokens inside the injection attacks and then monitoring the interfaces of the service under testing looking for these tokens to detect vulnerabilities. Finally, the **runtime anomaly detection approach** exercises the service for profiling its regular internal behavior and then attacks the service, reporting a vulnerability when some deviation is detected. This grasp on the internal behavior of the application allows going even further in terms of vulnerability detection coverage and false positive avoidance.

It is important to note that each technique presented targets the individual testing of web services while the integrated approach focusing on service-based infrastructures is presented in Chapter 5. The experimental evaluation results are discussed in Chapter 7, which shows the potential of the proposed techniques, when compared to the current state-of-the-art.

Chapter 5
Integrated Tool for Detecting Vulnerabilities in Service-Based Infrastructures

Although the problem of testing services for security has been addressed in the past (see Chapter 2 for related work and Chapter 4 for proposed techniques), most works **disregard the specific challenges raised by service-based environments,** as discussed in Section 3.1. The integrated approach proposed in Section 3.3 is a solution towards this problem. In short, the approach is based on continuous monitoring to discover the services, resources and interactions, which allows coping with the dynamicity of these environments. It is based on three generic steps: architecture description, profiling interactions, and testing services. The key idea is to take advantage of multiple vulnerability detection tools to test the web services depending on the level of access and information available about each service.

This chapter presents an extensible tool, named **SOA-Scanner**, that instantiates the approach presented Section 3.3. The tool is extensible in the sense that it follows a modular architecture and can be easily extended to more types of software services and additional kinds of security vulnerabilities, although the implementation described here targets only SOAP web services and injection vulnerabilities. In practice, the SOA-Scanner tool consists of three main components: a centralized controller, a monitoring system, and a set of testing tools.

The **centralized controller** is responsible for the coordination of the monitoring and testing process. During the description of the architecture of the environment, the controller acts as the interface with the user (that should introduce some description information). Next, the controller is responsible for executing the profiling interactions to exercise the web services under testing. Additionally, it centralizes the

flux of information originated by the monitoring system. Finally, it is in charge of assigning the available testing tools to the web services being tested and, after the completion of the testing process, of integrating the results reported.

To analyze the infrastructure we need to collect as much information as possible about the web services under testing; this is the responsibility of **monitoring system**, which afterwards sends the information to the controller. Obviously, the information that can be collected depends on the access conditions to each target service. In practice, the monitoring system consists of a set of probes deployed close to the services under control or partially under control (obviously, in the case of within reach services there is no control or access to deploy the probes).

To test the web services it is necessary to select, from a set of available **testing tools**, the ones that are most suited for the level of access and information available about each service. Three tools implementing the techniques presented in Chapter 4 are included in the SOA-Scanner to cover the three testing scenarios defined in Section 3.1.3. More tools can easily be added, provided that those tools follow the design approach proposed in Section 3.2.

The outline of this chapter is as follows. The next section presents the overall architecture of the tool and explains the role of each component in the process. Section 5.2 presents the centralized controller component and describes its role of coordination the entire process. Section 5.3 presents the monitoring system and details the implementation of the probes that constitute it. Section 5.4 presents the testing tools, discusses how the tools are assigned to the services to be tested and explains how the results are integrated at the end. Finally, Section 5.5 concludes the chapter.

5.1 Architecture of the SOA-Scanner

As proposed in Section 3.3, the **SOA-Scanner** implements three main steps:

1. **Architecture Description**: the tool asks the user to specify the services that act as entry points for the system and, if possible, information about the services under control, including input domains. It is also possible to specify additional services (not under control), resources and the relations among them;
2. **Profiling Interactions**: based on the description provided, the tool issues a set of profiling interactions to discover additional resources and services and to gather complementary information. This is a progressive process that finishes when no more services can be discovered;

3. **Testing Services**: finally, the vulnerability detection phase consists on testing each service using the most effective technique available and also on integrating the reports from the multiple tools in a single one.

The tool is implemented in a modular fashion in order to be easily extended. Figure 5.1 depicts its overall architecture (the relation of the tool with the complete infrastructure is discussed ahead in this section).

As shown, the tool is divided in three main modules, identifiable in the figure:

- **Centralized Controller**: subdivided in three controllers (profiling controller, monitoring controller, and testing controller), it is responsible for the coordination of the process.
- **Monitoring System**: consisting of a set of probes (W), it is responsible for collecting information about the architecture of the infrastructure;
- **Testing Tools:** composed of three tools, and prepared to include additional ones, it is responsible for testing the services to detect vulnerabilities.

Note that some optional elements are represented in Figure 5.1 (elements that are not always applicable). For example, in the case of the workload emulator, the user has the option of replacing the included workload generator by one that is more effective in the context of the services to be tested. Also, the monitoring system (MS) is only applicable in the case of services under control (partially or totally). To better understand this and how the tool interacts with a service-based infrastructure, Figure 5.2 depicts where each component is located considering as context the reference infrastructure presented in Section 3.1. In the case represented, the depicted consumer (C) is an external user of the infrastructure. The user of the SOA-Scanner tool is assumed to be the provider represented by *P0*, and the services totally or partially under control are the ones that the tool can access and in which it is possible to deploy probes (W).

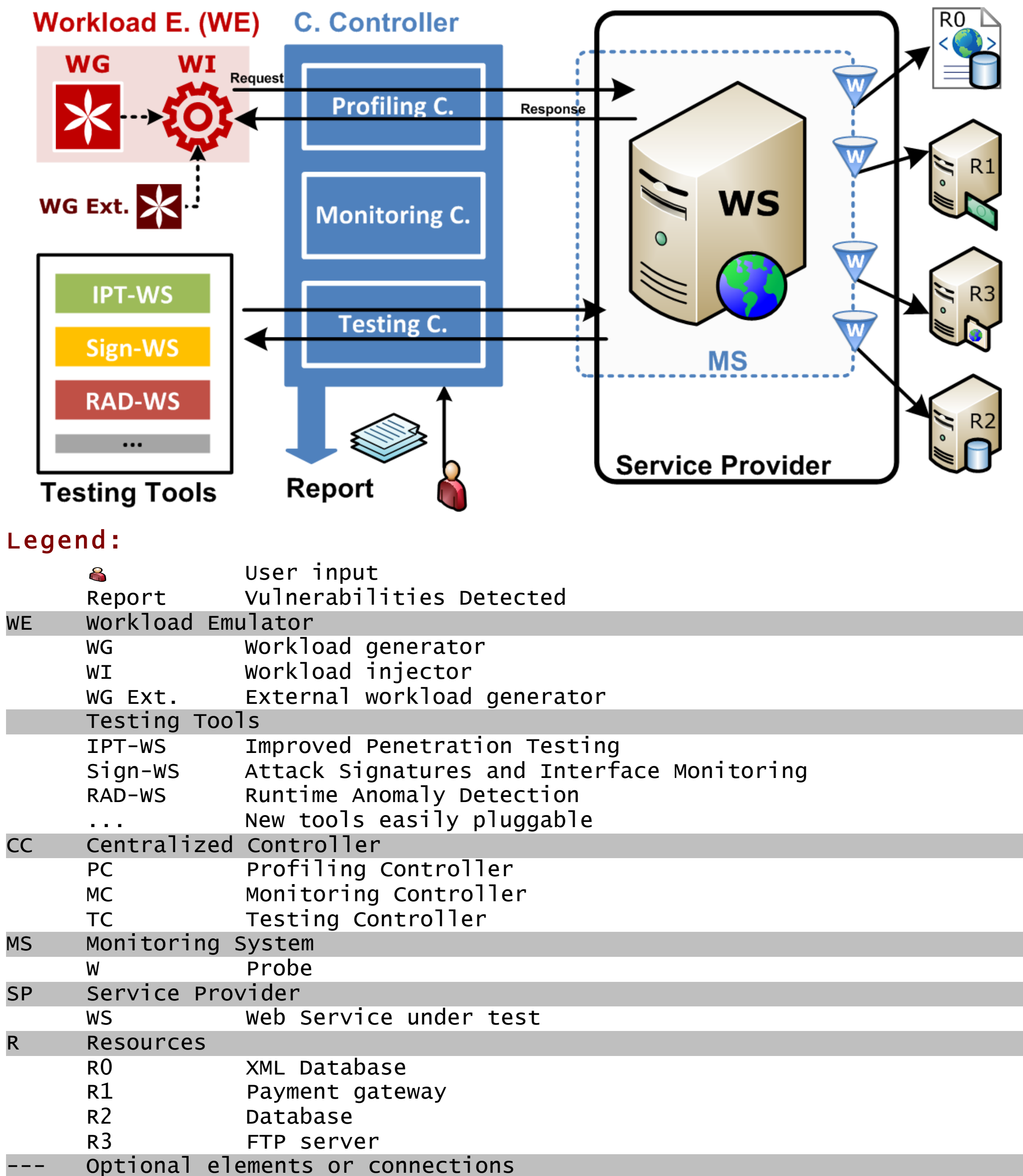


Figure 5.1 – Design of SOA-Scanner.

Interactions between the modules and also with a web service under testing. Some of the presented modules are optional, depending on the web service testing scenario and on user options to improve workload generator.

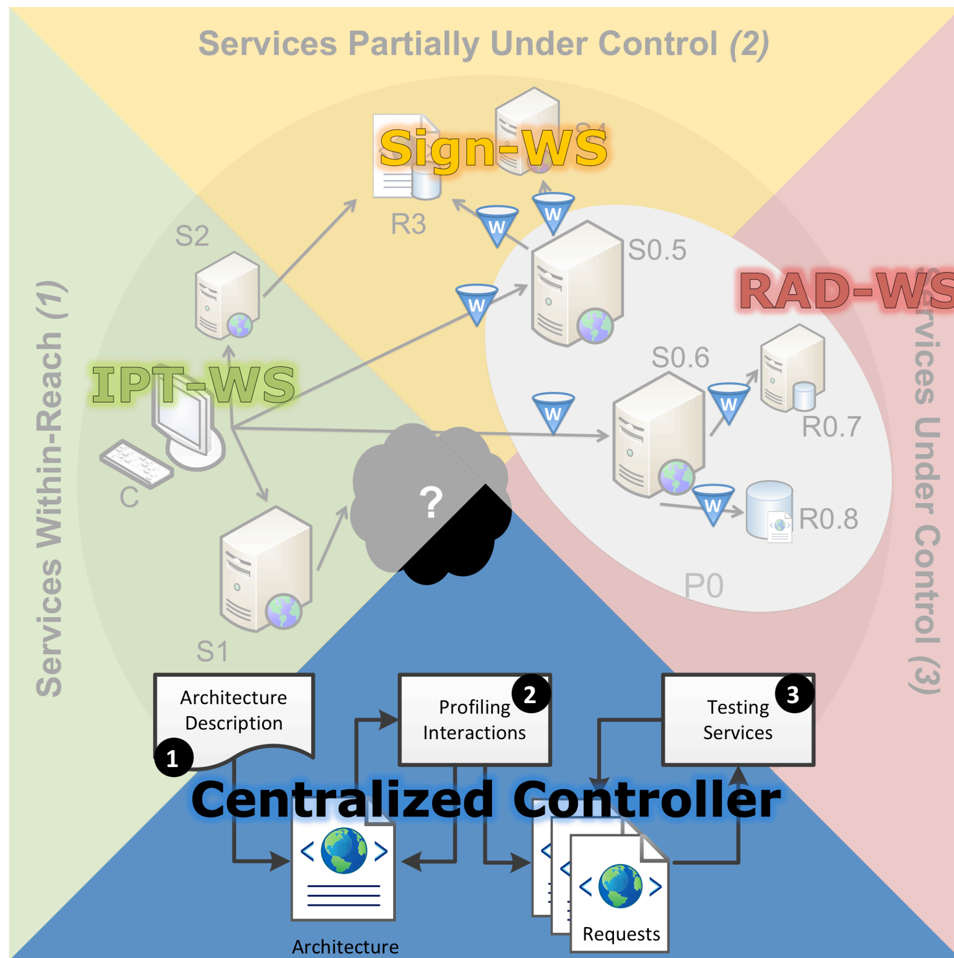


Figure 5.2 – SOA-Scanner components in the infrastructure.
The centralized controller deploys probes in the interfaces of the services under control and uses different testing techniques depending on the testing scenario.

5.2 Centralized Controller

The centralized controller is the core of the tool and consists of three subcomponents that interoperate to coordinate the process:

1. **Profiling controller**: interfaces with the tool's users and is responsible for controlling the generation of the profiling interactions;
2. **Monitoring controller**: centralizes the monitoring elements and maintains a mapping of the service-based infrastructure;
3. **Testing controller**: schedules the testing tasks and merges the results in a report to be presented to the user.

5.2.1 Profiling Controller

The user is provided with a graphical user interface (GUI) that allows defining *services* and *resources* that belong to the target infrastructure. The user must provide information about, at least, the services that act as entry points to the system. However, when possible he must also provide information about other services and resources, as well as the relations between them (Service-to-Service and Service-to-Resource).

For each service or resource, the **user must provide the URL** via which it can be accessed or additional information about how this can be obtained (e.g. a set of connection properties). The controller will, in background, access the URL to discover information about the service or resource and, when applicable, the user is asked for additional information (e.g. parameter input types and domains). For each service the user also needs to define if it is under control (or not). For each resource it may be necessary to select the type of resource, although in most cases the tool can deduct this from the information provided.

For example, in the specific case of a SOAP Web Service, the user must provide the URL of the WSDL file that describes it. This file is processed automatically to obtain the list of operations, parameters, and data types. However, as most cases the valid values for each parameter (i.e. the domain restrictions of the parameter) are not available, the user is asked to provide additional information about the valid domains for each parameter.

Based on the specification provided, the controller creates an initial mapping of the service-based infrastructure (*Architecture* in Figure 5.2). It is a responsibility of the monitoring controller to store and manage this mapping, as explained in Section 5.2.2. Using information about the architecture, the controller creates and issues a set of **profiling interactions** with two goals: to discover additional resources and services and to gather complementary information (e.g. to train vulnerability detection tools based on runtime anomaly detection, as is the case of the RAD-WS tool described in Section 4.3).

In practice, profiling interactions consist of running a set of **workloads,** one for each of the services known. A workload is a set of invocations to the service that simulates a valid and non-malicious usage. The tool is able to automatically generate a random workload for each service using the *random workload generator* presented for the IPT-WS technique in Section 4.1.1 (as discussed there, the main problem of using a random workload generation is that the representativeness of the interactions is not guaranteed). Nevertheless, the modularity of the tool allows easily integrating components that implement other workload generation approaches, including real applications. In practice, the tool allows loading the workload from *xml* files that respect a certain format, but the user can also opt by program an adaptor using the provided API.

5.2.2 Monitoring Controller

Using the specification provided by the user, an initial mapping of the service-based infrastructure is created. The monitoring controller is responsible for managing this mapping of the architecture, which is stored using an *xml* format, following the schema depicted in Figure 5.3.

As we can observe, the format allows to store *services, resources and relations*. Each one consists of an URL and a type. The relations are always between an origin service and a target service or resource. As the resources are not tested (this is out of the scope of the tool), it is neither possible nor important to know if they use other resources. The services are the tricky part of the definitions as it is necessary to store, in addition to the URL and type, information about the operations, fields (parameters) of those operations, and the domains of each field. It is also possible to store the vulnerabilities found in each parameter for later use.

After the creation of the initial architecture, the monitoring controller starts the monitoring phase. In practice, it deploys a set of probes close to the services under control defined initially (see Section 5.3). The monitoring controller then listens for information reported by these probes to update the architecture mapping.

As the workloads are progressively submitted to the services, exercising the infrastructure, additional services and resources are discovered and reported by the probes. For each new service discovered the profiling controller is requested to create the respective workload (repeating the steps discussed above). In the case of the services that are totally or partially under control, the monitoring controller deploys the probes before the workload is submitted. Finally, the monitoring controller receives the information collected by the probes and updates the architecture file accordingly (Section 5.3 provides more details on the functioning of the monitoring probes).

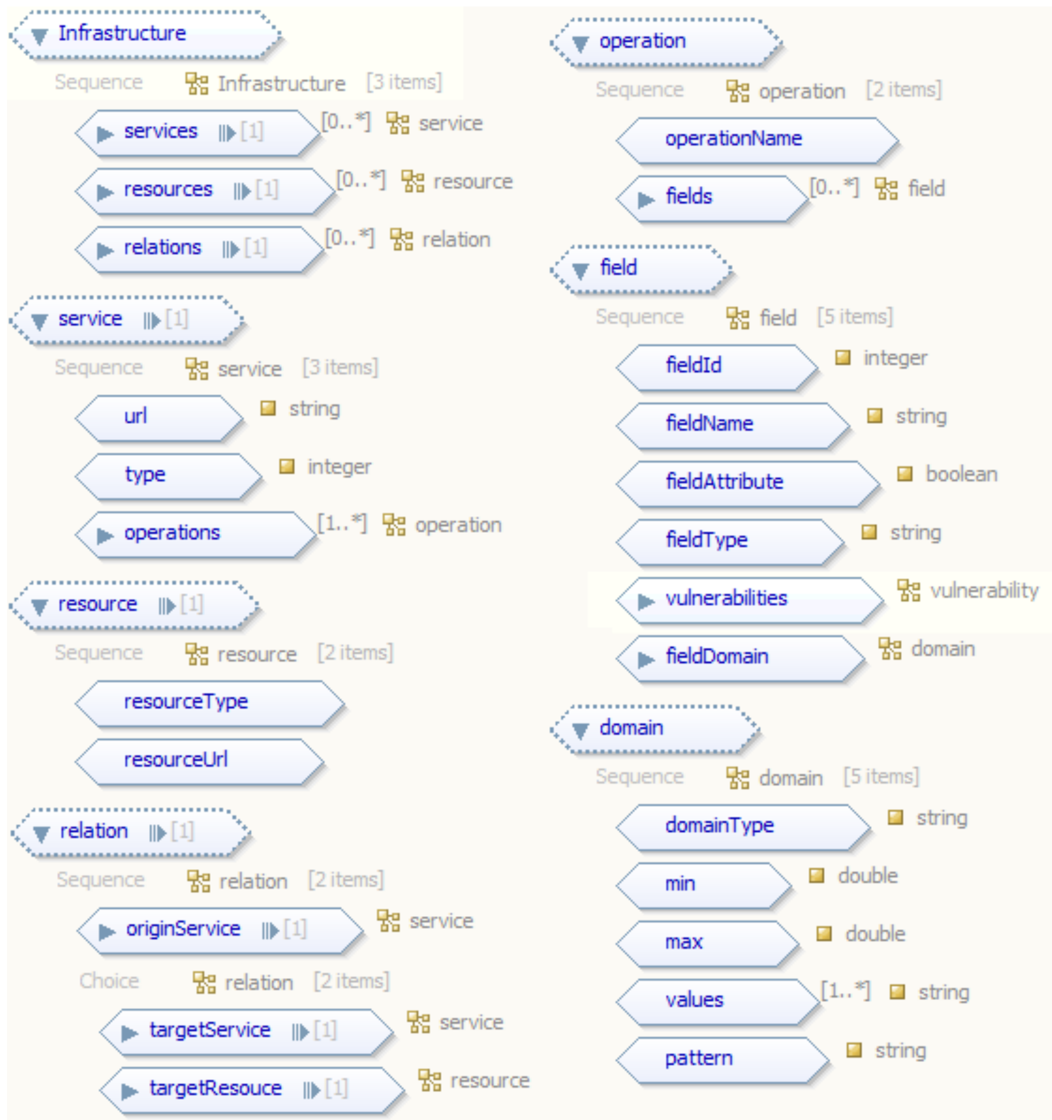


Figure 5.3 – Schema of the XML format used to store the architecture.
This format allows storing all the information about the services and resources that constitute the architecture to be tested, including vulnerability information.

5.2.3 Testing Controller

During the testing phase, the testing controller must schedule the testing tasks. As discussed, the services to be tested may be under the control of multiple providers. To accomplish the different testing scenarios, the testing controller needs several vulnerability detection techniques, which should be implemented following the generic design proposed in Section 3.2.

Different techniques require diverse types of access to the services and diverse types of information to implement the vulnerability detection process (i.e. several testing scenarios exist, as discussed in Section 3.1.3). It is thus a responsibility of the testing controller to assign to each service the technique (or techniques) that will be used to test it, according to the specific testing scenario. A key aspect is that, depending on the level of access and information available about each service, multiple techniques may be applicable. This way, using the knowledge available, the testing controller ranks the applicable detection techniques and, for each service, selects one or more (depending on the configuration) from the highest ranked techniques (Section 5.4 presents the vulnerability detection tools used, the scenarios applicable to each one, and discusses how the SOA-Scanner ranks these tools).

After completing the testing tasks, the testing controller builds an integrated report to present to the user. In practice, the vulnerabilities detected are associated with its location (web service, operation, parameter) and added to the architecture mapping. When more than one vulnerability detection technique is scheduled to test the same web service, it is necessary to deal with contradictory results because, as explained previously, different tools frequently report distinct vulnerabilities for the same piece of code (the current implementation associates to each vulnerability the tools that reported it and weighs this information using the ranking defined in Section 5.4).

5.3 Monitoring System

The monitoring system consists of a **network of probes** (distributed as shown in Figure 5.2) that have the responsibility of collecting information about the interface activity originated in services totally or partially under control. To achieve this, each time a new service is found and classified as under control, the controller module deploys probes in this service, following the steps presented later in this section.

As discussed in Section 3.3, there are multiple alternatives available for implementing these probes, including: network packet sniffing, use of proxies and driver instrumentation (details on these alternatives are provided in Section 3.2.3). In its current implementation, the SOA-Scanner uses probes implemented making use of driver instrumentation with aspect-oriented programing (AOP), which allows injecting crosscutting concerns into any application in a non-intrusive way (Kiczales et al. 2002). In this case, the aspects used are as simple as possible to avoid introducing bugs. Figure 5.4 presents an example of an aspect.

As it is possible to observe, minimum changes are introduced in the drivers. In practice, a light thread is added to each driver. This thread runs in background and is responsible for handling the task of reporting the new interactions found to the centralized controller. The *Around advices* (line 3) are introduced to intercept the necessary methods. In this case, the interception consists of extracting the necessary

values from the intercepted method arguments (line 5), check if they are new (line 6) and if so deliver it to the background thread (line 7), and finally proceed with the execution of the method (line 8). This way, the impact of the probe in the behavior of the application is reduced to a minimum. Nevertheless, the introduced method is the result of well tested code effort and the resulting drivers were also thoroughly tested to detect potential bugs.

```
1: @Aspect
2: public class AroundConnectionProbe extends AbstractProbe {

3:    @Around("execution(* java.sql.Driver.connect(..))")
4:    public Object aroundConnect(ProceedingJoinPoint p) throws {
5:       final String dbUrl = (String) p.getArgs()[0];
6:       if (databaseUrls.isNew(dbUrl)) {
7:          Thread.enqueue(dbUrl);
         }
8:       return p.proceed();
      }
   }
```

Figure 5.4 – Example of an Aspect in Java.

