Security Benchmarking of Transactional Systems

Afonso Comba de Araújo Neto

Dissertation submitted to the University of Coimbra
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

September 2012

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. The work is within the Dependable 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).

This work was partially supported by the Programme Alβan, the European Union Programme of High Level Scholarships for Latin America, scholarship no. E07D403033BR.

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

Don't you believe in flying saucers, they ask me?

Don't you believe in telepathy? — in ancient astronauts? — in the Bermuda triangle? — in life after death?

“No”, I reply. “No, no, no, no, and again no.”

One person recently, goaded into desperation by the litany of unrelieved negation, burst out “Don't you believe in anything?”

“Yes”, I said. “I believe in evidence. I believe in observation, measurement, and reasoning, confirmed by independent observers. I'll believe anything, no matter how wild and ridiculous, if there is evidence for it. The wilder and more ridiculous something is, however, the firmer and more solid the evidence will have to be.”

—Isaac Asimov, The Roving Mind (1997), 43

~...~

Abstract

Most organizations nowadays depend on some kind of computer infrastructure to manage business critical activities. This dependence grows as computer systems become more reliable and useful, but so does the complexity and size of systems. Transactional systems, which are database-centered applications used by most organizations to support daily tasks, are no exception. A typical solution to cope with systems complexity is to delegate the software development task, and to use existing solutions independently developed and maintained (either proprietary or open source).

The multiplicity of software and component alternatives available has boosted the interest in suitable benchmarks, able to assist in the selection of the best candidate solutions, concerning several attributes. However, the huge success of performance and dependability benchmarking markedly contrasts with the small advances on security benchmarking, which has only sparsely been studied in the past.

This thesis discusses the security benchmarking problem and main characteristics, particularly comparing these with other successful benchmarking initiatives, like performance and dependability benchmarking. Based on this analysis, a general framework for security benchmarking is proposed. This framework, suitable for most types of software systems and application domains, includes two main phases: *security qualification* and *trustworthiness benchmarking*. Security qualification is a process designed to evaluate the most obvious and identifiable security aspects of the system, dividing the evaluated targets in acceptable or unacceptable, given the specific security requirements of the application domain. Trustworthiness benchmarking, on the other hand, consists of an evaluation process that is applied over the qualified targets to estimate the probability of the existence of hidden or hard to detect security issues in a system (the main goal is to cope with the uncertainties related to security aspects).

The framework is thoroughly demonstrated and evaluated in the context of transactional systems, which can be divided in two parts: the *infrastructure* and the *business applications*. As these parts have significantly different security goals, the framework is used to develop methodologies and approaches that fit their specific characteristics. First, the thesis proposes a security benchmark for transactional systems infrastructures and describes, discusses and justifies all the steps of the

process. The benchmark is applied to four distinct real infrastructures, and the results of the assessment are thoroughly analyzed.

Still in the context of transactional systems infrastructures, the thesis also addresses the problem of the selecting software components. This is complex as evaluating the security of an infrastructure cannot be done before deployment. The proposed tool, aimed at helping in the selection of basic software packages to support the infrastructure, is used to evaluate seven different software packages, representative alternatives for the deployment of real infrastructures.

Finally, the thesis discusses the problem of designing trustworthiness benchmarks for business applications, focusing specifically on the case of web applications. First, a benchmarking approach based on static code analysis tools is proposed. Several experiments are presented to evaluate the effectiveness of the proposed metrics, including a representative experiment where the challenge was the selection of the most secure application among a set of seven web forums. Based on the analysis of the limitations of such approach, a generic approach for the definition of trustworthiness benchmarks for web applications is defined.

Keywords: Security, Benchmarking, Transactional Systems, Databases, Security Metrics, Security Evaluation, Security Benchmarking.

Resumo

A maioria das organizações depende atualmente de algum tipo de infraestrutura computacional para suportar as atividades críticas para o negócio. Esta dependência cresce com o aumento da capacidade dos sistemas informáticos e da confiança que se pode depositar nesses sistemas, ao mesmo tempo que aumenta também o seu tamanho e complexidade. Os sistemas transacionais, tipicamente centrados em bases de dados utilizadas para armazenar e gerir a informação de suporte às tarefas diárias, sofrem naturalmente deste mesmo problema. Assim, uma solução frequentemente utilizada para amenizar a dificuldade em lidar com a complexidade dos sistemas passa por delegar sob outras organizações o trabalho de desenvolvimento, ou mesmo por utilizar soluções já disponíveis no mercado (sejam elas proprietárias ou abertas).

A diversidade de software e componentes alternativos disponíveis atualmente torna necessária a existência de testes padronizados que ajudem na seleção da opção mais adequada entre as alternativas existentes, considerando uma conjunto de diferentes características. No entanto, o sucesso da investigação em testes padronizados de desempenho e confiabilidade contrasta radicalmente com os avanços em testes padronizados de segurança, os quais têm sido pouco investigados, apesar da sua extrema relevância.