The code intercepts jdbc connections and in the case of the new ones, enqueues to a background thread that will report to the controller, and proceeds immediately.

The tool includes aspects that can be used to instrument three types of drivers, which is sufficient to test a complex infrastructure consisting of SOAP web services using relational database solutions or XML based solutions. Those aspects are:

- **JAX-WS:** the Java API for XML-Based Web Services is used for creating web services. It can be also be used to invoke other web services (Kotamraju 2007). In this case, the probe monitors interactions among services.
- **JDBC:** the Java Database Connectivity API is designed to access any kind of tabular data, but it is mostly used to access relational databases in Java applications (Reese and Oram 2000). In this case, the probe monitors the interactions with database resources.
- **JDOM:** a complete Java-based solution for accessing, manipulating, and outputting XML data from Java code (Hunter 2002). It supports XPath, a query language for selecting nodes from an XML document. In this case, the probes monitor the access to XML resources.

The output of this process consists of new driver libraries that can be used by referring the modified versions in the *classpath* of the deployed application (instead of the original ones). In practice, the SOA-Scanner copies the required set of libraries to the application servers where the web services are running and replaces the

original ones by updating the *classpath* parameter. At runtime, this enhanced driver library monitors the target interface. During the execution of the workloads (profiling phase), the probes monitor the interactions and report the newly found ones to the centralized controller via the network, which updates the architecture accordingly. This process continues in an iterative way, expanding the mapping of the architecture of the infrastructure, until no new services are detected, i.e. until finishing the execution of the workloads for all the services known.

5.4 Testing Tools

After completing the profiling phase, the services are tested. It is possible, although improbable, that new services are found in this phase. These services must also be reported to the controller in order for the process to be applied to them also (starting from the profiling phase).

As mentioned before, the SOA-Scanner supports the integration of testing tools that implement the generic approach and components presented in Section 3.2. The current implementation includes three tools implementing the techniques presented in Chapter 4, which cover the three testing scenarios defined in the reference infrastructure in Section 3.1.3: 1) within-reach, 2) partially under control, and 3) fully under control. Just to recall, the included tools are:

1. **Penetration Testing (IPT-WS)**: black-box technique that tries to penetrate the service by issuing a huge amount of interactions;
2. **Attack Signatures and Interface Monitoring (Sign-WS)**: penetration testing improved with extra information, yet without needing to access or modify the service code;
3. **Runtime Anomaly Detection (RAD-WS)**: profiles the behavior of the service to detect vulnerabilities by finding deviations from the normal (earned) behavior during an attack phase.

To better understand the context, Table 5.1 crosses the testing techniques against the scenarios where it is possible to apply them (report to Chapter 4 for details on these techniques and the discussion on their weaknesses and strengths).

The techniques in the table are ordered from the least effective (IPT-WS) to the most effective (RAD-WS). This ranking was established using the results from the benchmarking campaigns presented in Chapter 7. These results are far from unexpected, as the most effective tools use more information than the less effective ones, constraining the scenarios in which they can be used (e.g. the most effective tool (RAD-WS) can only be used in services under control, but the less effective can be used in all scenarios).

Table 5.1 – Correlation between testing techniques and scenarios.
The check symbol marks the scenarios in which each tool is available. The tools are ordered from the least effective (left) to the most effective (right).

Scenario \ Tool	IPT-WS (see Section 4.1)	Sign-WS (see Section 4.2)	RAD-WS (see Section 4.3)
Within-Reach	✓	✗	✗
Partially Under Control	✓	✓	✗
Under Control	✓	✓	✓

New tools that extend the design presented in Section 3.2 can be easily added to SOA-Scanner. For this, it is necessary to add the libraries of the tool to the SOA-Scanner and register the tool in the configuration file. It is also necessary to configure in which scenarios the tool can be used (in practice, fill another column in the Table 5.1) and to provide information about the ranking of the tool in respect with the existing ones.

The testing task finishes with the tool reporting to the testing controller the vulnerabilities identified. This report includes, for each tested service, information about the vulnerabilities detected: web service, operation, parameter, and type of vulnerability. The testing controller maintains a global report of the vulnerabilities, executing a process based in two key steps: 1) the reported vulnerabilities are added to the architecture description and indexed by their location in the infrastructure: web service, operation, parameter; 2) the tool extracts all the vulnerabilities reported and unifies them in a single report, keeping the information of the location of the vulnerability, its type and which tools reported it vulnerability. Figure 5.5 depicts this process.

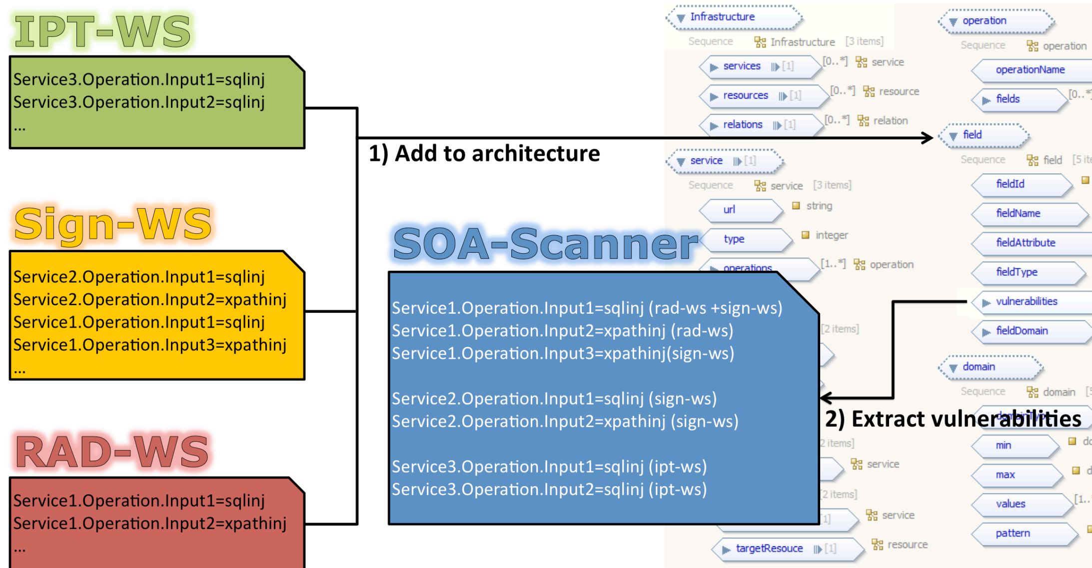


Figure 5.5 – Process to Integrate the Reports of multiple tools.

The vulnerabilities are added to the mapping of the architecture indexed by operation parameter (field). In the second step, all the vulnerabilities are extracted to create an integrated report.

5.5 Conclusion

This chapter presented the SOA-Scanner, an extensible tool implementing the integrated approach for vulnerability detection in service-based infrastructures presented Section 3.3. Although the current implementation targets only SOAP web services, and focuses on injection vulnerabilities, the tool follows a modular architecture and can be easily extended to more types of software services and security vulnerabilities.

The tool is based in three main components that allow implementing the generic steps of the approach. The first is a centralized controller that, besides controlling the execution of the complete process, is in charge of receiving input information from the user and reporting the testing results to him. Additionally, it is responsible for scheduling the profiling interactions and testing tasks. The second is the monitoring system that, using a set of probes deployed to the services, collects information about the interactions between the parts of the infrastructure, also discovering new services and resources to be tested. The final component is in charge of integrating a set of vulnerability detection tools following a generic and modular design. Three testing tools are already included, covering different testing scenarios and taking

advantage of the different levels of access to the services. Adding new tools to the component is easy and improves the effectiveness of the testing process.

A case study demonstrating the capabilities of the tool is presented in Chapter 7. The case study consists of a service-based infrastructure that represents a simple organization. The infrastructure is composed by different services with different levels of access and using different types of resources and, consequently, containing different types of vulnerabilities.

Chapter 6
Benchmarking Vulnerability Detection Tools for Services

Vulnerability detection tools are frequently considered the silver-bullet for finding vulnerabilities in web services. As mentioned before, developers and system integrators widely use such tools to perform automated security checking in web applications and services, which makes them some of the best examples of critical artifacts for secure software development.

Due to time constraints or resource limitations, developers frequently have to select a specific tool from the large set of tools available (usually without really knowing how good each tool is) and strongly rely on that tool to detect potential security problems in the code being developed. Furthermore, it is clear that the performance of a given tool strongly depends on the specificities of the application scenario (i.e. the class of target web services (e.g. SOAP, REST), the types of vulnerabilities to detect, etc.), and that the same tool may have different performance levels in different scenarios.

In this context, developers and researchers urge the definition of a practical approach that helps them assessing and comparing alternative tools concerning their ability to detect vulnerabilities. Benchmarking is a standard way to evaluate and compare different systems or components according to specific characteristics (e.g. performance, dependability, etc.) (Gray 1992). Several works have tried to assess the effectiveness of vulnerability detection tools (e.g., (Antunes and Vieira 2009a; Vieira, Antunes, and Madeira 2009; Fonseca, Vieira, and Madeira 2007; Wagner et al. 2005)) however, none has proposed a standard approach that allows the comparison of results, as is the case of benchmarking.

To address this problem, we propose an **approach for benchmarking vulnerability detection tools for web services**. This approach specifies all the components and

steps needed to define benchmarks to assess and compare alternative tools, with particular focus on two well known metrics: precision and recall. Additionally, it defines the other required components, which include a workload (work that the vulnerability detectors under testing have to do, in the form of a set of web services that should be searched for vulnerabilities) and a well-defined benchmarking procedure (set of steps that have to be followed for conducting a benchmarking campaign, ranging from the preparation of the experiments to the ranking of the tools and selection of the most adequate one). A key aspect if that the proposed approach is generic and can be used to specify different benchmarks for specific application domains and different types of vulnerabilities.

The benchmarking approach has been used to define two concrete benchmarks. The first targets tools capable of detecting SQL Injection vulnerabilities in SOAP web services, including detection approaches based on penetration testing, static code analysis, and runtime anomaly detection. This benchmark is **based on a well defined and large set of web services adapted from standard performance benchmarks**, and includes both vulnerable and non-vulnerable versions of the services. The main limitation of this benchmark is that, although based on a well-defined set of rules, it is not protected against "*gaming*" (i.e. adaptations/tuning that allow producing optimistic or biased results). In fact, as the workload is known, providers can easily tune their tools to maximum effectiveness in the context of the benchmark, while failing in different scenarios.

To overcome this limitation, we propose a second benchmark for penetration testing tools capable of detecting SQL Injection vulnerabilities in SOAP web services. This benchmark circumvents the “*gaming*” problem by **allowing the benchmark user to specify the workload** (i.e. the workload is not predefined and is unknown to the tools’ providers) that best represents his specific development conditions, thus providing more realistic (and specific to the development environment) results. To support the user in the task of characterizing the workload, the benchmark includes a procedure and a tool to identify vulnerabilities in the target web services, thus avoiding the need for conducting such analysis manually.

The outline of this chapter is as follows. Next section presents the generic benchmarking approach and its components. Section 6.2 presents the benchmark based on the predefined workload, while Section 6.3 discusses the benchmark based on the user-defined workload. Section 6.4 discusses aspects related to the validation of key benchmarking properties. Finally, Section 6.5 concludes the chapter.

6.1 Generic Benchmarking Approach

Our proposal to benchmark vulnerability detection tools is inspired on measurement-based techniques. The basic idea is to exercise the tools under benchmarking using web services code with and without vulnerabilities and, based

on the detected vulnerabilities, calculate a small set of measures that portray the detection capabilities of the tools.

Due to the high diversity of web services, types of vulnerabilities, and vulnerability detection approaches, the definition of a benchmark for all vulnerability detection tools is an unattainable goal. This way, as recommended in (Gray 1993), a benchmark must be specifically targeted to a particular domain. In fact, the division of the spectrum into well-defined areas is necessary to make it possible to make choices during the definition of the benchmark components. In the context of this work, the **definition of the benchmarking domain** includes selecting the class of web services, the type of vulnerabilities, and the vulnerability detection approaches for the target tools under benchmarking, which mainly influence the definition of the workload (see Section 6.1.2).

The main components of a benchmark are:

- **Metrics**: characterize the effectiveness of the tools under benchmarking in detecting the vulnerabilities that exist in the workload services. The metrics must be easy to understand and must allow the comparison among different tools.
- **Workload**: represents the work that a tool must perform during the benchmark execution. In practice, it consists of a set of services (with and without security vulnerabilities) that will be used to exercise the vulnerability detection tools during the benchmarking process. Depending on the goal of the benchmark, the workload can be predefined (i.e. defined in the benchmark specification itself) or provided by the benchmark user.
- **Procedure**: describes the procedure and rules that must be followed when executing the benchmark.

The procedure and rules have to be specified during the definition of the benchmark. In fact, those procedures and rules are the core of the benchmark specification. Although this is, obviously, dependent on the specific benchmark, in the following points we identify some guidelines on specific aspects needed in most of the cases:

- *Standardized procedures* for “translating” the workload defined in the benchmark specification into the actual workload that will be applied to the tools under benchmarking. These procedures guarantee that the different users understand and use the benchmark in a consistent way.
- *Uniform conditions* to build the experimental benchmark setup, perform initialization tasks that might be defined in the specification, and run the benchmark according to the specification (i.e. apply the workload and calculate the metrics).

- *Rules* related to the collection of the experimental results. These rules may include, for example, available possibilities for system instrumentation, degree of interference allowed, common references and precision for timing measures, etc.
- *Rules* for the production of the final measures from the direct experimental results, such as calculation formulas, ways to deal with uncertainties, errors and confidence intervals, etc.
- *Disclosures* required for interpreting the benchmark measures. In a similar way to what happens in other domains (Kanoun and Spainhower 2008), a report may be required in order for results to be considered compliant with the benchmark specification. The goal is to allow the reproduction of the experiments in other sites using the same vulnerability detection tools.
- *Rules* to avoid "*gaming*" to produce optimistic or biased results. For example, rules regarding the use and/or definition of the workload.

6.1.1 Metrics

The benchmark metrics should be computed from the information collected during the benchmark run and must follow the well-established measuring philosophy typically used in performance and dependability benchmarking (Gray 1993; Kanoun and Spainhower 2008). In fact, benchmarks should provide relative measures (i.e. measures related to the conditions disclosed in the benchmark report) that can be used for comparison or for improvement and tuning. For example, it is well known that performance benchmarking results do not represent an absolute measure of performance and cannot be used for planning the capacity or to predict the actual performance of the system in field. In a similar way, the measures in a benchmark for vulnerability detectors must be understood as results that can only be used to characterize the tools in a relative fashion (e.g. to compare alternative tools).

A key difficulty related to the definition of the benchmark metrics is that different vulnerability detection tools report vulnerabilities in different ways. For example, for penetration testing tools (that identify vulnerabilities based on the application response) vulnerabilities are reported for each vulnerable input. On the other hand, for static analysis tools (that vet code looking for possible security issues) vulnerabilities are reported for each vulnerable line in the code. Due to this dichotomy, it is very difficult (or even impossible) to compare the effectiveness of tools that implement different vulnerability detection approaches, based on the number of vulnerabilities reported for the same piece of code. This way, our proposal is to characterize vulnerability detection tools using the F-Measure proposed by van Rijsbergen (Van Rijsbergen 1979), which is largely independent of the way vulnerabilities are counted. In fact, it represents the harmonic mean of two

very popular measures (precision and recall), which, in the context of vulnerability detection, can be defined as:

- **Precision**: the ratio of correctly detected vulnerabilities to the number of all detected vulnerabilities. In our context it can be represented as:

$$precision = \frac{TP}{TP + FP} \qquad (1)$$

- **Recall**: the ratio of correctly detected vulnerabilities to the number of known vulnerabilities. In our context it can be represented as:

$$recall = \frac{TP}{TV} \qquad (2)$$

Where:

- TP (true positives) is the number of true vulnerabilities detected (i.e. vulnerabilities that, in fact, exist in the code);
- FP (false positives) is the number of vulnerabilities detected that, in fact, do not exist;
- TV (true vulnerabilities) is the total number of vulnerabilities that exist in the code.

Assuming an equal weight for precision and recall, the formula for the **F-Measure** is:

$$F - Measure = \frac{2 \cdot precision \cdot recall}{precision + recall} \qquad (3)$$

A highly effective tool will generate a high F-Measure (which obviously ranges between 0 and 1). For example, consider a set of 100 vulnerabilities in a piece of code. A tool that achieves a precision of 0.7 is able to detected vulnerabilities with a probability of 70%. A recall of 0.8 expresses that 80% of all the known vulnerabilities are detected and that 20% are missed. In this case the F-Measure would be approximately 0.7466.

The three metrics can be used to establish a ranking of several tools depending on the purposes of the benchmark user (i.e. to select the tool with the highest precision, the highest recall, or having the best compromise between precision and recall). Note that these are proven metrics that are typically used to portray the effectiveness of

many computer systems (Van Rijsbergen 1979), particularly in information retrieval. Thus, they are easy to be understood by most users.

6.1.2 Workload

The workload defines the work that has to be done by the vulnerability detection tools during the benchmark execution. In other words, the workload should include the code that will be used to exercise the vulnerability detection capabilities of the tools under benchmarking. It is mainly influenced by three factors:

- The **class of web services** (e.g. SOAP, REST), which allows defining the characteristics of the services that will be used to exercise the tools under benchmarking;
- The **types of vulnerabilities** (e.g. SQL Injection, XPath Injection, file execution) to be detected by the tools. This defines vulnerabilities that must exist in the workload;
- The **vulnerability detection approaches** (e.g. penetration testing, static analysis, anomaly detection), which specify the approaches used by the tools under benchmarking to detect vulnerabilities.

Three different types of workloads can be considered for benchmarking purposes:

- **Real workloads**: these are made of applications used in real environments that have real vulnerabilities. Benchmarks using real workloads are expected to be quite representative. However, many applications and vulnerabilities may be needed to achieve good representativeness and those applications frequently require some adaptation.
- **Realistic workloads**: artificial workloads that are based on the adaptation of real applications in the domain of the benchmark. Although artificial, realistic workloads still reflect real situations and are more portable than real workloads.
- **Synthetic workloads**: a synthetic workload can be a set of randomly selected code elements in which vulnerabilities are artificially injected. Synthetic workloads are easy to use but their representativeness is questionable.

In summary, the workload includes the web services code that will be used to exercise the vulnerability detection capabilities of the tools under benchmarking. For example, the workload for a benchmark targeting static code analysis tools capable of detecting SQL Injection vulnerabilities in web services must include the source code of web services with realistic (and well identified) SQL Injection vulnerabilities. On the other hand, for penetration testers access to the source code is not needed.

Two options are available regarding the definition of the workload (these apply to the three types of workloads defined above):

- **Predefined workload**: the benchmark includes a predefined set of web services with vulnerabilities.
- **User-defined workload**: the benchmark leaves to the user the responsibility of selecting the target set of services.

While the first approach guarantees some level of standardization and uniformity of results across different executions of the benchmark, the second allows circumventing the "*gaming*" problem and best represents the user specific development conditions, thus providing more realistic results. A key aspect is that the workload should include both vulnerable and non-vulnerable services in order to better characterize the tools under assessment (e.g. vulnerable services are useful to gather coverage metrics while non-vulnerable services help on assessing false positive rates).

In both cases, information about the vulnerabilities that exist in the target web services is needed in order to be able to calculate the metrics. This can be obtained by extensively searching the web services for vulnerabilities, using different techniques, including penetration testing, code inspection, static analysis, etc. For the case of predefined workloads this information must be provided together with the benchmark specification. On the other hand, for user-defined workloads, the benchmark may provide only guidelines on how to perform the workload characterization or include tools to facilitate the work.

As different vulnerability detection approaches report vulnerabilities in different ways, different characterizations about the existing vulnerabilities may be required (e.g. the number of vulnerable inputs is needed for penetration testing tools, while the number of vulnerable lines of code is required for static analysis tools), depending on the vulnerability detection approaches of tools targeted by the benchmark (as defined in the benchmark domain).

6.1.3 Procedure

Although the detailed procedure depends on the specificities of the benchmark, we proposed three main phases:

1. **Preparation**: prepare the benchmark execution, including:

    a. **Workload selection and characterization**: this is a step that is required only when the benchmark leaves to the user the responsibility of selecting the target services. In such case, the user

has to define the web services and characterize the existing vulnerabilities (as discussed before).

b. **Tools identification:** select the vulnerability detection tools to be benchmarked.

2. **Execution**: use the tools under benchmarking to detect vulnerabilities in the workload services.
3. **Comparison**: characterize the tools benchmarked. This includes two steps:

   a. **Metrics calculation**: analyze the vulnerabilities reported by the tools (i.e. confirm true positives and identify false positives) and calculate the metrics.

   b. **Ranking and selection**: rank the tools under benchmarking using F-Measure, precision, and recall. Based on the preferred ranking, select the most effective tool (or tools).

In the case of benchmarks based on a predefined workload Step 1.a is not required, as the target web services are characterized in the benchmark specification (including the number of existing vulnerabilities). On the other hand, for benchmarks based on a user-defined workload Step 1.a is extremely relevant, as it greatly influences the benchmark results (e.g. if the workload services do not contain representative vulnerabilities then the measures will not be representative of the tools effectiveness).

The benchmark execution is a straightforward process and consists of using each tool to detect vulnerabilities in the workload code. Depending on the tool under benchmarking this may require some configuration of the parameters. After executing the benchmark it is necessary to compare the vulnerabilities detected by the tool with the ones that effectively exist in the workload code. Vulnerabilities correctly detected are counted as true positives and vulnerabilities detected but that do not exist in the code are counted as false positives. This is the information needed to calculate the precision and recall of the tool, and consequently the F-measure.

6.2 Benchmark with a Predefined Workload [VDBenchWS-pd]

In this section we present a benchmark (VDBenchWS-pd) based on a workload consisting of a well defined and large set of web services adapted from standard performance benchmarks, including both vulnerable and non-vulnerable versions of the services. The benchmark targets the following domain:

- **Class of web services**: SOAP web services implemented in Java;
- **Type of vulnerabilities**: SQL Injection;
- **Vulnerability detection approaches**: penetration testing, static code analysis, and runtime anomaly detection.

The reasoning behind the selection of this domain is as follows. Web services implemented in Java are nowadays widely used in many scenarios, with several frameworks available to implement and support the development (Curbera et al. 2002; D. A Chappell and Jewell 2002a). SQL Injection vulnerabilities are particularly relevant in web services (Christey and Martin 2007), as these frequently use a data persistence solution based in a relational database. Finally, we have all the information needed regarding the vulnerabilities in predefined set of web services, which are quite representative of real scenarios. Although (virtually) any vulnerability detection approach can be used to detect vulnerabilities in these services, the benchmark targets specifically penetration testing, static code analysis, and runtime anomaly detection (Arkin, Stender, and McGraw 2005; Ayewah et al. 2008; Kruegel and Vigna 2003), which are widely used techniques (and that include the ones used by the tools proposed in Chapter 4).

As mentioned before, the workload is the component most influenced by the benchmarking domain and strongly determines the benchmark results. In order to define a representative workload we have decided to adapt code from three standard benchmarks developed by the Transactions processing Performance Council, namely: TPC-App, TPC-C, and TPC-W (see details on these benchmarks at (Transaction Processing Performance Council 2009)). TPC-App is a performance benchmark for web services infrastructures and specifies a set of web services accepted as representative of real environments. TPC-C is a performance benchmark for transactional systems and specifies a set of transactions that include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. Finally, TPC-W is a benchmark for web-based transactional systems. The business represented by TPC-W is a retail store over the Internet where several clients access the website to browse, search, and process orders[7].

As an adaptation of real applications, the proposed workload follows the **realistic workloads** approach, and thus needs to include realistic SQL Injection vulnerabilities. Although feasible, artificial vulnerabilities injection (Fonseca, Vieira, and Madeira 2007) would introduce complexity and suffer from representativeness issues. When possible, the option should be to consider code with real

[7] Although TPC-C and TPC-W do not define the transactions in the form of services, they can easily be implemented and deployed as such.

vulnerabilities, (inadvertently) introduced by the developers during the coding process. This way, for the present work we invited an external developer to implement the TPC-App web services (without disclosing the objective of the implementation in order not to influence the final result) and successfully searched the web for publicly available implementations of TPC-C and TPC-W, which were adapted to the form of web services by the same external developer (this adaptation consisted basically on the encapsulation of the transactions as web services, without modifying the functional structure of the code). Obviously, this was a risky choice as there was some probability of getting code without vulnerabilities. However, as expected, the final code includes several SQL Injection vulnerabilities (see Table 6.1), which is representative of the current situation in real web services development (as shown in (Vieira, Antunes, and Madeira 2009; NTA Monitor 2011b)).

The workload services are currently implemented in Java. Although this is not relevant when benchmarking penetration testing tools (that detect vulnerabilities based on the application responses and do not need to have access to the source code), it limits the application of the benchmark to static code analyzers and runtime anomaly detectors that look for vulnerabilities in Java code. This way, to increase the domain of the benchmark, we would need to develop the workload in more languages. The problem is that different implementations of the services might have different vulnerabilities, which limits tools comparison. For example, the comparison of a static code analyzer for Java with a static code analyzer for C# is not possible if the Java code has vulnerabilities that are different from the ones in the C# code. One possibility to mitigate this problem is to use automatic code transformation to translate the current implementation of the services to other languages, while maintaining the existing vulnerabilities. Obviously, this process requires a subsequent manual verification step to check the correctness and usefulness of the transformed code. Nevertheless, the current implementation is sufficient to demonstrate the benchmarking approach proposed in this chapter, as Java is a language widely used to implement web services and there are many vulnerability detection tools that focus on Java code.

To characterize the vulnerabilities that exist in the workload code, we invited a team of 3 external developers, with two or more years of experience in security of database centric applications, to conduct a formal inspection of the code looking for vulnerabilities. As the different vulnerability detection approaches considered report vulnerabilities in different ways (penetration testers report the vulnerable inputs, while static code analyzers and runtime anomaly detectors report the vulnerable lines of code), we asked the security experts to identify both the input parameters and the source code lines prone to SQL Injection attacks. Table 6.1 presents the summary of the vulnerabilities detected by the security experts, the total number of lines of code (LoC) per service, and the average Cyclomatic Complexity (Lyu 1996) (Avg. C.) of the code (calculated using SourceMonitor (Campwood Software 2008). The results show a total of 56 vulnerable inputs and of 49 vulnerable SQL queries in

the set of services considered. As shown, the services have diverse sizes and complexities, which is representative of real scenarios.

In order to exercise the tools under benchmarking in a more exhaustive and realistic manner we decided to generate additional versions of the web services. The first step consisted of creating a new version for each service with all the known vulnerabilities fixed. Then we generated several versions for each service, each one having only one vulnerable SQL query. This way, for each web service we have one version without known vulnerabilities, one version with N vulnerabilities, and N versions with one vulnerable SQL query each. This accounts for a total of 80 versions, with **158 vulnerable inputs and 87 vulnerable queries** as listed in Table 6.2, which we believe is enough to exercise detection tools (as shown in Chapter 7). In summary, we have the following versions for each web service:

Table 6.1 – Vulnerabilities found in the workload services.

For each service it is presented the vulnerabilities reported, the number of lines of code and the average complexity of the code. The differences of the presented values show the diversity of the services.

Source	Service Name	Vuln. Inputs	Vuln. Queries	LoC	Avg. C.
TPC-App	ProductDetail	0	0	121	5
	NewProducts	1	1	103	4.5
	NewCustomer	15	4	205	5.6
	ChangePaymentMethod	2	1	99	5
TPC-C	Delivery	2	7	227	21
	NewOrder	3	5	331	33
	OrderStatus	4	5	209	13
	Payment	6	11	327	25
	StockLevel	2	2	80	4
TPC-W	AdminUpdate	2	1	81	5
	CreateNewCustomer	11	4	163	3
	CreateShoppingCart	0	0	207	2.67
	DoAuthorSearch	1	1	44	3
	DoSubjectSearch	1	1	45	3
	DoTitleSearch	1	1	45	3
	GetBestSellers	1	1	62	3
	GetCustomer	1	1	46	4
	GetMostRecentOrder	1	1	129	6
	GetNewProducts	1	1	50	3
	GetPassword	1	1	40	2
	GetUsername	0	0	40	2
Total		**56**	**49**	**2654**	**-**

- **Version with all the vulnerabilities**: includes all the vulnerabilities introduced by the developers, which makes it representative of real scenarios. In fact, it simulates the situations in which a tool is used over a web service that includes multiple (and maybe interdependent) vulnerabilities.
- **Version without known vulnerabilities**: useful to characterize the tools in terms of false positives. In fact, tools should not detect any vulnerabilities in this code as it does not have known vulnerabilities (guaranteed by the experts that reviewed the code).
- **Versions with one vulnerability**: represent more subtle scenarios in which there are few vulnerabilities in the code. This makes vulnerability detection more complex.

Table 6.2 – Final numbers of vulnerabilities in the workload servicers.
For each service it is presented the number of versions created and the total number of vulnerabilities exiting.

Source	Service Name	Versions	Vuln. Inputs	Vuln. Queries
TPC-App	ProductDetail	2	0	0
	NewProducts	2	1	1
	NewCustomer	6	35	8
	ChangePaymentMethod	2	2	1
TPC-C	Delivery	9	10	14
	NewOrder	7	15	10
	OrderStatus	7	18	10
	Payment	13	34	22
	StockLevel	4	6	4
TPC-W	AdminUpdate	2	2	1
	CreateNewCustomer	6	27	8
	CreateShoppingCart	2	0	0
	DoAuthorSearch	2	1	1
	DoSubjectSearch	2	1	1
	DoTitleSearch	2	1	1
	GetBestSellers	2	1	1
	GetCustomer	2	1	1
	GetMostRecentOrder	2	1	1
	GetNewProducts	2	1	1
	GetPassword	2	1	1
	GetUsername	2	0	0
Total		**80**	**158**	**87**

It is important to emphasize that we are aware of the limitations of the workload code. In fact, this code may not be representative of all the SQL Injection vulnerability patterns found in real web services. However, what is important is to define the benchmark components in such a way that allow characterizing the effectiveness of the tools under benchmarking in a relative manner (i.e. that allow establishing comparisons between tools). Based on the extensive experimental evaluation conducted (see Chapter 7), and in particular on the benchmark properties discussion, we believe that the proposed workload is sufficient to assess and compare the effectiveness of SQL Injection vulnerability detection tools for web services. Nevertheless, the proposed benchmark can easily be extended to include more services. Readers can find details on the benchmark (with detailed results) at (Antunes 2013).

6.3 Benchmark with a User-Defined Workload [PTBenchWS-ud]

In this section we present a benchmark (PTBenchWS-ud) based on a user-provided workload (any set of services), allowing to the user to overcome the "*gaming*" problem and providing, at the same time, more realistic results. The benchmark targets the following domain:

- **Class of web services**: SOAP web services (Curbera et al. 2002).
- **Type of vulnerabilities**: SQL Injection (Christey and Martin 2007).
- **Vulnerability detection approaches**: penetration testing (Arkin, Stender, and McGraw 2005).

Contrarily to the benchmark presented in Section 6.2, this benchmark is not limited to Java, as it is independent from the technology used to implement the web services. The main reason for this is that penetration testing, the target vulnerability detection approaches, does not require access to source code testing the web services from an external point of view. Together with the focus of approach for characterizing the workload (which is based on testing, as discussed later on this section), this is also a reason for focusing this benchmark on penetration testing.

The set of web services that compose the benchmark workload is to be defined by the benchmark user. This should include a number of SOAP web services with and without SQL Injection vulnerabilities. As defined in Section 6.1.2, this workload can be real, realistic, or synthetic. What is important is to understand that the workload definition determines the benchmark results and properties, thus the user should be aware of the impact of the decisions regarding the web services being considered.

A key aspect is the characterization of the existing vulnerabilities. As the target of the benchmark is penetration testing tools, the number of vulnerable inputs is needed to later calculate the metrics. Such characterization can be based on an extensive **manual analysis** of the selected web services in order to identify the existing vulnerabilities (in a similar way to what we did for the benchmark proposed in Section 6.2). The problem is that such process can become extremely expensive if the set of services is large and complex. Thus, as an alternative, we propose an **automatic approach for identifying the base set of vulnerabilities**, grounded on the use of a tool that combines attack signatures and interface monitoring to detect SQL Injection vulnerabilities in web services presented in Section 4.2.

As mentioned before, the Sign-WS technique addresses the limitations of penetration testing by using attack signatures and interface monitoring for the detection of injection vulnerabilities in web services. The goal is to improve the detection process by providing enhanced visibility, yet without needing to access or modify the code of the target service. The key assumption is that most injection attacks manifest, in some way, in the interfaces between the attacked web service and other resources (e.g. database, operating system) and services. Although the proposed approach does not guarantee the detection of all existing vulnerabilities, it assures that no false positives are reported. The vulnerabilities detected will serve as reference to estimate the number of true positives and false positives of the tools under benchmarking, as discussed next.

In practice, the signatures and monitoring approach provides information that is not available to the penetration testing tools under benchmarking, thus it is expected to detect more vulnerabilities and present less false positives. In fact, and based on the precise detection of signatures, no false positives are expected (see Section 4.2). Thus, the vulnerabilities identified using interface monitoring can be effectively used as a baseline for evaluating other tools.

The vulnerability detection coverage is the percentage of real vulnerabilities that are detected by a tool. Assuming that the number of vulnerabilities reported by the signatures and monitoring approach is a valid estimation of the total number of existing vulnerabilities, then the percentage of those vulnerabilities that are reported by a given penetration testing tool is also a valid estimation for its vulnerability detection coverage. In a similar way, we can estimate the false positives rate, which represents the percentage of vulnerabilities reported by the tool that in fact do not exist. Considering that the set of vulnerabilities detected using our approach does not include false positives (guaranteed by the use of adequate signatures), we can estimate the false positives rate of a penetration testing tool by calculating the difference between the vulnerabilities reported by such tool and the vulnerabilities identified via interface monitoring.

To better understand our proposal, let’s take a simple scenario. Consider that the signatures system is able to detect 10 SQL Injection vulnerabilities in a given web

service and that a penetration testing tool A detects 8 of those and 6 more, and that a penetration testing tool B detects 4 of those and 1 more. As shown in Table 6.3 we can use these values to estimate the coverage and false positives of both tools. A similar approach can be used to estimate the metrics of our benchmark (precision, recall, and F-Measure). Note that, the considered total number of vulnerabilities is only an estimated value, as there is no guarantee of perfect detection coverage from the signatures system. This way, the total number of vulnerabilities will be always equal or superior to the estimated number of vulnerabilities and this fact can diminish the importance of the evaluation in two ways.

First, the coverage rates calculated for the evaluated tools may be overestimated. Although this seems a key problem, it is important to stress that the evaluation of the different tools is done for benchmarking purposes (e.g. to select one) and not for assessing actual effectiveness (as this depends on several factors, including the target application, programing language, type of vulnerability, etc.). Thus, taking a relative perspective of the results (rather than an absolute perspective), the overestimation should be equivalent for all the evaluated tools, affecting them in a similar manner, while maintaining a fair comparison.

Second, the false positive rates for the evaluated penetration testers may also be overestimated. Again, although this seems a major issue, in practice the impact will be minor: it is highly probable that a vulnerability detected by a tester will also be detected by our approach as it is based on the internal behavior of the application provided by the interface monitoring. This way, the estimation for the false positives should be close to the real values, which again is adequate for a relative view of the results.

Table 6.3 – Example of coverage and false positive rates estimation.
The values presented are estimated using the results of Sign-WS as the baseline to evaluate the penetration testing tools.

Tool	Estimated Coverage Rate	Estimated False Positive Rate
PTA	8 / 10 = 80%	6 / (8 + 6) ~= 43%
PTB	4 / 10 = 40%	1 / (4 + 1) = 20%

It is important to note that the proposed benchmark can be easily extended to other types of injection vulnerabilities. The only constraint is that the benchmark user has to define a workload containing other types of vulnerabilities and then characterize those vulnerabilities. Although in this benchmark we are targeting only SQL Injection, the Sign-WS technique can also be used to detect other Injection vulnerabilities (see Section 4.2 for additional details).

6.4 Benchmarking Properties

Computer benchmarking is primarily an experimental approach (Gray 1993). As an experiment, its acceptability is largely based on two salient facets of the experimental method: 1) the ability to reproduce the observations and the measurements, either on a deterministic or on a statistical basis, and 2) the capability of generalizing the results through some form of inductive reasoning. The first aspect (ability to reproduce) gives confidence in the results and the second (ability to generalize) makes the benchmark results meaningful and useful beyond the specific setup used in the benchmarking process.

In practice, benchmarking results are normally reproducible in a statistical basis. On the other hand, the necessary generalization of the results is inherently related to the representativeness of the benchmark experiments. The notion of representativeness is manifold and touches almost all the aspects of benchmarking, as it really means that the conditions used to obtain the measures are representative of what can be found in the real world.

The key aspect that distinguishes benchmarking from existing evaluation and validation techniques is that a benchmark fundamentally represents an agreement (explicit or tacit) that is accepted by the computer industry and by the user community. This technical agreement is in fact the key that turns a benchmark into a standard. In other words, a benchmark is something that the user community and the computer industry accept as representative enough of a given application domain to be deemed useful and to be generally used as a (standard) way of measuring specific features of a computer system and, consequently, a way to compare different systems.

The concept of benchmarking can then be summarized in three words: **representativeness, usefulness, and agreement.** A benchmark must be as representative as possible of a given domain but, as an abstraction of that domain, it will always be an imperfect representation of reality. However, the objective is to find a useful representation that captures the essential elements of the application domain and provides practical ways to characterize the computer features that help the vendors/integrators to improve their products and help the users in their purchase decisions.

To achieve acceptance by the computer industry or by the user community a benchmark should fulfill a set of key properties (Gray 1993): representativeness, portability, repeatability, non-intrusiveness, and simplicity of use. These properties were taken into account from the beginning of the definition of the components of the proposed benchmarks and were validated after the benchmark has been completely defined (see sections 7.2.4 and 7.3.4).

To be credible, a benchmark for vulnerability detection tools must report similar results when run more than once over the same tool. However, **repeatability** has to

be understood in statistical terms, as it might be impossible to reproduce exactly the same conditions concerning the tool and the web services state during the benchmark run. In practice, small deviations in the measurements in successive runs are normal and just reflect the non-deterministic nature of web applications.

Another important property is **portability**, as a benchmark must allow the comparison of different tools in a given domain. In practice, the workload is the component that has more influence on portability, as it must be able to exercise the vulnerability detection capabilities of a large set of tools in the domain.

In order to report relevant results, a benchmark must represent real world scenarios in a realistic way. In our work, **representativeness** is mainly influenced by the workload, which must be based on realistic code and must include a realistic set of vulnerabilities. This can more easily be taken into account in the case of benchmarks based on a predefined workload, as it is possible to address representativeness issues during the benchmark specification. However, this may be an issue in the case of user-defined workloads, as the benchmark user may not be aware of the representativeness issues of the services considered and, consequently, of the results obtained.

A benchmark must require minimum changes (or no changes at all) in the target tools. If the implementation or execution of the benchmark requires changes in the tools (either in the structure or in the behavior) then the benchmark is **intrusive** and the results might not be valid.

Finally, to be accepted, a benchmark must be as **easy to implement and run** as possible. Ideally, the benchmark should be provided in a form ready to be used or, if that is not possible, as a document specifying in detail how the benchmark should be implemented and executed. In addition, the benchmark execution should take the smallest time possible (preferably not more than a few hours per tool). This is obviously easier to achieve in benchmarks based on a predefined workload, as in the case of user-defined workloads the benchmark user has the added work of defining and characterizing the workload (if possible, the such benchmark should include guidelines and/or tools to facilitate this task).

6.5 Conclusion

This chapter presented a generic approach to **define benchmarks for vulnerability detection tools in web services**. It specifies the guidelines for the definition of the benchmark components (i.e. workload, metrics and procedure) and the steps necessary to implement concrete benchmarks, focusing on two key metrics: precision and recall. This approach has been used to define two concrete benchmarks targeting tools able to detect SQL Injection vulnerabilities.

The first benchmark is **based on a predefined workload** that consists of large set of web services adapted from standard performance benchmarks, and including versions of the services both with and without vulnerabilities. Being based on a predefined set of services, the main limitation of this benchmark is that it is not fully protected against "*gaming*".

The second benchmark **leaves to the user the responsibility for defining that workload**, thus avoiding the "*gaming*" problem. The problem of such approach is that it leaves to the user the task of characterizing the workload in terms of the existing vulnerabilities. To support the user in this task, the benchmark includes a procedure and a tool to identify injection vulnerabilities in the web services, providing a good estimation of the benchmark metrics.

The experimental evaluation of these benchmarks will be presented in Chapter 7. In practice, the benchmarks were used to evaluate and rank several tools with capabilities to detect SQL injection vulnerabilities in web services, including the tools that implemented the techniques presented in Chapter 4.

Finally, this chapter discussed the **benchmarking properties** that guarantee that its results are reproducible and can be generalized in that specific domain. To be accepted by the computer industry and by the user community a benchmark should fulfill such a set of properties, namely: representativeness, portability, repeatability, non-intrusiveness, and simplicity of use. These properties will be discussed in more detail for both benchmarks together with the results in Chapter 7.

Chapter 7
Case Studies

This chapter presents the practical application and experimental evaluation of the techniques and tools proposed in the previous chapters. Four case studies are presented with the objective of assessing how effective are the vulnerability detection techniques and tools proposed. Furthermore, the case studies show how to assess and compare vulnerability detection tools using the benchmarks presented in Chapter 6.

The first case study focuses on the use of well known web security scanners in publicly available web services. In these experiments we use four web security scanners to identify security flaws in 300 publicly available web services. The purpose of this experiment is twofold. In the one hand, it allows understanding the **effectiveness of well-known web security scanners**. On the other hand, it allows understanding the **most common types of vulnerabilities** in web services environments, providing some insight on the priority of the types to be addressed.

The second case study presented is a benchmarking campaign conducted using the VDBenchWS-pd benchmark to evaluate and rank a large set of vulnerability detection tools including web security scanners, static code analyzers, and the three tools proposed in Chapter 4. The objectives of this campaign are: **assessing the effectiveness of the vulnerability detection tools proposed in this work** (comparing to other existing ones) and **validating the proposed benchmark**.

The third case study is another benchmarking campaign, now using the PTBenchWS-ud benchmark, to evaluate four penetration testing tools, including three commercial scanners and the improved penetration testing tool presented in Chapter 4. This campaign was conducted with the objective of **validating the PTBenchWS-ud benchmark as an alternative** that can be effectively used in specific scenarios for comparing penetration testing tools (recall that this benchmark is based on a user-defined workload).

The final case study demonstrates the use of the SOA-Scanner presented in Chapter 5. This case study uses a simple **infrastructure based on SOAP web services having injection vulnerabilities**. In practice, this infrastructure is a subset of jSeduite SOA (Delerce-Mauris et al. 2009). Although the resulting infrastructure is quite simple, it allows demonstrating all the different scenarios and thus, exploring the functionalities of the tool.

The outline of this chapter is as follows. The next section presents the first case study where web security scanners are used to detect vulnerabilities in publicly available web services. Section 7.2 presents a benchmarking campaign using VDBenchWS-pd to evaluate and compare several tools, including the ones proposed in Chapter 4. Section 7.3 presents a benchmarking campaign using PTBenchWS-ud to evaluate and compare four penetration testing tools. Section 7.4 presents the case study based on a service based infrastructure to demonstrate the effectiveness of the integrated tool proposed in Chapter 5. Finally, Section 7.5 concludes the chapter.

7.1 Case Study #1: Assessing Public Web Services

As discussed before, commercial web security scanners are widely used by developers and are considered as representative of the state of the art in web applications black-box testing (Acunetix 2008a). These scanners are regarded as an easy way to test applications searching for security vulnerabilities. However, previous research suggests that the effectiveness of scanners in the detection of vulnerabilities varies a lot, most times providing unsatisfactory results (Fonseca, Vieira, and Madeira 2007). This section presents an experimental campaign conducted to understand the strengths and limitations of penetration testing in public web services and to try to identify the common types of vulnerabilities in such environments. In summary, the experiments were conducted to answer to the following three questions:

- What is the **detection coverage** (percentage of total existing vulnerabilities detected by the tool) of the vulnerability scanners tested?
- What is the **false positives rate** (percentage of vulnerabilities identified by the tool but that do not really exist) of the web vulnerability scanners tested?
- What are the most **common types of vulnerabilities** in web services environments?

The experimental study consisted of four steps:

1. **Preparation:** select the security scanners to use and the web services to scan.
2. **Execution:** use the security scanners to scan the services in order to identify potential vulnerabilities. The tests were executed with the scanners

configured to use their most complete profile (i.e. to use all the tests that they have available for web services assessment).

3. **Verification:** perform manual testing to confirm that the vulnerabilities identified by the scanners do exist (i.e. are not false positives).
4. **Analysis:** analyze the results obtained and systematize the lessons learned.

Four commercial web vulnerability scanners widely used were selected, including two different versions of a specific brand. The three brands, introduced in Chapter 2, are: HP WebInspect (HP 2008), IBM Rational AppScan (IBM 2008), and Acunetix Web Vulnerability Scanner (Acunetix 2008a). For the results presentation we decided not to mention the brand and the versions of the scanners to assure neutrality and because commercial licenses do not allow in general the publication of tool evaluation results. This way, the scanners are referred in this section as VS1.1, VS1.2, VS2, and VS3 (without any order in particular). Vulnerability scanners VS1.1 and VS1.2 refer to the two versions of the same product (being VS1.2 the most recent).

Three hundred publicly available web services were randomly selected. The reason for random selection is twofold: allow a fair comparison between the scanners and allow us to gather information about vulnerability distribution in web services context without biasing the results. The set of web services included services implemented with several technologies (.NET, Java, php, etc.) and services owned by different relevant parties, including Microsoft, Google, and Xara.

7.1.1 Overall Results

Table 7.1 presents the overall results of the study. For each scanner it is presented the total number of vulnerabilities reported and the number of services in which those vulnerabilities were found. The scanners pointed six different types of vulnerabilities (already introduced in Chapter 2).

As we can see, different scanners report different types of vulnerabilities. This is a first indicator that tools implement different forms of vulnerability identification and that the results from different scanners may be difficult to compare. Some additional observations are:

- Tool VS1.1 and VS1.2 (two different versions of the same brand) are the only ones that detected XPath Injection vulnerabilities. An important aspect is that, when compared to SQL Injection, the number of XPath-related vulnerabilities is quite small. In fact, XPath vulnerabilities were detected in a single service, suggesting that most web services make use of a database instead of XML documents to store information.
- Tools VS1.1 and VS1.2 detected a code execution vulnerability. This is a particularly critical vulnerability that allows attackers to execute code in the

server. After discovering this vulnerability we performed some manual tests and we were amazed by the possibility of executing operating system commands and get the corresponding answer in a readable format.

- VS3 was the only one pointing vulnerabilities related to buffer overflow, username and password disclosure, and server path disclosure.
- SQL Injection is the only type of vulnerability that was reported by the four scanners used. However, different scanners reported different vulnerabilities in different web services. In fact, the number of SQL Injection vulnerabilities detected by VS1.1 and VS1.2 is much higher than the number of vulnerabilities detected by VS2 and VS3.

Table 7.1 – Overall results obtained.

For each type of vulnerability scanner, the table shows the number reported by each scanner and the number of web services with vulnerabilities.

Vulnerability Types	VS1.1		VS1.2		VS2		VS3	
	# Vuln.	# WS	# Vuln.	# WS	# Vuln.	# WS	# Vuln.	# WS
SQL Injection	217	38	225	38	25	5	35	11
XPath Injection	10	1	10	1	0	0	0	0
Code Execution	1	1	1	1	0	0	0	0
Possible Parameter Based Buffer Overflow	0	0	0	0	0	0	4	3
Possible Username or Password Disclosure	0	0	0	0	0	0	47	3
Possible Server Path Disclosure	0	0	0	0	0	0	17	5
Total	**228**	**40**	**236**	**40**	**25**	**5**	**103**	**22**

As SQL Injection is the only type of vulnerability reported by all the scanners, we look at this vulnerability type in more detail. The intersection areas of the circles in Figure 7.1 represent the number of vulnerabilities detected by more than one scanner (the number of vulnerabilities detected is shown; zero is the value when no number is presented). Note that the area of each circle is roughly proportional to the number of vulnerabilities detected, but there is no correspondence between the size of the intersection areas and the number of vulnerabilities (it is too complex to represent graphically).

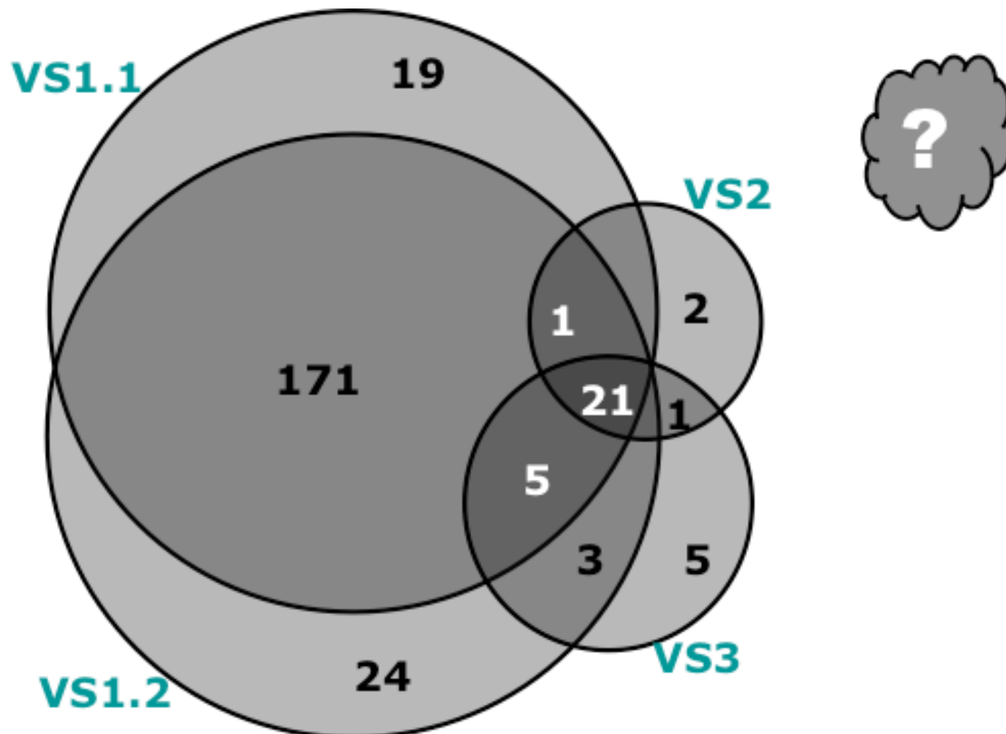


Figure 7.1 – Interception of SQL Injection vulnerabilities reported.
The circles areas are proportional to the number of vulnerabilities reported by the respective scanner. The cloud represents the unknown vulnerabilities.

Figure 7.1 clearly shows that the four scanners detected different sets of SQL Injection vulnerabilities and the differences are considerable, pointing again to relatively low detection coverage of each vulnerability scanner individually. In fact, even for VS1.1 and VS1.2, two consecutive versions of the same scanner, there are considerable differences. VS1.1, the older version, reported 19 SQL Injection vulnerabilities that were not reported by VS1.2. On the other hand, VS1.2 reported 27 vulnerabilities that were not reported by VS1.1.

7.1.2 False Positive Analysis

The results presented so far do not consider false positives (i.e. situations where tools detected a vulnerability that in the reality does not exist). However, it is well known that false positives are very difficult to avoid. This way, we decided to manually confirm the existence (or not) of each vulnerability reported.

Confirming the existence of a vulnerability without having access to the source code is quite difficult. Thus, we defined a set of rules and corresponding checks to classify

the vulnerabilities detected by the penetration testing tools in three groups: a) False positives, b) Confirmed vulnerabilities, and c) Doubtful.

Reported vulnerabilities were classified as **false positives** when meeting at least one of the following cases:

- For SQL Injection vulnerabilities, if the error/answer obtained is related to an application robustness problem and not to a SQL command (e.g. a NumberFormatException).
- The error/value in the web service response is not caused by the elements "injected" by the scanner. In other words, the same problem occurs when the service is executed with valid inputs.
- For path and username/password disclosure, the information returned by the service is equal to the information submitted by the client (e.g. the vulnerability scanner) when invoking the web service. In other words, there is no information disclosure.

Reported vulnerabilities were classified as **confirmed vulnerabilities** if satisfying one of the following conditions:

- For SQL Injection vulnerabilities, if it is possible to observe that the SQL command executed was invalidated by the values "injected" by the scanner (or manually). This is possible if the SQL command or part of it is included in the web service response (e.g. stack trace).
- For SQL Injection vulnerabilities, if the “injected” values lead to exceptions raised by the database server.
- If it is possible to access unauthorized services or web pages (e.g. by breaking the authentication process using SQL Injection).
- For Path disclosure, if it is possible to observe the location of folders and files in the server.
- For XPath Injection, if the “injected” values lead to exceptions raised by the XPath parser.
- For Buffer Overflow, if the server does not answer to the request or raises an exception specifically related to buffer overflow.

If none of these rules can be applied then there is no way to confirm whether a vulnerability really exists or not. These cases were classified as doubtful. Figure 7.2 shows the results for SQL Injection vulnerabilities (the only type detected by all the scanners tested).

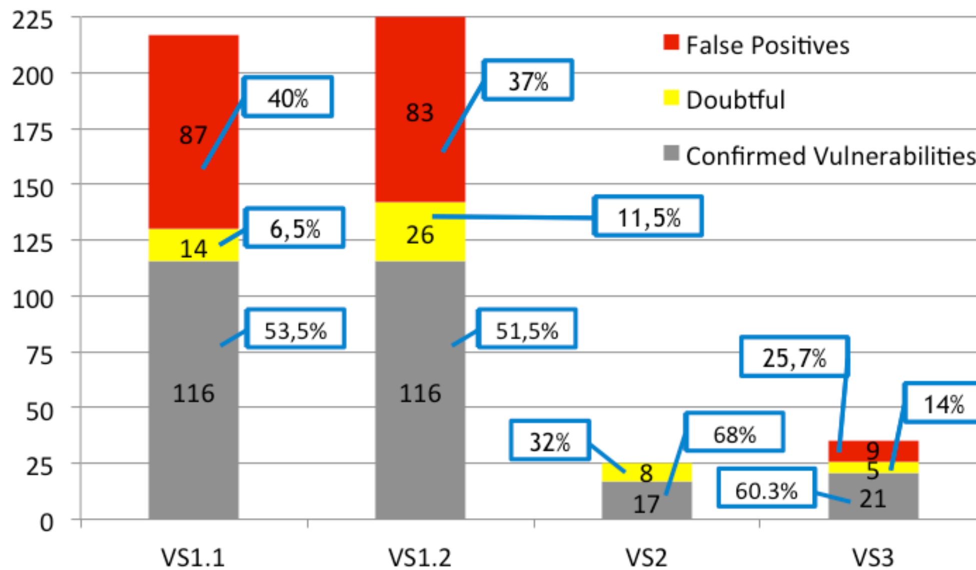


Figure 7.2 – False positives for SQL Injection in the public services.
Doubtful cases are cases in which the vulnerability was not confirmed but also it was not possible to rule it as a false positive.

As shown, the percentage of vulnerabilities that we were not able to confirm (doubtful cases) is low for VS1.1, VS1.2, and VS3 (always less than 15%), but considerably high for VS2 (32%). This means that the false positive results are relatively accurate for the first three tools, but it is an optimistic figure (zero false positives) for scanner VS2. Obviously, we can also read the false positive results shown in Figure 7.2 as a range, going from an optimistic value (confirmed false positives) to a pessimistic value (confirmed false positives + doubtful cases).

The number of (confirmed) false-positives is high for scanners VS1.1 and VS1.2, and is also high for VS3, in relative terms. Scanner VS2 shows zero confirmed false positives, but it detected a fair percentage (8 out of 25) of vulnerabilities that were classified as doubtful, thus a pessimistic interpretation of results is that 8 out of 25 vulnerabilities may be false positives. Obviously, the low number of vulnerabilities detected by VS2 and VS3 (25 and 35 respectively) also limits the absolute number of false positives.

Table 7.2 presents the false positive results for the other vulnerabilities. In this case, we were able to confirm the existence (or inexistence) of all vulnerabilities and no doubts remained. An interesting aspect is that all XPath injection and Code Execution vulnerabilities were confirmed. On the other hand, all vulnerabilities related to username and password disclosure were in fact false positives (in all cases the username/password information returned is equal to the one sent by the scanner).

Table 7.2 – False positives for other vulnerability types.
As these types of vulnerabilities were reported by only one brand of scanners, for each type of vulnerability the table also indicates the scanner that reported.

Vulnerability	Scanner	Confirmed	F. Positives
XPath Injection	VS1.1 & VS1.2	10	0
Code Execution	VS1.1 & VS1.2	1	0
Possible Parameter Based Buffer Overflow	VS3	1	3
Possible Username or Password Disclosure	VS3	0	47
Possible Server Path Disclosure	VS3	16	1

Due to the large percentage of false positives observed for SQL Injection vulnerabilities, we decided to repeat the analysis of the interceptions of the vulnerability report sets. Figure 7.3 presents the SQL Injection vulnerabilities intersections after removing the false positives. The doubtful situations were in this case considered as existing vulnerabilities (i.e. optimistic assumption from the point of view of scanners detection effectiveness).

Results clearly show that, even after removing the false positives, the four tools report different vulnerabilities. An interesting result is that three vulnerabilities were detected by VS1.1 and were not detected VS1.2 (the newer version of the scanner). The reverse also happens for 15 vulnerabilities, which is expectable as a newer version is anticipated to detected more vulnerabilities than an older one (but that should happen without missing any of the vulnerabilities identified by the older version, which was not the case). These results called our attention and we tried to identify the reasons. After analyzing the detailed results we concluded that all of these 18 vulnerabilities are in the group of the doubtful ones (maybe they are really false positives, but we were not able to demonstrate that), preventing us from drawing a definitive conclusion.

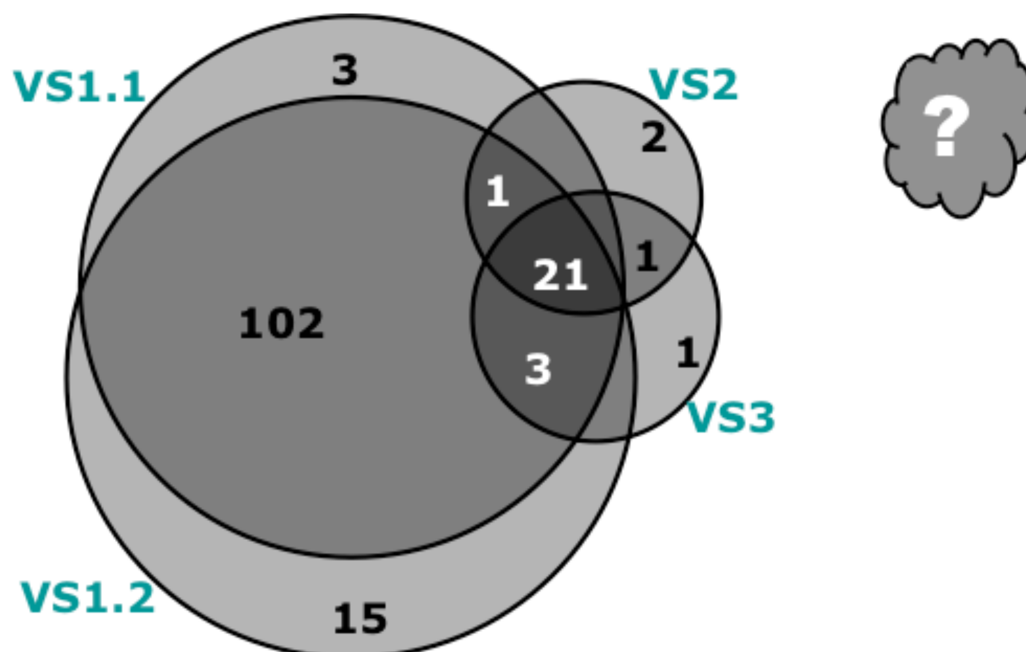


Figure 7.3 – Interception of SQL Injection vulnerabilities without False Positives.
The circles areas are proportional to the number of vulnerabilities represented. The cloud represents the unknown vulnerabilities.

7.1.3 Coverage Analysis

A key aspect is to understand the detection coverage of the vulnerabilities detected. Detection coverage compares the number of vulnerabilities detected against the total number of vulnerabilities. Obviously, in our case it is impossible to know how many vulnerabilities were not disclosed by any of the scanners (we do not have access to the source code). Thus, it is not possible to calculate the coverage. However, it is still possible to make a relative comparison based on the data available.

In practice, we know the total number of vulnerabilities detected (which correspond to the union of the vulnerabilities detected by the four scanners after removing the false positives) and the number of vulnerabilities detected by each individual scanner. Based on this information it is possible to get an optimistic coverage indicator for each scanner (i.e. the real coverage will be lower than the value presented). Obviously, this is relevant only for SQL Injection vulnerabilities as it is the only type that is detected by all the scanners.

Table 7.3 presents the coverage results. As shown, 149 different SQL Injection vulnerabilities were detected (as before, we decided to include the doubtful situations as existing vulnerabilities). Each scanner detected a subgroup of these

vulnerabilities, resulting in partial detection coverage. VS1.1 and VS1.2 presented quite good results. On the other hand, the coverage of VS2 and VS3 is very low.

Table 7.3 – Coverage for SQL Injection vulnerabilities.
As it is not possible to know the total existing number of vulnerabilities, the values presented are probably overestimated.

Scanner	# SQL Injection Vulnerabilities	Detection Coverage
VS1.1	130	87.2%
VS1.2	142	95.3%
VS2	25	16.8%
VS3	26	17.4%
Total	149	100.0%

7.1.4 Lessons Learned

The results presented before allow us to observe some interesting aspects. The first observation is that different scanners detected different types of faults. SQL Injection was the only type that was detected by all scanners. The two scanners of the same brand (VS1.1 and VS1.2) were the only ones that detected XPath and code execution vulnerabilities. Only one scanner (VS3) detected vulnerabilities related to buffer overflow, username and password disclosure, and server path disclosure. VS2 only detected SQL Injection vulnerabilities.

SQL Injection vulnerabilities are the dominant type in the web services tested (see Figure 7.4). However, different scanners detected different vulnerabilities of this type. In fact, VS1.1 and VS1.2 detected a huge number of vulnerabilities (215 and 225 respectively) while VS2 and VS3 detected a very low number (25 and 35 respectively). Additionally, SQL Injection vulnerabilities together with the other types of injection vulnerabilities reported (XPath Injection and Code Execution) represent approximately 90% of the vulnerabilities detected.

A key observation is the very large number of false positives. In fact, for three of the scanners the percentage of false positives was more than 25%. VS2 presented zero false positives, but 8 out of the 25 SQL Injection vulnerabilities detected by this scanner remained as doubtful (i.e. could not be manually confirmed as real vulnerabilities nor as false positives). This reduces the confidence on the precision of the vulnerabilities detected.

A very low coverage, lower than 18%, was observed for two of the scanners (VS2 and VS3), while the other two scanners (VS1.1 and VS1.2) present a coverage superior to 87%. Note that this value represents an optimistic coverage, as the real

coverage of the tested scanners (at least for the 300 web services used in the experiments) is definitely lower than the value observed, with many vulnerabilities probably remaining undetected.

To sum up, the results show that selecting a vulnerability scanner to use for detecting vulnerabilities in web services is a very difficult task. First, different scanners detect different types of vulnerabilities. Second, the number of false positives is quite high, reducing the confidence on the scanners' results. Finally, the coverage is in some cases very low, suggesting that many vulnerabilities probably remain undetected. This also highlights the need that users have for techniques that allow to evaluate and compare the effectiveness of different vulnerability detection tools in a fair way, as addressed previously in this thesis.

The final remark goes to the final distribution of vulnerabilities per type, presented in Figure 7.4, after removing the confirmed false positives but including the doubtful cases (i.e. optimistic evaluation of the scanners). As the doubtful cases only affect the SQL Injection, it means that the number of SQL injection vulnerabilities could be overestimated. Scanners have found 177 different vulnerabilities in 25 different services, which represent approximately 8.33% of the tested services. As mentioned before, the predominant vulnerability is SQL Injection, representing 84.18% of the vulnerabilities found, with injection vulnerabilities representing approximately 90% of the vulnerabilities detected. This is a very important observation due to the high number of cases found and the high severity of this type of vulnerability.

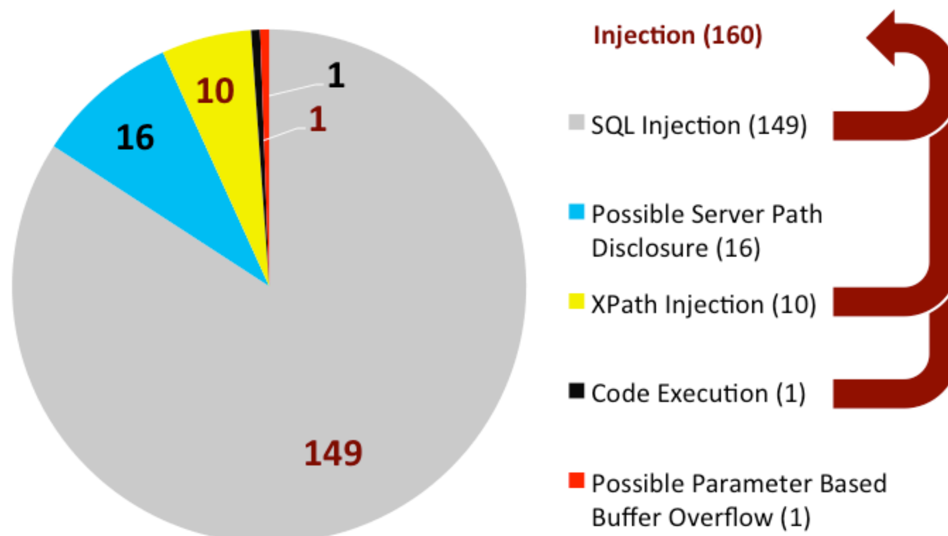


Figure 7.4 – Vulnerabilities detected distributed per type.
The values represented exclude false positives. The sum of injection vulnerabilities (~90%) further highlights the importance of this type of vulnerabilities.

7.2 Case Study #2: Using VDBenchWS-pd to Benchmark Vulnerability Detection Tools

Using the VDBenchWS-pd to benchmark a set of vulnerability detection tools is basically a straightforward process that consists in following the steps defined in the benchmark procedure (see Section 6.1.3). As specified, the tools under benchmarking must be selected in **Phase 1: Preparation**. Phase 1 also specifies the step of selecting and characterizing a workload, however this is only necessary in the case of benchmarks based on a user defined workload, which is not the case of VDBenchWS-pd. Table 7.4 summarizes the tools used in this experimental campaign, which are able to detect SQL Injection vulnerabilities in SOAP web services, the target domain of the benchmark.

Table 7.4 – Tools under benchmarking.
The third party penetration testing and static code analyzers are referred to throughout the section by using the codes VS1, VS2, VS3, SA1, SA2, and SA3.

Provider	Tool	Technique
HP	WebInspect	Penetration testing / Identify vulnerable inputs
IBM	Rational AppScan	
Acunetix	Web Vulnerability Scanner	
Univ. Coimbra	IPT-WS (see Section 4.1)	
Univ. Coimbra	Sign-WS (see Section 4.2)	Attack signatures and interface monitoring / Identifies vulnerable inputs
Univ. Maryland	FindBugs	Static code analysis / Identify vulnerable queries
SourceForge	Yasca	
JetBrains	IntelliJ IDEA	
Univ. Coimbra	RAD-WS (see Section 4.3)	Anomaly detection / Identifies vulnerable queries

As shown, four **penetration testing tools** have been benchmarked, including three well-known commercial tools, namely: HP WebInspect (HP 2008), IBM Rational AppScan (IBM 2008), and Acunetix Web Vulnerability Scanner (Acunetix 2008a). These tools were introduced in Section 2.3.1. The last penetration tester considered is the improved penetration testing (IPT-WS) tool presented in Section 4.1.

Three vastly used **static code analyzers** that provide the capability of detecting vulnerabilities in Java applications' source or *bytecode* have also been considered in

this study, namely: FindBugs (University of Maryland 2009), Yasca(Scovetta 2008), and IntelliJ IDEA (JetBrains 2009). These tools were introduced in Section 2.3.2.

The last two tools used are also proposed in this thesis. Sign-WS is presented in Section 4.2 and implements a technique based on attack signatures and interface monitoring. RAD-WS is presented in Section 4.3 and combines **runtime anomaly detection** with penetration testing for uncovering SQL Injection vulnerabilities in web services.

The second phase of the benchmark consists of running the selected tools over the workload code (**Phase 2. Execution**). Basically, the tools are used to detect vulnerabilities in the web services of the workload. An important aspect is that, in the case of testing tools and when allowed, information about the domain of each web service input parameter was provided. If the tool requires the user to set an exemplar invocation per operation, the exemplar respected the input domains of operation. All the tools in this situation used the same exemplar to guarantee a fair evaluation. Similarly, during the experiments the static analyzers were configured to fully analyze the services code. For the analyzers that use binary code, the deployment ready version was used.

The vulnerabilities detected were then compared with the existing ones and used to calculate the benchmark metrics and rank the tools (**Phase 3. Comparison**). Vulnerabilities correctly detected by the tools were counted as true positives. Vulnerabilities detected by the tools that did not match any of the known ones were manually analyzed before being classified as false positives. Although this is not a step included in the benchmarking approach, it was useful to validate the result of the code reviews conducted by the security experts (during the definition of the benchmark, as detailed in Chapter 6). The outcome was that no additional vulnerabilities were identified by any of the tools (i.e. all the vulnerabilities reported by the tools were already known or were in fact false positives), which gives us some guarantee that the code reviews were conducted in an appropriate manner and identified all the vulnerabilities.

7.2.1 Overall benchmarking results

Table 7.5 presents the overall benchmarking results. As we can see, the anomaly detection tool (RAD-WS) is the one that presents the higher F-Measure. Additionally, two of the static code analysis tools (SA1 and SA2) present better results than the penetration testing tools. SA3 and VS3 are the tools with the lowest F-Measure.

Table 7.5 – VDBenchWS-pd Benchmarking results.

After the execution of the benchmark it is possible to calculate the measures for each tool.

Tool	F-Measure	Precision	Recall
VS1	0.378	0.455	0.323
VS2	0.297	0.388	0.241
VS3	0.037	1.000	0.019
IPT-WS	0.338	0.567	0.241
Sign-WS	0.851	1.000	0.741
SA1	0.691	0.923	0.552
SA2	0.780	0.640	1.000
SA3	0.204	0.325	0.149
RAD-WS	0.885	1.000	0.793

The benchmark measures can be used to rank the tools under benchmarking (**Step 3.b: Ranking and Selection**) according to three criteria: precision (focus on the balance between true positives and false positives), recall (focus on the true positives rate), and F-Measure (focus on the balance between precision and recall). Table 7.6 presents a possible ranking for the tools. We divide the ranking in two, considering the approach used to report vulnerabilities (vulnerable inputs or vulnerable SQL queries), as defining a single ranking for tools that report vulnerabilities in different ways may not be meaningful (nevertheless, the benchmark measures allow such a ranking). Tools presented in the same cell are ranked in the same position due to the similarity of the results.

Table 7.6 – VDBenchWS-pd tools ranking.

The tools can be ranked according to three metrics, depending on the objectives of the benchmark user.

<table>
<tr><th></th><th>Criteria</th><th>1st</th><th>2nd</th><th>3rd</th><th>4th</th><th>5th</th></tr>
<tr><td rowspan="3">Inputs</td><td>F-Measure</td><td>Sign-WS</td><td>VS1</td><td>IPT-WS</td><td>VS2</td><td>VS3</td></tr>
<tr><td>Precision</td><td>Sign-WS</td><td>VS3</td><td>IPT-WS</td><td>VS1</td><td>VS2</td></tr>
<tr><td>Recall</td><td>Sign-WS</td><td>VS1</td><td colspan="2">VS2/IPT-WS</td><td>VS3</td></tr>
<tr><th></th><th>Criteria</th><th>1st</th><th>2nd</th><th>3rd</th><th colspan="2">4th</th></tr>
<tr><td rowspan="3">Queries</td><td>F-Measure</td><td>RAD-WS</td><td>SA2</td><td>SA1</td><td colspan="2">SA3</td></tr>
<tr><td>Precision</td><td>RAD-WS</td><td>SA1</td><td>SA2</td><td colspan="2">SA3</td></tr>
<tr><td>Recall</td><td>SA2</td><td>RAD-WS</td><td>SA1</td><td colspan="2">SA3</td></tr>
</table>

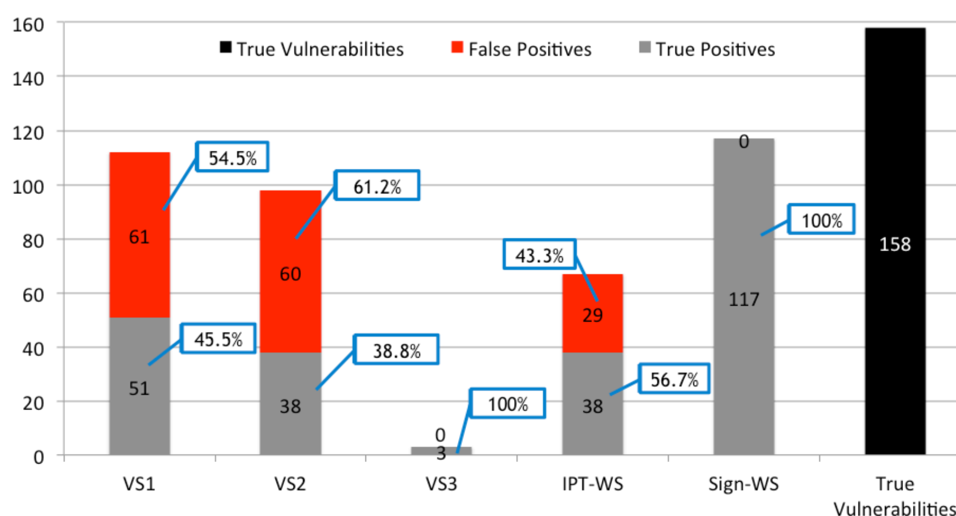
As we can see in Table 7.6, RAD-WS is the most effective tool considering both F-Measure and precision. However, the most effective tool when we consider recall is SA2, being RAD-WS the second best. VS3 seems to be the least effective tool in terms of F-Measure and recall. However, it has a very good precision (in fact, it reported no false positives, but detected only 3 of the existing vulnerabilities). Excluding SA3,

static analysis appears to be a better option than penetration testing. The following subsections discuss the benchmark results in more detail.

7.2.2 Results for Tools that Report Vulnerable Inputs

Figure 7.5 shows the vulnerabilities reported by the penetration testing tools and Sign-WS. Columns 1, 2, and 3 show the results for the commercial tools and the last bar in the graph presents the total number of vulnerabilities in the workload parameters.

As we can see, the different tools reported a different number of vulnerabilities and the coverage for the commercial tools is always under 35%. Among these, VS1 identified the higher number of vulnerabilities (≈32% of the total vulnerabilities). However, it also reports a very higher number of false positives (≈54%). The very low number of vulnerabilities detected by VS3 can be partially explained by the fact that this tool does not allow the user to set any information about input domains, nor it accepts any exemplar request. This means that the tool generates a completely random workload that, probably, is not able to test parts of the code.



Tool	Detection Coverage	False Positive Rate
VS1	32.28%	54.46%
VS2	24.05%	61.22%
VS3	1.90%	0.00%
IPT-WS	24.05%	43.28%
Sign-WS	74.05%	0.00%

Figure 7.5 – VDBenchWS-pd results for pen. testing and Sign-WS.
The percentages presented in the image are relative to the vulnerabilities reported.

Comparing our tools with the commercial scanners, we can observe that Sign-WS consistently present better results, both in terms of coverage and false positives, largely outperforming any of the penetration tester, including the commercial tools. Sign-WS was able to detect 117 of a total of 158 vulnerabilities (74%), presenting much higher detection coverage than any of the commercial penetration testers. This suggests that our approach is an effective alternative to perform detection of SQL Injection vulnerabilities in web services. As expected, the increased visibility on the web service interfaces, provided by the use of signed attacks and increased interface monitoring, allows the approach to detect vulnerabilities that otherwise would not be possible to detect. In other words, while the scanners are limited to the user point of view, Sign-WS takes advantage of the information added by monitoring other interfaces.

The second observation is the fact that the tool did not report false positives. In the case of two of the commercial penetration testers (VS1 and VS2), more than 54% of the vulnerabilities reported are, in fact, false positive alarms (for VS3 the number of false positives is 0, but the tool only detects 3 vulnerabilities). This increases the confidence in the vulnerabilities detected by Sign-WS in future campaigns. As mentioned before, the rare scenario where false positives would manifest is when tokens similar to the signatures (both normal and reversed) are used in the construction of application's queries. That is not the case in this set of services and also in the large majority of applications (anyway, the tool user is able to configure the signature model, which may allow avoiding matching similar keywords). This way, we do believe that these results can be generally reproduced in real world web services.

The differences between the performance of Sign-WS justify the use of this tool in the PTBenchWS-ud benchmark (see Section 6.3) for helping the user in the characterization of the workload. In fact, although the tool is not able to detect all the existing vulnerabilities, the capacity to detect a much higher number of vulnerabilities than all penetration testers, together with the precision presented (no false positive reported) makes it good enough to produce the baseline results for penetration testing evaluation. Next section will discuss this in detail.

The tool based on improved penetration testing (IPT-WS) presents better results than two of the commercial tools (VS2 and VS3), but presents a lower coverage than VS1 (≈24% against ≈32%). However, in terms of false positives IPT-WS performs better than VS1 (≈43% and ≈54%, respectively). A detailed analysis of the results and of the web services under testing showed that IPT-WS is better than VS1 on the identification of false positives (i.e. the detection rules it implements are more precise), but is less effective on exercising the target services (the workload is less effective).

Figure 7.6 illustrates the intersection of the vulnerabilities detected by the different tools. As before, the areas of the circles are roughly proportional to the number of

vulnerabilities detected by the respective tool, whose name is indicated close to the circle. The same does not happen with the intersection areas, as it would be impossible to represent it graphically.

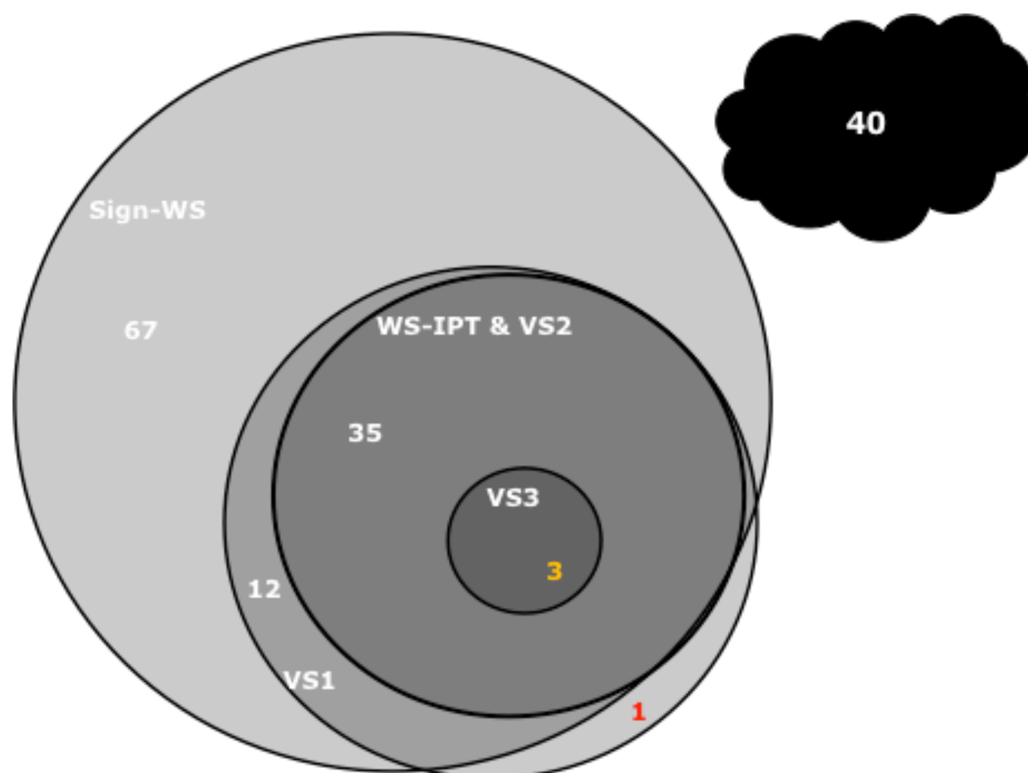


Figure 7.6 – Intersection of vulnerabilities detected.
The circles areas are proportional to the number of vulnerabilities represented. The cloud represents the vulnerabilities that no tool was able to detect.

As we can see, there are 67 vulnerabilities that are detected only by Sign-WS. This represents more than 55% of the total number vulnerabilities reported, emphasizing the advantage of tools with extra information when compared with penetration testing tools. Additionally, only 3 vulnerabilities were detected by all the penetration testing tools, but this number is limited by the low coverage of VS3.

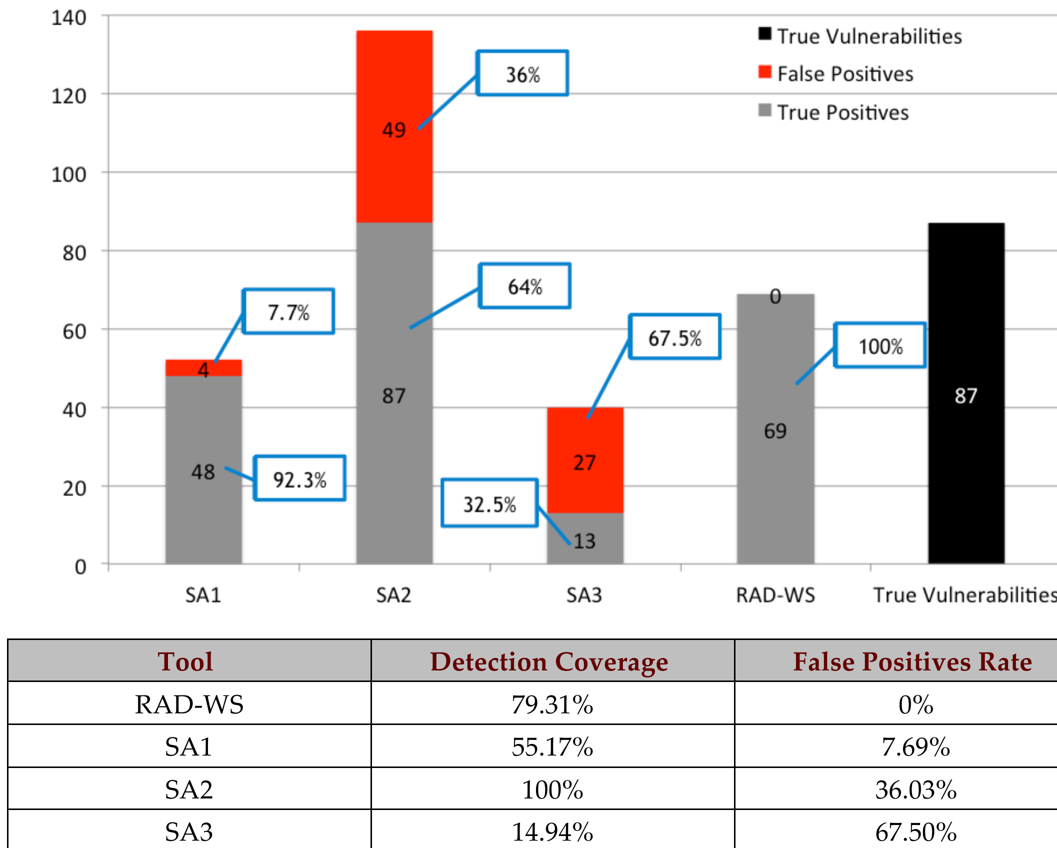
Regarding the penetration testers, VS1 presented the higher detection coverage, as it is able to identify all the vulnerabilities detected by IPT-WS and VS2, plus 13 vulnerabilities. VS1 was also able to identify one vulnerability that no other tool detected. Although none of the tested tools was able to exploit this vulnerability, VS1 uses heuristics to identify vulnerabilities that are in most cases very liberal, reporting vulnerabilities from little evidences that in many cases result in false positives, but in this case resulted in a really existing vulnerability. This is an indication that, although Sign-WS achieves very good coverage, we may need to improve the attackload generation technique (e.g. by including some features available in VS1).

The final remark is relative to the 40 vulnerabilities that were not detected. After a manual analysis of the services, we concluded that many of those are vulnerabilities located in places in the code hard to reach via black-box testing, and the workloads used are not yet complete enough to be able to execute those code paths. There are also situations where a vulnerability is preceded by another very similar vulnerability and so, the second can only be detected after fixing the first. The solution for the first case is to develop new and better ways for generating the workload and attackload. Regarding the second case, we need to apply an iterative process with alternate cycles of vulnerability detection and correction.

7.2.3 Results for Tools that Report Vulnerable SQL Queries

Figure 7.7 shows the number of vulnerable SQL queries identified by the static analyzers and by the RAD-WS tool. Columns 1, 2, and 3 show the results for the third party static code analyzers and the last bar in the graph presents the total number of vulnerable queries in the workload code.

As shown, in general, RAD-WS presents better results than the static analyzers (although it has true positives rate lower than SA2). In fact, RAD-WS presents the best F-Measure value of all the tools benchmarked, detecting almost 80% of the existing vulnerabilities, while avoiding false positives, which constitutes a recall value only lower than SA2 while achieving maximum precision. Considering only the static analyzers, SA2 detected the higher number of vulnerabilities, with 100% of true positives (an excellent result), but identified 49 false positives, which represents ≈36% of the vulnerabilities pointed. The high rate of false positives is, in fact, a problem shared by SA3, which reported more than ≈67% of false positives. The reason is that these tools detect certain patterns that usually indicate vulnerabilities, but many times they detect vulnerabilities that do not exist, due to intrinsic limitations of the static profile of the code analysis.



Tool	Detection Coverage	False Positives Rate
RAD-WS	79.31%	0%
SA1	55.17%	7.69%
SA2	100%	36.03%
SA3	14.94%	67.50%

Figure 7.7 – VDBenchWS-pd results for static analysis and RAD-WS.
The true vulnerabilities are the vulnerabilities identified by the review team. The percentages depicted are relative to the total of vulnerabilities reported by the tool.

Figure 7.8 illustrates the intersection of vulnerable lines detected by the different tools. As we can see, SA2 detects all the vulnerabilities found by the other tools (excluding the false positives) plus 9. Our anomaly detector detects a large number of the vulnerabilities detected by the static code analyzers. Finally, only 11 out of 87 vulnerabilities were detected by all tools.

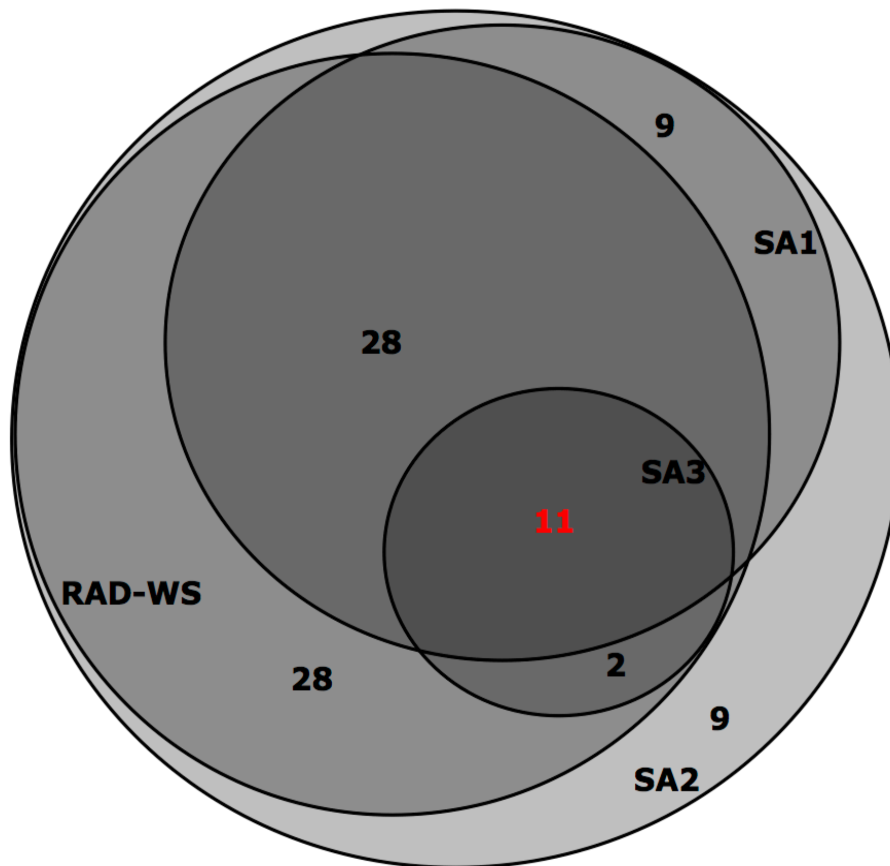


Figure 7.8 – Intersections for static analysis and RAD-WS.
The circles areas are proportional to the number of vulnerabilities represented. All the vulnerabilities were reported by at least one tool, with 11 reported by all tools.

7.2.4 Properties Discussion

The **representativeness** of the workload greatly influences the representativeness of the benchmark. As mentioned before, we are aware of the limitations of the benchmark workload code, as it may not be representative of all the SQL Injection vulnerability patterns found in web services. Our thesis is that the workload should be good enough to allow the comparison of vulnerability detection tools. In fact, what is important is that the benchmark results accurately portray the tools effectiveness in a relative manner. Comparing the benchmarking results with the effectiveness of the tools under benchmarking in different scenarios allows us to check if the benchmark accurately portrays the effectiveness of vulnerability detection tools in a relative manner. This way, we used the tools to detect vulnerabilities in a small set of third-party web services.

Eight web services implementing 28 operations were considered in the experimental evaluation (see Table 7.7). To avoid selecting services that fit the characteristics of the benchmark workload (and thus get biased results) we adopted the services from a previous study (see (Antunes et al. 2009b)). Four of these services implement a subset of the web services specified by the standard TPC-App performance benchmark (Transaction Processing Performance Council 2008), which has also been used to define the VDBenchWS-pd benchmark. However, these versions where implemented by a different developer, inclusively using different technologies (thus resulting in different web services). The remaining four services have been adapted from code publicly available on the Internet. These height services use a database to store data and SQL commands to manage it.

Table 7.7 characterizes the web services (the source code can be found at (Antunes 2013)), including the number of operations per service (#Op), the total lines of code (LoC) per service, the average number of lines of code per operation (LoC/Op), and the average cyclomatic complexity (Lyu 1996) of the operations (Avg. C.). These indicators were calculated using SourceMonitor (Campwood Software 2008). To perform a correct evaluation we extensively reviewed the source code looking for vulnerabilities. The table also includes the characterization of the services in terms of the vulnerabilities existing in the code: 61 vulnerable parameters and 28 vulnerable queries were identified (false positives were eliminated by cross-checking the results from different experts).

Table 7.7 – Third-party web services characterization.

For each service it is presented the vulnerabilities reported, the number of lines of code, the average complexity of the code, the number of vulnerable inputs and lines.

Service		#Op	LoC	LoC/Op	Avg. C.	V. Inputs	V. Lines
TPC-App	ProductDetail	1	105	105.0	6.0	0	0
	NewProducts	1	136	136.0	6.0	1	1
	NewCustomer	1	184	184.0	9.0	15	2
	ChangePaymentMethod	1	97	97.0	11.0	2	1
Public-Code	JamesSmith	5	270	54.0	6.0	20	5
	PhoneDir	5	132	26.4	2.8	6	4
	Bank	5	175	35.0	3.4	4	3
	Bank3	6	377	62.8	9.0	13	12
Total		25	1476	59.0	6.8	61	28

As expected, the measures are not equal to the ones reported by the benchmark. This is normal as this new set of services has different code characteristics and different

SQL Injection vulnerabilities; in fact, what is important is the ranking of the tools, which is presented in Table 7.8 (tools in the same cell are ranked in the same position).

Table 7.8 – Results for third-party web services.
The measures are calculated using the same rules as in the benchmark.

Tool	True Pos.	False P.	F-Measure	Precision	Recall
VS1	31	5	0.639	0.861	0.508
VS2	22	1	0.524	0.957	0.361
VS3	6	0	0.179	1.000	0.098
IPT-WS	28	0	0.629	1.000	0.459
Sign-WS	61	0	1.000	1.000	1.000
SA1	23	7	0.793	0.767	0.821
SA2	28	10	0.849	0.737	1.000
SA3	11	4	0.512	0.733	0.393
RAD-WS	28	0	1.000	0.100	1.000

Comparing this ranking with the one proposed using the benchmark measures (see Table 7.6) we can observe the following: 1) the ranking based on the F-Measure is precisely the same; 2) the ranking based on precision differs for VS2 and VS1 (the services used for the validation are simpler services and represent a lower challenge to the scanners, resulting in higher values of precision for all scanners, leading to this minor change in the ranking); and 3) the ranking based on recall is the same. This suggests that the tools' ranking derived from the benchmarking campaign adequately portrays the relative effectiveness of the tools. However, to prove the property and improve the benchmark representativeness, more vulnerable web services need to be added to the workload.

Table 7.9 – Ranking based on third-party services.
The tools can be ranked according to the three metrics used.

	Criteria	1st	2nd	3rd	4rd	5th
Inputs	F-Measure	Sign-WS	VS1	IPT-WS	VS2	VS3
	Precision	Sign-WS	VS3/IPT-WS		VS2	VS1
	Recall	Sign-WS	VS1	IPT-WS	VS2	VS3
	Criteria	**1st**	**2nd**	**3rd**	**4th**	
Queries	F-Measure	RAD-WS	SA2	SA1	SA3	
	Precision	RAD-WS	SA1	SA2	SA3	
	Recall	SA2/RAD-WS		SA1	SA3	

Regarding **portability**, the benchmark seems to be quite portable. In fact, we were able to successfully benchmark four penetration testers, three static code analyzers, one anomaly detector, and one tool based on attack signatures and interface

monitoring. It is important to emphasize that these tools are provided by different entities and have very different functional characteristics. The benchmark is portable because it is not based on the implementation details of any specific tool (e.g. the workload follows the adequate standards and is generic enough to be tested by any tool).

The proposed benchmark must report similar results when used more than once over the same tool. To check **repeatability** we executed the benchmark for VS1 and SA2 (the penetration tester and the static code analyzer with the higher F-Measure) two more times. Table 7.10 presents the results of the three executions. As we can see, the results for the SA2 are always the same. This was expected as static code analyzers analyze the code in a deterministic manner, which removes variance from the results. On the other hand, some small variations can be observed for VS1. However, these variations are always under 0.01, which suggests that the benchmark is quite repeatable.

The benchmark does not require any changes to the benchmarked tools, which guarantees the **non-intrusiveness** property. This is possible because the measures portray tools effectiveness from the point-of-view of the service they provide (i.e. vulnerabilities reported) and not based on the internal behavior.

Table 7.10 – VDBenchWS-pd repeatability results.

The repetition of the benchmark execution leads to slightly different, but equivalent, results due to the non-deterministic characteristics of the workload

	VS1			SA2		
	Run 0	**Run 1**	**Run 2**	**Run 0**	**Run 1**	**Run 2**
F-Measure	0.378	0.381	0.378	0.78	0.78	0.78
Precision	0.455	0.452	0.455	0.64	0.64	0.64
Recall	0.323	0.329	0.323	1.00	1.00	1.00

The proposed benchmark is quite **simple to use** (in part, because most steps are automatic). In fact, we have been able to run it for all the tools in about 6 man-days, which correspond approximately to an average of 0.60 man-days per benchmarking experiment. Running the benchmark only requires executing the tools and comparing the reported vulnerabilities with the ones that effectively exist. As different tools report vulnerabilities in different formats (e.g. XML file, text file, GUI), to automate the vulnerability comparison step, we need to convert the output of the tools to a common format. Although possible, we decided not to do it in this work (it is just a technical issue with no scientific relevance).

7.3 Case Study #3: Using PTBenchWS-ud to Benchmark Penetration Testing Tools

Similarly to the previous case, using the PTBenchWS-ud to benchmark a set of vulnerability detection tools is basically a straightforward process that consists in following the defined steps (see Section 6.1.3). However, as this benchmark is based in a user defined workload, it is also necessary to select and characterize the workload to be used, as specified in **Phase 1: Preparation**. Also according to the preparation phase, it is necessary to select the tools under benchmarking.

As shown in Table 7.11, we used the same four **penetration testing tools** benchmarked in the previews case study, including three well-known commercial tools (introduced in Section 2.3.1, namely: HP WebInspect (HP 2008), IBM Rational AppScan (IBM 2008), and Acunetix Web Vulnerability Scanner (Acunetix 2008a). The last penetration tester considered is the improved penetration tester presented in Section 4.1.

To demonstrate the benchmarking approach we considered the same set of web services included in the VDBenchWS-pd (see Section 6.2), which include both vulnerable and non-vulnerable versions of the services. This allows comparing the results of both benchmarking campaigns. However, it is important to observe that no knowledge about the existing vulnerabilities is assumed. This way, to characterize the workload (**Phase 1. Preparation**) we used the attack signatures and interface monitoring approach, as proposed in the PTBenchWS-ud specification (see Section 6.3).

The penetration testing tools under benchmarking were run over the workload code (**Phase 2. Execution**). Again, when allowed by the testing tool, information about the domain of each web service input parameter or an exemplar invocation per operation was provided. The vulnerabilities reported were manually confirmed and compared with the ones identified by the Sign-WS tool in the preparation phase to calculate the benchmark metrics and rank the tools (**Phase 3. Comparison**).

Table 7.11 – Tools under benchmarking.
The third party penetration testing are referred to throughout the section by using the codes VS1, VS2, VS3.

Provider	Tool	Technique
HP	WebInspect	Penetration testing / Identify vulnerable inputs
IBM	Rational AppScan	
Acunetix	Web Vulnerability Scanner	
Univ. Coimbra	IPT-WS (see Section 4.1)	

7.3.1 Characterization of the workload

The vulnerabilities detected by the Sign-WS tool have been manually confirmed to guarantee the absence of false positives (as discussed in Section 7.2.2). The tool indeed reported 0 false positives, but the coverage was only of 74.05% (**117 true positives** out of 158 true vulnerabilities). Table 7.12 shows the distribution of the 117 vulnerabilities reported by the Sign-WS tool (the *Versions* column represents the number of different version of each service, as explained in Section 6.2). As we will see later, although not all the true vulnerabilities are included in the calculation of the metrics, the ones reported by Sign-WS are enough for a good estimation of the tools effectiveness.

Table 7.12 – Workload vulnerabilities as reported by Sign-WS.
For each service it is presented the number of versions existing and the total number of vulnerabilities considered for this benchmarking campaign.

Source	Service Name	Versions	Reported Inputs
TPC-App	ProductDetail	2	0
	NewProducts	2	1
	NewCustomer	6	35
	ChangePaymentMethod	2	2
TPC-C	Delivery	9	2
	NewOrder	7	6
	OrderStatus	7	13
	Payment	13	17
	StockLevel	4	4
TPC-W	AdminUpdate	2	2
	CreateNewCustomer	6	27
	CreateShoppingCart	2	0
	DoAuthorSearch	2	1
	DoSubjectSearch	2	1
	DoTitleSearch	2	1
	GetBestSellers	2	1
	GetCustomer	2	1
	GetMostRecentOrder	2	1
	GetNewProducts	2	1
	GetPassword	2	1
	GetUsername	2	0
Total		**80**	**117**

7.3.2 Benchmarking results

Table 7.13 presents the benchmark metrics for each tool, considering the characterization presented in the previous subsection (i.e. using as base set the 117 vulnerabilities reported by the Sign-WS tool). As we can see, VS1 is the tool with the highest F-Measure, closely followed by IPT-WS. VS2 presents very poor F-Measure results. Regarding precision, VS3 is the best as it reported no false positives, and IPT-WS presents the best results. Finally, in terms of recall, VS1 has the best results, while VS2 and IPT-WS performed equally. The recall of VS3 is very low as it detected only 3 vulnerabilities.

Table 7.13 – PTBenchWS-ud benchmarking results.
Overall results after executing the tools over the workload web services.

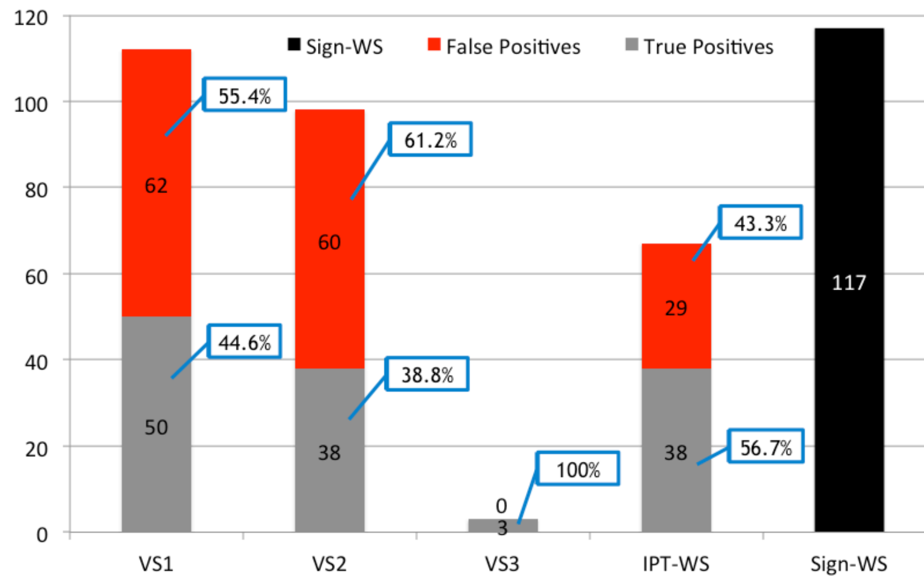
Tool	F-Measure	Precision	Recall
VS1	0.437	0.446	0.427
VS2	0.353	0.388	0.325
VS3	0.050	1.000	0.026
IPT-WS	0.413	0.567	0.325

The results presented in Table 7.13 were used to rank the tools according to the different criteria: F-Measure, Precision, and recall. Table 7.14 shows the proposed ranking. As it is possible to observe, VS1 leads the ranking in terms of both recall and F-Measure values, while VS3 has the best precision value (again, the tool reported only three vulnerabilities and none of them were false positives). Our tool, IPT-WS, ranks second in all the three metrics.

Figure 7.9 shows details on the vulnerabilities reported by the tools (the last bar in the graph presents the number of vulnerabilities detected by the Sign-WS tool). A key observation is that all the tools detected less than 43% of the vulnerabilities reported by Sign-WS, which makes the base set of vulnerabilities a good reference. Another important aspect is that VS1 reported a true vulnerability that was not reported by Sign-WS, but this was the only case. We will discuss later the impact of this in the metrics calculation, when compared to the benchmark based on a predefined workload.

Table 7.14 – PTBenchWS-ud tools ranking.
The tools ranked according to each one of the three metrics depending on the objectives of the benchmark user.

Criteria	1st	2nd	3rd	4th
F-Measure	VS1	IPT-WS	VS2	VS3
Precision	VS3	IPT-WS	VS1	VS2
Recall	VS1	VS2/IPT-WS		VS3



Tool	Coverage	False Positives Rate
VS1	42.74%	55.36%
VS2	32.48%	61.22%
VS3	2.56%	0.00%
IPT-WS	32.48%	43.28%

Figure 7.9 – PTBenchWS-ud results for the penetration testing.
The percentages presented in the image are relative to the vulnerabilities reported.

7.3.3 Comparison with the VDBenchWS-pd benchmark

A key aspect is to compare the results of the present benchmark with the ones of the benchmark based on a predefined workload. Note that, although we are considering the same set of web services, in the benchmark based on a user-defined workload we consider only a subset of the existing vulnerabilities (as reported by the Sig-WS tool). This is obviously also a way for validating the workload characterization and the metrics estimation approaches proposed to support the benchmark.

Table 7.15 summarizes the metrics obtained for both benchmarks (it is a merge of Table 7.13 and Table 7.5 for the case of the penetration testing tools). As expected, the metrics differ slightly because the base set of true vulnerabilities is different in the two cases. The F-Measure values are consistently lower in VDBenchWS-pd. This is due to the higher values for recall in PTBenchWS-ud, which are related to the lower number of true vulnerabilities considered as reference. Finally, precision is the same in both benchmarks, except for the case of VS1. This is due to the fact that VS1 detected a vulnerability that was not reported by the Sign-WS tool, and thus was not

included in the base set of true vulnerabilities. This obviously harms the reported tool precision, but as the coverage of the Sig-WS is very high, the impact is minimum. In fact, it does not affect the relative results and the tools' ranking is precisely the same for both benchmarks (see Table 7.6 and Table 7.14).

Table 7.15 – Results for both benchmarks.
Using these values it is possible to compare the results for PTBenchWS-ud with the VDBenchWS-pd.

	Tool	F-Measure	Precision	Recall
VDBenchWS-pd	VS1	0.378	0.455	0.323
	VS2	0.297	0.388	0.241
	VS3	0.037	1.000	0.019
	IPT-WS	0.338	0.567	0.241
PTBenchWS-ud	VS1	0.437	0.446	0.427
	VS2	0.353	0.388	0.325
	VS3	0.050	1.000	0.026
	IPT-WS	0.413	0.567	0.325

7.3.4 Properties Discussion

The **representativeness** of the benchmark depends on workload defined by the user. In fact, although leaving to the user the responsibility for defining the workload allows obtaining environment-specific results and prevents "*gamming*", it may also affect the validity of the results if the web services and vulnerabilities in the workload are not representative of real scenarios. Obviously, in the case of the experimental evaluation presented in the previous section, the representativeness issues are as discussed in Section 7.2.4. The ranking obtained (equal to the benchmark used in Section 7.2) suggests that the procedure and the approaches for characterizing the workload and estimating the metrics are quite adequate for characterizing the tools under assessment even when there is no previous knowledge about the existing vulnerabilities.

Regarding **portability**, the benchmark seems to be quite portable in the specified domain. In fact, we were able to benchmark four penetration testers, from different vendors and having diverse functional characteristics. However, it is important to understand that the portability is tightly related to the services defined by the user as workload. For example, if the user opts by using services that, differently from the ones used in this experiment, do not follow standard protocols and do not present a standard interface that every testing tool is able to understand and test, then it may limit the portability of the benchmark.

In terms of **repeatability** we executed the benchmark for VS1 (penetration tester with the highest F-Measure) two more times. Small variations where observed, but

they were always under 0.01, which suggests that the benchmark is quite repeatable. In fact, the repeatability results are similar to the ones discussed in Section 7.2.4.

The **non-intrusiveness** property is guaranteed because the benchmark does not require any changes to the benchmarked tools.

Although the proposed benchmark is quite **simple to use** (most steps are automatic), the fact that the user has to provide the workload and characterize the existing vulnerabilities, may increase its complexity. Obviously, the approach proposed for the metrics estimation based on the Sign-WS approach makes the work easier. Nevertheless, by only detecting injection vulnerabilities, the use of Sign-WS limits the use of the benchmark to services with this type of vulnerabilities (manual work is required for the characterization of workloads having other types of vulnerabilities).

Another important aspect is that we have been able to run the benchmark for all the tools in less than 4 man-days, which corresponds to about 1.5 man-days to select the workload and characterize it using Sign-WS plus an average of 0.6 man-days per benchmarking experiment. Obviously, in case the user applies another solution for characterizing the services (e.g. code review), the time required to run the benchmark may be higher. Anyway, after having the workload characterized, running the benchmark only requires executing the tools and comparing the reported vulnerabilities with the ones reported by the Sign-WS tool. The problem of different formats in the reports (e.g. XML file, text file, GUI) also applies, and automating the vulnerability comparison step would require converting the output of the tools to a common format.

7.4 Case Study #4: Detecting Vulnerabilities in a Service-Based Infrastructure

A simplified service based infrastructure was developed to demonstrate the usability of the SOA-Scanner (the integrated tool for detecting injection vulnerabilities in service-based infrastructures). This infrastructure uses a subset of the jSeduite SOA (Delerce-Mauris et al. 2009). In practice, the infrastructure implements one of the orchestrations of the jSeduite SOA that has been selected due to its proneness to Injection vulnerabilities and the possibility of having services with different levels of access. Some modifications were implemented (e.g. BPEL orchestrations were replaced by direct Service-to-Service invocations) to allow demonstrating all the different scenarios and functionalities in a simple infrastructure.

Figure 7.10 depicts the architecture of the system. As it is possible to observe, the services can be divided in three different groups:

1. **JSeduite** – contains services that implement the main orchestrations of jSeduite. These services are considered under control;

2. **JSeduite-WS** – contains the services used by those orchestrations and that use as data source database management systems. These are considered as partially under control services;
3. **JSeduite-WS-XML** – contains the services used by the orchestrations and use as data source XML-based resources. These are services within-reach.

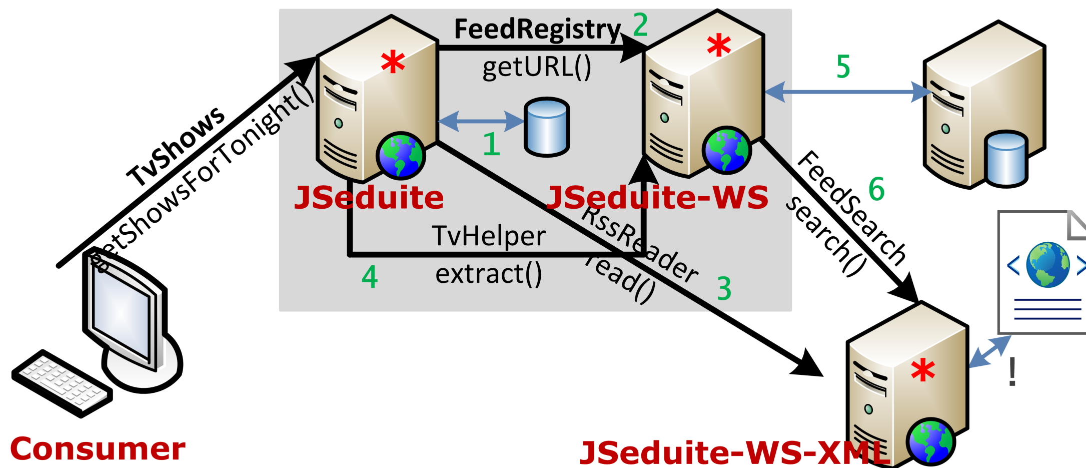


Figure 7.10 – Architecture for the Case study #4.
The gray area represents the area under control. The numbers represent the order in which the interactions are discovered. The * *represents the existence of injection vulnerabilities.*

The consumer has available the service *TvShows* from JSeduite, which uses services from both of the JSeduite-WS and JSeduite-WS-XML. The code of services *TvShows*, *FeedRegistry* and *FeedSearch* contains injection vulnerabilities. Details and code are available at (Antunes 2013).

SOA-Scanner is initially configured to test the service *TvShows*, classified by the user as Under Control, making the tool to deploy probes to the service. The remaining services are automatically discovered and tested. The SOA-Scanner was able to detect automatically the existing services, resources and relations. During the profiling phase, the interaction with a DBMS (**1**) is detected by a probe in charge of monitoring JDBC traffic, while the interactions with *FeedRegistry* (**2**), *RssReader* (**3**) and *TvHelper* (**4**) services are detected by the probe in charge of monitoring SOAP traffic. During this process, the tool uses a GUI similar to the one presented in Figure 7.11 to request to the user complementary information about the access level to the service (and, consequently, the testing scenario applicable). The user classifies the services according to the information presented in Figure 7.10. Probes are then deployed to the *FeedRegistry* and *TvHelper* (partially under control). As *RssReader* is only within-reach, no probes are deployed. When the user finishes inserting this

information, the tool starts profiling the discovered services, discovering an external DBMS (**5**) and the service *FeedSearch* (**6**), which the user classifies as within-reach (again, no probes deployed). The interaction represented by (**!**) is not detected because the services it uses has no deployed probes.

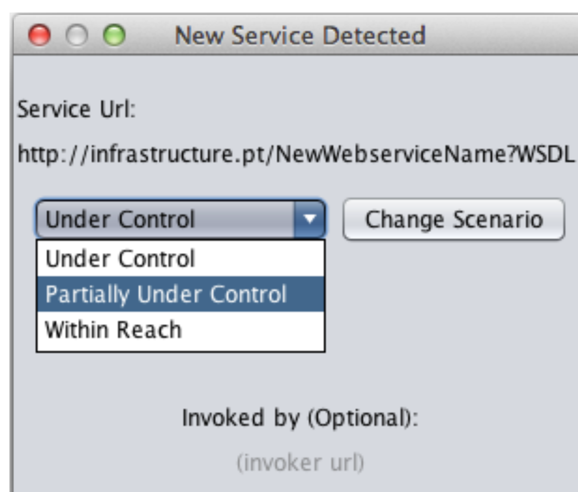


Figure 7.11 – GUI for classifying newly found services.
The service can be classified as belonging to one of the three testing scenarios considered. The service may be invoked by other service or may be considered an entry point of the infrastructure.

Table 7.16 presents the results obtained. The column "**Rev**." shows the number of vulnerabilities reported by a team of security specialists during a formal code inspection , with "**S**" and "**X**" representing the type of vulnerability (respectively SQL and XPath Injection). As in Section 6.2, the review team consisted of 3 external developers with two or more years of experience in security of database centric applications. These results were used as baseline to assess results reported by the SOA-Scanner.

Regarding vulnerability detection, for the services under control and partially under control, the SOA-Scanner was able to detect and report **all known** vulnerabilities, while **avoiding** false positives (due to the capabilities of the Sign-WS and RAD-WS to eliminate false positives, as demonstrated in the previous case studies).

For the service within-reach, the existing XPath Injection vulnerability was reported. However, the tool also reported a SQL Injection vulnerability that in fact does not exist. This **false positive** is reported because XPath and SQL injection vulnerabilities many times present very similar behaviors, and because the IPT-WS tool has no knowledge about the internals of the application. If it was possible to use one of the other detection techniques (Sign-WS or RAD-WS) to test this service, the false positive would be avoided, as the extra information provided would allow us to

precisely confirm the existence of the vulnerability. This shows the advantage of, when possible, using the SOA-Scanner tool in the mode that can take more advantage of the visibility available.

Table 7.16 – Results for SOA-Scanner vulnerability detection.
For each service and respective inputs, the table lists the vulnerabilities identified by the reviewers, the technique used, the vulnerabilities correctly reported and the false positives reported.

Service	Input	Rev.	Technique	V.	F.P.
TvShows	getShowsForTonight.provider	1S	RAD-WS	1S	
	getShowsForTonight.period		RAD-WS		
FeedRegistry	getURL.provider	1S	Sign-WS	1S	
TvHelper	extract.complexTitle		Sign-WS		
FeedSearch	search.token	1X	PT	1X	1S
RssReader	read.url		PT		

Although the infrastructure used in the case study is quite simple, we believe that the tool is able to achieve similar results in bigger and more complex infrastructures. In fact, the complexity of the discovery, profiling and a testing process grows linearly with the size of the infrastructure. This way, an increase in the number of the services in the infrastructure and in their complexity, increases proportionally the length of the process. Adding more relations between the same set of services has no impact on the performance of the tool.

7.5 Conclusion

This chapter presented four case studies that illustrate the practical application and experimental evaluation of the techniques and tools proposed in this thesis.

The first case study used well known web security scanners to detect vulnerabilities in publicly available web services. During the experiments, it was possible to observe a large number of vulnerabilities, confirming that many services are deployed without proper security testing. It was also possible to observe that existing web security scanners present very low effectiveness, showing **low detection coverage** in some cases (this suggests that many vulnerabilities probably remain undetected), while reporting a **high number of false positives** (which reduces the confidence on the precision of the vulnerabilities detected). This indicates that it is a very **difficult task to select a security scanner for web services**, also because different scanners detect different types of vulnerabilities. A final observation is that **injection vulnerabilities are prevalent in the web services tested**, as they represent approximately 90% of the vulnerabilities detected,

particularly due to SQL Injection vulnerabilities, which represent more than 84% of all vulnerabilities.

The second case study used the VDBenchWS-pd benchmark to evaluate and rank a large set of vulnerability detection tools. The results show that the proposed **benchmark can be easily used to assess and compare a wide range of tools**. In fact, the benchmark measures provided an easy way to rank the tools under benchmarking according to different user criteria. During these experiments it was also possible to evaluate the detection techniques proposed. The **IPT-WS tool ranked second among penetration testers** according to the three criteria, being able to present high recall value (only one commercial security scanner detected more vulnerabilities), while presenting an high precision value (only one other security scanner presented an higher value). The **Sign-WS tool was able to outperform penetration testing tools**, achieving much higher recall values with maximum precision. Finally, the **RAD-WS tool presented the highest F-Measure of all tools**, presenting a recall value only lower than one static code analyzer, while achieving maximum precision.

The third case study used the PTBenchWS-ud benchmark to evaluate four penetration testers. The experiments allowed **validating this benchmark as an alternative to the previous VDBenchWS-pd**, being able to overcome the limitation related to the possibility of *"gaming"* faced by VDBenchWS-pd. The benchmark results were compared with the ones from the VDBenchWS-pd benchmark and similar rankings were obtained for both cases, showing that that the procedure and the approaches for characterizing the user-defined workload and estimating the metrics are effective.

A key aspect is that benchmarking properties were discussed in detail for the two benchmarks. The results and discussion show that **the proposed benchmarking approach can be applied in the field** to specify benchmarks for vulnerability detection tools targeting different domains.

The final case study demonstrated the use of the SOA-Scanner. Although this is a very simple scenario, it as the elements necessary to **validate the capabilities of the approach to monitor and discover** the services of the infrastructure and to **use different testing techniques** to detect injection vulnerabilities according to the level of access and information available. In fact, results show that the tool was able to discover all the services in the infrastructure and the combination of testing tools allowed detecting all the existing vulnerabilities with minimum false positives reported.

Chapter 8
Conclusion and Future Work

This thesis proposes methodologies to detect software vulnerabilities in service-based infrastructures. We present a framework that defines the assumptions, the concepts, and the generic approaches that allow the development of innovative techniques and tools. The framework encompasses a reference service-based infrastructure, a generic approach for designing vulnerability detection tools for web services, which includes the definition of the testing procedure and of the tool components, and an integrated approach based on continuous monitoring to automatically discover and test the existing services, resources and interactions.

Three new techniques to detect vulnerabilities in web services implementing the generic design approach, were proposed: 1) an improved penetration testing technique to detect SQL Injection vulnerabilities using representative workloads, effective attackloads, and applies well-defined rules to improve detection coverage while reducing false positives; 2) a technique that uses attack signatures and interface monitoring to detect injection vulnerabilities, overcoming the visibility limitations of penetration testing; and 3) a runtime anomaly detection approach able to detect SQL Injection and XPath Injection vulnerabilities.

We presented also SOA-Scanner, a tool that implements the integrated approach to detect injection vulnerabilities in service-based infrastructures and that relies on the three vulnerability detection techniques mentioned above to test the services depending on the testing scenario applicable.

Comparing to previous works, such proposals innovate in the following ways. In first place, they are targeted to cope with the specificities of service-based infrastructures. For example, a continuous and dynamic discovery and testing process is used to test the infrastructure and multiple testing techniques are applied to maximize the effectiveness in the context of services with different levels of access. Finally, the proposed approaches put a strong emphasis on extensibility and modularity allowing to easily define new testing tools and also improving existing

tools by upgrading the existing the modules (e.g. if a more efficient workload emulator is developed, it is very simple to replace the old one by the new one in a tool that uses it).

This thesis also presented a generic approach for designing benchmarks for vulnerability detection tools for services, which specifies the requirements for the benchmark components and the steps needed to implement concrete benchmarks. It has been used to define two concrete benchmarks: 1) VDBenchWS-pd, a benchmark based on a predefined workload targeting tools able to detect SQL Injection vulnerabilities in web services, and 2) PTBenchWS-ud, a benchmark based on a user-provided targeting penetration testing tools for the detection of injection vulnerabilities in web services. The second benchmark overcomes the "*gaming*" problem faced by the first, by allowing the benchmark user to specify the workload.

Four case studies were devised to demonstrate and validate the proposed approaches and techniques. In the first we used several well known commercial web security scanners to detect vulnerabilities in publicly available web services. From this we drawn three main conclusions: 1) many services are deployed with security vulnerabilities; 2) it is a very difficult task to select a security scanner for web services, as different scanners report different vulnerabilities and present very low effectiveness regarding detection coverage and false positive rates; and 3) injection vulnerabilities are prevalent in the web services tested.

Two other case studies consisted of using the benchmarks proposed to conduct campaigns with the objective evaluating the proposed tools for vulnerability detection and, at the same time, validating the benchmarks. During these benchmarking campaigns it was possible to observe that VDBenchWS-pd can be easily used to assess and compare a very wide range of tools. Also, PTBenchWS-ud benchmark has shown to be an alternative to the VDBenchWS-pd to benchmark penetration testers, being able to overcome the limitation related to the possibility of "*gaming*" while ranking effectively the benchmarked tools. The benchmarking properties were discussed in detail for the two benchmarks and the results suggest that the proposed benchmarking approach can effectively be applied in the field to specify benchmarks for vulnerability detection tools targeting different domains.

Regarding the evaluation of our detection tools, it was possible to observe that the IPT-WS tool ranked second among all the penetration testers assessed according to the three criteria defined by the benchmarks. The second observation is that Sign-WS tool was able to outperform penetration testing tools, achieving much higher recall values with maximum precision. And finally, the RAD-WS tool presented the highest F-Measure of all tools, presenting a recall value only lower than one static code analyzer, while achieving maximum precision.

In the last case study we demonstrated the use of the SOA-Scanner. Although a very simple scenario, it contains services with different levels of access, it has different types of vulnerabilities and it uses different types of resources, thus providing the

elements necessary to demonstrate all the capabilities of the tool. During the experiments it was possible to observe that the tool was able to discover all the services in the infrastructure and the combination of the testing tools allowed detecting all the existing vulnerabilities.

Future work

Several research topics are currently in progress as a continuation of the work presented in this thesis.

1. **Implement the SOA-Scanner as a product that can easily be used by developing teams:** the tool is currently in a prototype status. Concluding the implementation of the tool in such way that it would be easy to use by any person would help widespreading its utilization. We believe that the tool will be a key contribution towards improving the security of the service-based infrastructures deployed. Additionally, we are also considering adding new features to the tool, some of which are described in the following points.
2. **Extend the techniques for other types of services:** although some of the proposed techniques target primarily SOAP web services, most of the concepts can be transposed to other technologies (e.g. RESTful web services). However, there is an important part of the work, currently in progress, that is to understand the differences between technologies and the impact these differences may have in the tools. We are currently conducting an field study to get knowledge about the characteristics of RESTful web services in the wild.
3. **Extend the techniques to other types of vulnerabilities:** although injection vulnerabilities rank at the top of the most dangerous vulnerabilities, there are others. A key part of this work will be to gather web services with these vulnerabilities to be used as case study to evaluate the researched techniques.
4. **Extend the techniques to detect second order vulnerabilities:** second-order injection happens when the malicious code is injected successfully but not executed immediately. Instead it is stored by the application in some resource to be retrieved and executed eventually, when that resource is accessed (W. G. Halfond, Viegas, and Orso 2006). The SOA-Scanner tool has the characteristics necessary to detect this kind of vulnerabilities: it monitors the interfaces between the web services and external resources or services. With the appropriate modifications, we intend to make the tool to attack these interfaces and detect this kind of vulnerabilities.
5. **Runtime Verification and Validation (V&V) of service-based infrastructures**: the traditional lifecycle in V&V assumes a structured and highly documented software or system development process that allows

gathering the required quality evidences, and presumes that the system does not evolve after deployment (i.e. the structure is stable over time). This represents a serious problem, as there are no V&V methods, tools and processes that can cope with the dynamic nature of service based infrastructures, as well as with many other prominent features of these systems. Complying with nowadays organizations' requirements demands for deployment and maintenance of trustworthy dynamic service-based software systems, which naturally results in the superposition of the design and runtime phases, thus imposing the need for a V&V paradigm shift. To overcome this problem new V&V approaches that can be applied ate runtime are necessary. Similarly to the SOA-Scanner, runtime V&V should take advantage of monitoring services and infrastructures, which will support the runtime assessment of the system through the collection of measurements for quantitative analysis of security and trustworthiness.

6. **Using vulnerability injection to develop benchmarks for vulnerability detection tools:** in the same way fault injection has become an attractive approach to validate specific fault handling and fault detection mechanisms, vulnerability injection is a powerful tool that can be used to evaluate the effectiveness of vulnerability prevention and detection tools and methodologies. By using a realistic vulnerability injection technique it will be very easy to create new workloads based on any set of web services, as the vulnerabilities injected would be known. Obviously, the main challenge is related to how realistic the injection vulnerabilities can be.

The work presented in this thesis has largely contributed to gain a broad experience on services security testing. In addition, this work provided us an excellent standard environment for the evaluation and comparison of alternative vulnerability detection tools based on their effectiveness. This way, several topics can be foreseen as a continuation of the present work:

1. **Use collaborative testing for the detection of vulnerabilities in services:** using this kind of collaboration it is possible to increase the level of access to some of some services of the infrastructure, depending on the existing collaboration agreements. Obviously, besides the negotiation of the collaboration agreement, it is necessary to configure the tools in such a decentralized way that all the parts of the collaboration would have an instance running that could transmit information to the others.
2. **Use static code analysis to improve vulnerability detection:** as demonstrated by the effectiveness of RAD-WS combining different tools can be a very profitable option. Although there are dynamic analysis techniques that combine static code analysis with the execution of tests, there is room to

research automated tools to perform this task in the context of service-based infrastructures.

3. **Research vulnerability removal techniques:** more than just detecting vulnerabilities it is important to automatically remove them, which consists of modifying the source code or byte code of the application in order to fix security flaws without modifying or harming the functional behavior of the web service. This is of utmost importance as, usually, developers are not specialized in security aspects and the vulnerability patterns are repeated several times, even in different applications.
4. **Research attack detection approaches:** attack detection is an alternative to mitigate vulnerabilities when vulnerability removal is not possible. It consists of introduce capabilities that, at runtime, detect and stop attacks (this is the reasoning behind intrusion detection systems, runtime anomaly detection systems, etc.). The idea here is to propose tools that can be integrated into the development environment to automatically detect and stop attacks.

References

Acunetix. 2007. “70% of Websites at Immediate Risk of Being Hacked!” February 12. http://www.acunetix.com/news/security-audit-results.htm.

———. 2008a. “Acunetix Web Vulnerability Scanner.” http://www.acunetix.com/vulnerability-scanner/.

———. 2008b. “AcuSensor Technology.” http://www.acunetix.com/vulnerability-scanner/acusensor.htm.

Antunes, N. 2013. “Tools and Techniques for Detecting Vulnerabilities in Service-Based Infrastructures.” September 10. http://eden.dei.uc.pt/~nmsa/thesis-materials.

Antunes, N., N. Laranjeiro, M. Vieira, and H. Madeira. 2009a. “Effective Detection of SQL/XPath Injection Vulnerabilities in Web Services.” In *2009 IEEE International Conference on Services Computing (SCC 2009)*, 260–267. Bangalore, India. doi:10.1109/SCC.2009.23.

———. 2009b. “Effective Detection of SQL/XPath Injection Vulnerabilities in Web Services.” In , 260–267. Bangalore, India. doi:10.1109/SCC.2009.23.

Antunes, N., and M. Vieira. 2009a. “Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services.” In *15th IEEE Pacific Rim International Symposium on Dependable Computing*, 301–306. Shanghai, China: IEEE Computer Society. doi:10.1109/PRDC.2009.54.

———. 2009b. “Detecting SQL Injection Vulnerabilities in Web Services.” In *Fourth Latin-American Symposium on Dependable Computing 2009 (LADC ’09)*, 17–24. Joao Pessoa, Brazil: IEEE Computer Society. doi:10.1109/LADC.2009.21.

———. 2010. “Benchmarking Vulnerability Detection Tools for Web Services.” In *IEEE Eighth International Conference on Web Services (ICWS 2010)*, 203–210. Miami, Florida, USA. doi:10.1109/ICWS.2010.76.

———. 2011. “Enhancing Penetration Testing with Attack Signatures and Interface Monitoring for the Detection of Injection Vulnerabilities in Web Services.” In *2011 IEEE International Conference on Services Computing (SCC)*, 104–111. IEEE. doi:10.1109/SCC.2011.67.

———. 2012a. “Defending Against Web Application Vulnerabilities.” *Computer* 45 (2): 66–72. doi:10.1109/MC.2011.259.

———. 2012b. “Evaluating and Improving Penetration Testing in Web Services.” In Dallas TX, USA.

———. 2013. “SOA-Scanner: An Integrated Tool to Detect Vulnerabilities in Service-Based Infrastructures.” In *10th International Conference on Services Computing (SCC 2013)*. Santa Clara, CA, USA.

Arkin, B., S. Stender, and G. McGraw. 2005. “Software Penetration Testing.” *IEEE Security & Privacy* 3 (1): 84–87.

Atlassian. 2010. “Clover - Code Coverage for Java.” http://www.atlassian.com/software/clover/.

Ayewah, N., D. Hovemeyer, J. D Morgenthaler, J. Penix, and W. Pugh. 2008. “Using Static Analysis to Find Bugs.” *IEEE Software* 25 (5): 22–29.

Balzarotti, D., M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. 2008. “Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications.” In *IEEE Symposium on Security and Privacy*. Vol. 66.

Baresi, Luciano, Carlo Ghezzi, and Sam Guinea. 2004. “Smart Monitors for Composed Services.” In *Proceedings of the 2nd International Conference on Service Oriented Computing*, 193–202. ICSOC ’04. New York, NY, USA: ACM. doi:10.1145/1035167.1035195. http://doi.acm.org/10.1145/1035167.1035195.

Bau, J., E. Bursztein, D. Gupta, and J. Mitchell. 2010. “State of the Art: Automated Black-box Web Application Vulnerability Testing.” In *2010 IEEE Symposium on Security and Privacy (SP)*, 332–345.

Beck, K. 2003. *Test-driven Development: By Example*. Addison-Wesley Professional.

Bennett, K., P. Layzell, D. Budgen, P. Brereton, L. Macaulay, and M. Munro. 2000. “Service-based Software: The Future for Flexible Software.” In *Proceedings Seventh Asia-Pacific Software Engineering Conference (APSEC 2000)*, 214–221. Singapore. doi:10.1109/APSEC.2000.896702.

Bertolino, Antonia, Guglielmo De Angelis, Lars Frantzen, and Andrea Polini. 2009. “The PLASTIC Framework and Tools for Testing Service-Oriented Applications.” In *Software Engineering*, edited by Andrea De Lucia and Filomena Ferrucci, 106–139. Lecture Notes in Computer Science 5413. Springer Berlin Heidelberg. http://link.springer.com/chapter/10.1007/978-3-540-95888-8_5.

Bertolino, Antonia, Lars Frantzen, Andrea Polini, and Jan Tretmans. 2006. "Audition of Web Services for Testing Conformance to Open Specified Protocols." In *Architecting Systems with Trustworthy Components*, edited by Ralf H. Reussner, Judith A. Stafford, and Clemens A. Szyperski, 1–25. Lecture Notes in Computer Science 3938. Springer Berlin Heidelberg. http://link.springer.com/chapter/10.1007/11786160_1.

Cachin, C., J. Camenisch, Y. Deswarte, J. Dobson, D. Horne, K. Kursawe, J. C. Laprie, et al. 2000. "MAFTIA: Reference Model and Use Cases." MAFTIA Deliverable. https://docs.di.fc.ul.pt/jspui/handle/10455/2927.

Calinescu, Radu. 2011. "When the Requirements for Adaptation and High Integrity Meet." In *Proceedings of the 8th Workshop on Assurances for Self-adaptive Systems*, 1–4. ASAS '11. New York, NY, USA: ACM. doi:10.1145/2024436.2024438. http://doi.acm.org/10.1145/2024436.2024438.

Campwood Software. 2008. "SourceMonitor Version 2.5." http://www.campwoodsw.com/sourcemonitor.html.

Carreira, J., H. Madeira, and J. G Silva. 1998. "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers." *IEEE Transactions on Software Engineering* 24 (2): 125–136.

Ceccarelli, A., M. Vieira, and A. Bondavalli. 2011a. "A Service Discovery Approach for Testing Dynamic SOAs." In *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*, 133 –142. doi:10.1109/ISORCW.2011.23.

———. 2011b. "A Testing Service for Lifelong Validation of Dynamic SOA." In *2011 IEEE 13th International Symposium on High-Assurance Systems Engineering (HASE)*, 1 –8. doi:10.1109/HASE.2011.18.

"Center for Internet Security." 2012. Accessed September 23. http://www.cisecurity.org/.

Chappell, D. A, and T. Jewell. 2002a. *Java Web Services*. Sebastopol, CA, USA: O'Reilly & Associates, Inc.

———. 2002b. *Java Web Services*. Sebastopol, CA, USA: O'Reilly & Associates, Inc.

Chappell, David A. 2009. *Enterprise Service Bus*. O'Reilly Media, Inc.

Christensen, E., F. Curbera, G. Meredith, and S. Weerawarana. 2001. "Web Service Definition Language (WSDL) 1.1." *World Wide Web Consortium (W3C)*. March 15. http://www.w3.org/TR/wsdl.

Christey, S., and R. A Martin. 2006. "Vulnerability Type Distributions in CVE." *V1. 0* 10: 04.

Christey, S., and R. A. Martin. 2007. "Vulnerability Type Distributions in CVE". V1.1. The MITRE Corporation.

Commission of the European Communities. 1993. *The IT Security Evaluation Manual (ITSEM)*.

Curbera, F., M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. 2002. “Unraveling the Web Services Web: An Introduction to SOAP, WSDL, andUDDI.” *IEEE Internet Computing* 6 (2): 86–93.

Curphey, M., D. Endler, W. Hau, S. Taylor, T. Smith, A. Russell, G. McKenna, R. Parke, K. McLaughlin, and N. Tranter. 2002. “A Guide to Building Secure Web Applications.” *The Open Web Application Security Project* 1.

De Barros, Marcelo, Jing Shiau, Chen Shang, Kenton Gidewall, Hui Shi, and Joe Forsmann. 2007. “Web Services Wind Tunnel: On Performance Testing Large-scale Stateful Web Services.” In *Dependable Systems and Networks, 2007. DSN’07. 37th Annual IEEE/IFIP International Conference On*, 612–617. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4273012.

Delerce-Mauris, Clémentine, Lionel Palacin, Stéphane Martarello, Sébastien Mosser, and Mireille Blay-Fornarino. 2009. “Plateforme SEDUITE : Une Approche SOA de La Diffusion d’Informations”. Research Report. Sophia Antipolis, France: University of Nice, I3S CNRS. http://www.i3s.unice.fr/%7Emh/RR/2009/RR-09.01-S.MOSSER.pdf.

Dobson, John E., and Brian Randell. 1986. “Building Reliable Secure Computing Systems Out of Unreliable Insecure Components.” In *Proceedings of the Conference on Security and Privacy*, 187–193. Oakland, USA.

Doliner, Mark. 2006. “Cobertura.” http://cobertura.sourceforge.net/.

Doupé, A., M. Cova, and G. Vigna. 2010. “Why Johnny Can’t Pentest: An Analysis of Black-Box Web Vulnerability Scanners.” *Detection of Intrusions and Malware, and Vulnerability Assessment*: 111–131.

Echtle, K., and M. Leu. 1992. “The EFA Fault Injector for Fault-tolerant Distributed System Testing.” In *Workshop on Fault-Tolerant Parallel and Distributed Systems*. Vol. 226. Amherst, MA: IEEE Computer Society Press.

Edvardsson, Jon. 1999. “A Survey on Automatic Test Data Generation.” In *Proceedings of the 2nd Conference on Computer Science and Engineering*, 21–28. http://staff.unak.is/not/andy/MScTestingMaintenance0607/Lectures/SurveyAutomaticTestDataGeneration.pdf.

Erl, Thomas. 2005. *Service-oriented Architecture : Concepts, Technology, and Design*. Upper Saddle River, NJ: Prentice Hall Professional Technical Reference.

eviware. 2008. “soapUI.” http://www.soapui.org/.

Fagan, M. E. 1976. “Design and Code Inspections to Reduce Errors in Program Development.” *IBM Systems Journal* 15 (3): 182–211.

Fogie, Seth, Jeremiah Grossman, Robert Hansen, Anton Rager, and Petko D. Petkov. 2007. *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress Publishing.

Fonseca, J., M. Vieira, and H. Madeira. 2007. “Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks.” In *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*, 365–372. Melbourne, Australia. doi:10.1109/PRDC.2007.55.

———. 2009. “Vulnerability & Attack Injection for Web Applications.” In *IEEE/IFIP International Conference on Dependable Systems & Networks, 2009. DSN '09*, 93–102.

Fortify Software. 2008. “Fortify 360 Software Security Assurance.” http://www.fortify.com/products/fortify-360/.

Foundstone, Inc. 2005. “Foundstone WSDigger.” *Foundstone Free Tools*. http://www.foundstone.com/us/resources/proddesc/wsdigger.htm.

Freedman, Daniel P., and Gerald M. Weinberg. 2000. *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products*. Dorset House Publishing Co., Inc. http://portal.acm.org/citation.cfm?id=556043#.

Fuentes, F., and D. C Kar. 2005. “Ethereal Vs. Tcpdump: a Comparative Study on Packet Sniffing Tools for Educational Purpose.” *Journal of Computing Sciences in Colleges* 20 (4): 169–176.

Gao, J., E.Y. Zhu, S. Shim, and Lee Chang. 2000. “Monitoring Software Components and Component-based Software.” In *Computer Software and Applications Conference, 2000. COMPSAC 2000. The 24th Annual International*, 403–412. doi:10.1109/CMPSAC.2000.884757.

Ghezzi, C., M. Jazayeri, and D. Mandrioli. 2002. *Fundamentals of Software Engineering*. Prentice Hall PTR Upper Saddle River, NJ, USA.

Gosling, J., B. Joy, G. Steele, and G. Bracha. 2005. *Java (TM) Language Specification, The (Java (Addison-Wesley))*. Addison-Wesley Professional. http://dl.acm.org/citation.cfm?id=1036643.

Gray, Jim. 1992. *Benchmark Handbook: For Database and Transaction Processing Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

———. 1993. *The Benchmark Handbook*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Grosclaude, Irene. 2004. “Model-Based Monitoring of Component-Based Software Systems.” http://citeseer.uark.edu:8080/citeseerx/viewdoc/summary?doi=10.1.1.1.7593.

Halfond, W. G., J. Viegas, and A. Orso. 2006. "A Classification of SQL-injection Attacks and Countermeasures." In *International Symposium on Secure Software Engineering*.

Halfond, W.G.J., and A. Orso. 2005. "AMNESIA: Analysis and Monitoring for NEutralizing SQL-injection Attacks." In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, 183.

Howard, Michael, and David E. Leblanc. 2002. *Writing Secure Code*. 2nd ed. Redmond, Washington: Microsoft Press.

HP. 2008. "HP WebInspect." https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-201-200%5E9570_4000_100__.

Huang, Y. W, F. Yu, C. Hang, C. H Tsai, D. T Lee, and S. Y Kuo. 2004. "Securing Web Application Code by Static Analysis and Runtime Protection." In *Proceedings of the 13th International Conference on World Wide Web*, 40–52.

Huang, Yao-Wen, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. 2003. "Web Application Security Assessment by Fault Injection and Behavior Monitoring." In *Proceedings of the 12th International Conference on World Wide Web*, 148–159. Budapest, Hungary: ACM.

Hunter, Jason. 2002. "JDOM Makes XML Easy." In *Sun's 2002 Worldwide Java Developer Conference*.

IBM. 2008. "IBM Rational AppScan." http://www-01.ibm.com/software/awdtools/appscan/.

Infrastructure, P. K., and T. P. Profile. 2002. "Common Criteria for Information Technology Security Evaluation". National Security Agency. http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA406677.

Jensen, Meiko, Nils Gruschka, Ralph Herkenhoner, and Norbert Luttenberger. 2007. "SOA and Web Services: New Technologies, New Standards - New Attacks." In *Fifth European Conference on Web Services, 2007. ECOWS '07.*, 35–44.

JetBrains. 2009. "IntelliJ IDEA." http://www.jetbrains.com/idea/free_java_ide.html.

Jovanovic, Nenad, Christopher Kruegel, and Engin Kirda. 2006. "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper)." In *IEEE Symposium on Security and Privacy*, 258–263. Berkeley/Oakland, California: IEEE Computer Society.

Kals, S., E. Kirda, C. Kruegel, and N. Jovanovic. 2006. "Secubat: a Web Vulnerability Scanner." In *Proceedings of the 15th International Conference on World Wide Web*, 256.

Kaner, Cem. 1996. "Software Negligence and Testing Coverage." In *Proceedings of STAR 96: The Fifth International Conference on Software Testing Analysis and Review*, 299–327. Orlando, FL, USA. http://stalatest.googlecode.com/svn/trunk/Literatur/negligence_and_testing_coverage.pdf.

Kanoun, Karama, and Lisa Spainhower. 2008. *Dependability Benchmarking for Computer Systems*. Wiley-IEEE Computer Society Pr.

Keen, Martin, Amit Acharya, Susan Bishop, Alan Hopkins, Sven Milinski, Chris Nott, Rick Robinson, Jonathan Adams, and Paul Verschueren. 2004. *Patterns: Implementing an SOA Using an Enterprise Service Bus*. IBM International Technical Support Organization. http://serviceorientedarchitecturesoa.net/goto/http://oss.org.cn/ossdocs/soa/ibm/sg246346.pdf.

Kiczales, G. J, J. O Lamping, C. V Lopes, J. J Hugunin, E. A Hilsdale, and C. Boyapati. 2002. "Aspect-oriented Programming."

Koopman, P., and J. DeVale. 2000. "The Exception Handling Effectiveness of POSIX Operating Systems." *IEEE Transactions on Software Engineering* 26 (9) (September): 837–848. doi:10.1109/32.877845.

Kopecky, J., K. Gomadam, and T. Vitvar. 2008. "hRESTS: An HTML Microformat for Describing RESTful Web Services." In *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, 2008. WI-IAT '08*, 1:619–625. doi:10.1109/WIIAT.2008.379. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4740521.

Kosuga, Y., K. Kono, M. Hanaoka, M. Hishiyama, and Y. Takahama. 2007. "Sania: Syntactic and Semantic Analysis for Automated Testing Against SQL Injection." In *23rd Annual Computer Security Applications Conference. IEEE Computer Society*, 107–117.

Kotamraju, Jitendra. 2007. "JSR 224: JavaTM API for XML-Based Web Services (JAX-WS) 2.0". Java Specification Request 224. https://jax-ws.java.net/.

Kruegel, C., and G. Vigna. 2003. "Anomaly Detection of Web-based Attacks." In , 251–261.

Laranjeiro, N., M. Vieira, and H. Madeira. 2008. "Experimental Robustness Evaluation of JMS Middleware." In *IEEE International Conference on Services Computing, 2008. SCC '08,* 1:119–126. Honolulu, HI: IEEE. doi:10.1109/SCC.2008.129.

Lindstrom, P. 2004. "Attacking and Defending Web Service." *A Spire Research Report*.

LittleShoot. 2010. "LittleProxy HTTP Proxy." http://www.littleshoot.org/littleproxy/.

Livshits, V. B, and M. S Lam. 2005. "Finding Security Vulnerabilities in Java Applications with Static Analysis." In *Proceedings of the 14th Conference on USENIX Security Symposium-Volume 14*, 18.

Lowis, L., and R. Accorsi. 2009. "On a Classification Approach for SOA Vulnerabilities." In *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, 2:439–444. doi:10.1109/COMPSAC.2009.173.

Lyu, Michael R, ed. 1996. *Handbook of Software Reliability Engineering*. Hightstown, NJ, USA: McGraw-Hill, Inc.

MacKenzie, C Matthew, Ken Laskey, Francis McCabe, Peter F Brown, Rebekah Metz, and Booz Allen Hamilton. 2006. "Reference Model for Service Oriented Architecture 1.0". OASIS Standard. http://docs.oasis-open.org/soa-rm/v1.0/.

Martin, M., B. Livshits, and M. S Lam. 2005. "Finding Application Errors and Security Flaws Using PQL: a Program Query Language." *ACM SIGPLAN Notices* 40 (10): 383.

Maxion, R.A., and K.M.C. Tan. 2000. "Benchmarking Anomaly-based Detection Systems." In *Proceedings International Conference on Dependable Systems and Networks, 2000. DSN 2000*, 623 –630. doi:10.1109/ICDSN.2000.857599.

McAllister, Sean, Engin Kirda, and Christopher Kruegel. 2008. "Leveraging User Interactions for In-Depth Testing of Web Applications." In *Recent Advances in Intrusion Detection*, 191–210.

McGraw, G. 2006. *Software Security: Building Security In*. Addison-Wesley Professional.

McGraw, Gary, and Bruce Potter. 2004. "Software Security Testing." *IEEE Security and Privacy*.

Meier, Wolfgang. 2003. "eXist: An Open Source Native XML Database." In *Web, Web-Services, and Database Systems*, edited by Akmal B. Chaudhri, Mario Jeckle, Erhard Rahm, and Rainer Unland, 169–183. Lecture Notes in Computer Science 2593. Springer Berlin Heidelberg. http://link.springer.com/chapter/10.1007/3-540-36560-5_13.

Michelsen, John, and Jason English. 2012. *Service Virtualization: Reality Is Overrated*. Apress.

Mukherjee, A., and D. P Siewiorek. 1997. "Measuring Software Dependability by Robustness Benchmarking." *IEEE Transactions on Software Engineering* 23 (6): 366–378.

Myers, Glenford J., Corey Sandler, and Tom Badgett. 2011. *The Art of Software Testing*. John Wiley & Sons. http://books.google.com/books?hl=en&lr=&id=GjyEFPkMCwcC&oi=fnd&pg=PT5&dq=The+Art+Of+Software+Testing&ots=AfxUHWk_7l&sig=Xk4hmJE4daTT5CLGc3smT_h5bVY.

Neto, A.A., and M. Vieira. 2008. "Towards Assessing the Security of DBMS Configurations." In *IEEE International Conference on Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008*, 90 –95. doi:10.1109/DSN.2008.4630074.

———. 2009. "A Trust-Based Benchmark for DBMS Configurations." In *15th IEEE Pacific Rim International Symposium on Dependable Computing, 2009. PRDC '09*, 143 –150. doi:10.1109/PRDC.2009.31.

Neto, Afonso Araújo, and Marco Vieira. 2010. "Benchmarking Untrustworthiness." *International Journal of Dependable and Trustworthy Information Systems* 1 (2): 32–54. doi:10.4018/jdtis.2010040102.

Newkirk, James W., and Alexei A. Vorontsov. 2004. *Test-Driven Development in Microsoft .Net*. Microsoft Press. http://portal.acm.org/citation.cfm?id=983793#.

NTA Monitor. 2007. "Annual Security Report." http://www.nta-monitor.com/posts/2007/05/annualsecurityreport.html.

———. 2008a. "Annual Security Report."

———. 2008b. "Annual Web Application Security Report." http://www.nta-monitor.com.

———. 2011a. "Annual Web Application Security Report."

———. 2011b. "Annual Web Application Security Report."

OWASP Foundation. 2001. "Open Web Application Security Project." *OWASP Foundation*. http://www.owasp.org/.

———. 2008. "OWASP WSFuzzer Project." http://www.owasp.org/index.php/Category:OWASP_WSFuzzer_Project.

———. 2010. "OWASP Application Security FAQ Version 3." http://www.owasp.org/index.php/OWASP_Application_Security_FAQ.

———. 2013. "OWASP Top 10 2013." OWASP Top 10. Open Web Application Security Project. https://www.owasp.org/index.php/Top_10_2013.

Papazoglou, Mike P., and Willem-Jan van den Heuvel. 2007. "Service Oriented Architectures: Approaches, Technologies and Research Issues." *The VLDB Journal* 16 (3) (July 1): 389–415. doi:10.1007/s00778-007-0044-3.

Pautasso, Cesare, Olaf Zimmermann, and Frank Leymann. 2008. "Restful Web Services Vs. Big'web Services: Making the Right Architectural Decision." In *Proceedings of the 17th International Conference on World Wide Web*, 805–814. http://dl.acm.org/citation.cfm?id=1367606.

Perrey, R., and M. Lycett. 2003. "Service-oriented Architecture." In *Proceedings 2003 Symposium on Applications and the Internet Workshops (SAINT 2003 Workshps)*, 116–119. Orlando, FL, USA. doi:10.1109/SAINTW.2003.1210138.

Pistore, M., F. Barbon, P. Bertoli, D. Shaparau, and P. Traverso. 2004. "Planning and Monitoring Web Service Composition." In , 106–115.

Qiu, L., Y. Zhang, F. Wang, M. Kyung, and H. R. Mahajan. 1985. "Trusted Computer System Evaluation Criteria." In *National Computer Security Center*. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.118.9902.

Ramakrishnan, Raghu, and Johannes Gehrke. 2003. *Database Management Systems*. Boston: McGraw-Hill.

Reese, G., and A. Oram. 2000. *Database Programming with JDBC and JAVA*. O'Reilly & Associates, Inc. Sebastopol, CA, USA.

Richardson, L., and S. Ruby. 2007. *RESTful Web Services*. O'Reilly Media, Inc.

Sabhnani, M., and G. Serpen. 2004. "Why Machine Learning Algorithms Fail in Misuse Detection on KDD Intrusion Detection Data Set." *Intelligent Data Analysis* 8 (4).

Sandia National Laboratories. 2012. "Information Operations Red Team and Assessments™." Accessed September 23. http://www.sandia.gov/iorta/.

Sandoval, José, Atanas Roussev, and Richard Wallace. 2009. *RESTful Java Web Services*. Birmingham, UK: Packt Pub. http://site.ebrary.com/id/10430464.

Santiago, Valdivino, Ana Silvia Martins do Amaral, N. L. Vijaykumar, Md F. Mattiello-Francisco, Eliane Martins, and Odnei Cuesta Lopes. 2006. "A Practical Approach for Automated Test Case Generation Using Statecharts." In *Computer Software and Applications Conference, 2006. COMPSAC'06. 30th Annual International*, 2:183–188. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4020165.

Scovetta, Michael. 2008. "Yet Another Source Code Analyzer." www.yasca.org.

Sekar, R. 2009. "An Efficient Black-box Technique for Defeating Web Application Attacks." In *Proceedings of the 16th Annual Network and Distributed System Security Symposium*.

Shema, Mike. 2010. *Seven Deadliest Web Application Attacks*. Burlington, MA: Syngress.

Shin, Y., L. Williams, and T. Xie. 2006. "SQLUnitGen: SQL Injection Testing Using Static and Dynamic Analysis." In *Proceedings of the 17th IEEE International Conference on Software Reliability Engineering (ISSRE 2006)*. Raleigh, NC, USA.

Singhal, A., T. Winograd, and K. Scarfone. 2007. "Guide to Secure Web Services: Recommendations of the National Institute of Standards and Technology." *Report, National Institute of Standards and Technology, US Department of Commerce*: 800–95.

Stock, A., J. Williams, and D. Wichers. 2007. "OWASP Top 10." http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

Stuttard, D., and M. Pinto. 2007. *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws*. Wiley Publishing, Inc.

Teixeira, Emanuel, João Antunes, and Nuno Neves. 2007. "Avaliação de Ferramentas de Análise Estática de Código Para Detecção de Vulnerabilidades." In *Proceedings of the Segurança Informática Nas Organizações*. SINO.

Transaction Processing Performance Council. 2008. "TPC BenchmarkTM App (Application Server) Standard Specification, Version 1.3." February 28. http://www.tpc.org/tpc_app/.

———. 2009. "Transaction Processing Performance Council." http://www.tpc.org/.

University of Maryland. 2009. "FindBugs™ - Find Bugs in Java Programs." http://findbugs.sourceforge.net/.

Van Rijsbergen, C. J. 1979. "Information Retrieval." *Buttersworth, London*.

Vieira, M., N. Antunes, and H. Madeira. 2009. "Using Web Security Scanners to Detect Vulnerabilities in Web Services." In *IEEE/IFIP International Conference on Dependable Systems & Networks, 2009. DSN'09.*, 566–571. Estoril, Lisbon, Portugal. doi:10.1109/DSN.2009.5270294.

Vieira, M., N. Laranjeiro, and H. Madeira. 2007. "Assessing Robustness of Web-services Infrastructures." In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2007. DSN'07*, 131–136.

Vieira, M., and H. Madeira. 2005. "Towards a Security Benchmark for Database Management Systems." In *International Conference on Dependable Systems and Networks, 2005. DSN 2005.*, 592 – 601. Yokohama, Japan. doi:10.1109/DSN.2005.93.

W3C. 2004. "Web Services Glossary." http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/.

Wagner, S., J. Jürjens, C. Koller, and P. Trischberger. 2005. "Comparing Bug Finding Tools with Reviews and Tests." *Testing of Communicating Systems*: 40–55.

WASC. 2008. "Web Application Security Consortium - Classes of Attacks." http://www.webappsec.org/projects/threat/classes_of_attack.shtml.

Wassermann, Bruno, and Wolfgang Emmerich. 2011. "Monere: Monitoring of Service Compositions for Failure Diagnosis." In *Service-Oriented Computing*, edited by Gerti Kappel, Zakaria Maamar, and Hamid R. Motahari-Nezhad, 344–358. Lecture Notes in Computer Science 7084. Springer Berlin Heidelberg. http://link.springer.com/chapter/10.1007/978-3-642-25535-9_23.

Wassermann, G., C. Gould, Z. Su, and P. Devanbu. 2007. "Static Checking of Dynamically Generated Queries in Database Applications." *ACM Transactions on Software Engineering and Methodology (TOSEM)* 16 (4): 14.

Wassermann, G., and Z. Su. 2004. "An Analysis Framework for Security in Web Applications." *SAVCBS 2004 Specification and Verification of Component-Based Systems*: 70.

Xie, Y., and A. Aiken. 2006. "Static Detection of Security Vulnerabilities in Scripting Languages." In *15th USENIX Security Symposium*, 179–192.

Zubin67. 2010. "Monitoring Services in Service Oriented Architecture" May 6. http://www.slideshare.net/Zubin67/monitoring-services-in-service-oriented-architecture.