Esta tese discute o problema da definição de testes padronizados de segurança, comparando-o com outras iniciativas de sucesso, como a definição de testes padronizados de desempenho e de confiabilidade. Com base nesta análise é proposta um modelo de base para a definição de testes padronizados de segurança. Este modelo, aplicável de forma genérica a diversos tipos de sistemas e domínios, define duas etapas principais: qualificação de segurança e teste padronizado de confiança. A qualificação de segurança é um processo que permite avaliar um sistema tendo em conta os aspectos e requisitos de segurança mais evidentes num determinado domínio de aplicação, dividindo os sistemas avaliados entre *aceitáveis* e *não aceitáveis*. O teste padronizado de confiança, por outro lado, consiste em avaliar os sistemas considerados *aceitáveis* de modo a estimar a probabilidade de existirem problemas de segurança ocultados ou difíceis de detectar (o objetivo do processo é lidar com as incertezas inerentes aos aspectos de segurança).

O modelo proposto é demonstrado e avaliado no contexto de sistemas transacionais, os quais podem ser divididos em duas partes: a *infraestrutura* e as *aplicações de negócio*. Uma vez que cada uma destas partes possui objetivos de segurança distintos, o modelo é utilizado no desenvolvimento de metodologias adequadas para cada uma delas. Primeiro, a tese apresenta um teste padronizado de segurança para infraestruturas de sistemas transacionais, descrevendo e justificando todos os passos e decisões tomadas ao longo do seu desenvolvimento. Este teste foi aplicado a quatro infraestruturas reais, sendo os resultados obtidos cuidadosamente apresentados e analisados.

Ainda no contexto das infraestruturas de sistemas transacionais, a tese discute o problema da seleção de componentes de software. Este é um problema complexo uma vez que a avaliação de segurança destas infraestruturas não é exequível antes da sua entrada em funcionamento. A ferramenta proposta, que tem por objetivo ajudar na seleção do software básico para suportar este tipo de infraestrutura, é aplicada na avaliação e análise de sete pacotes de software distintos, todos alternativas tipicamente utilizadas em infraestruturas reais.

Finalmente, a tese aborda o problema do desenvolvimento de testes padronizados de confiança para aplicações de negócio, focando especificamente em aplicações Web. Primeiro, é proposta uma abordagem baseada no uso de ferramentas de análise de código, sendo apresentadas as diversas experiências realizadas para avaliar a validade da proposta, incluindo um cenário representativo de situações reais, em que o objetivo passa por selecionar o mais seguro de entre sete alternativas de software para suportar fóruns Web. Com base nas análises realizadas e nas limitações desta proposta, é de seguida definida uma abordagem genérica para a definição de testes padronizados de confiança para aplicações Web.

Palavras Chave: Segurança, Testes Padronizados, Sistemas Transacionais, Bases de dados, Métricas de Segurança, Avaliação de Segurança, Testes Padronizados de Segurança.

List of Papers

This thesis relies on the published scientific research presented in the following peer reviewed publications.

Book Chapter:

Afonso Araújo Neto, Marco Vieira. 2012. Assessing the Security of Software Configurations. *In Threats, Countermeasures, and Advances in Applied Information Security*. IGI Global, 2012. Pages 129-157.

Journal Papers:

1. Afonso Araújo Neto, Marco Vieira. 2011. Selecting Secure Web Applications Using Trustworthiness Benchmarking. *International Journal of Dependable and Trustworthy Information Systems (IJDTIS)*. Volume 2(2):1-16.
2. Afonso Araújo Neto, Marco Vieira. 2011. Security Gaps in Databases: A Comparison of Alternative Software Products for Web Applications Support. *International Journal of Secure Software Engineering (IJSSE).* Volume 2(3): 42-62.
3. Afonso Araújo Neto, Marco Vieira. 2010. Benchmarking Untrustworthiness: An Alternative to Security Measurement. *International Journal of Dependable and Trustworthy Information Systems (IJDTIS).* Volume 1(2): 32-54.

Conference Papers:

1. Afonso Araújo Neto, Marco Vieira. 2011. Trustworthiness Benchmarking of Web Applications Using Static Code Analysis. *Proceedings of the Sixth International Conference on Availability, Reliability and Security (ARES).* Pages 224-229.
2. Afonso Araújo Neto, Marco Vieira. 2011. Selecting Software Packages for Secure Database Installations. *Proceedings of the Sixth International Conference on Availability, Reliability and Security (ARES).* Pages 67-74.
3. Afonso Araújo Neto, Marco Vieira. 2011. Towards benchmarking the trustworthiness of web applications code. *Proceedings of the 13th*

European Workshop on Dependable Computing (EWDC 2011). Pages 29-34.

4. Afonso Araújo Neto, Marco Vieira: TO BEnchmark or NOT TO BEnchmark security: That is the question. 2011. *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops. (HotDep 2011).* Pages 182-187.
5. Afonso Araújo Neto, Marco Vieira. 2009. Untrustworthiness: A trust-based security metric. *Proceedings of the Fourth International Conference on Risks and Security of Internet and Systems (CRiSIS 2009).* Pages 123-126.
6. Afonso Araújo Neto, Marco Vieira. 2009. Benchmarking Untrustworthiness in DBMS Configurations. *Proceedings of the Fourth Latin-American Symposium on Dependable Computing (LADC'09).* Pages 1-8.
7. Afonso Araújo Neto, Marco Vieira 2009. Appraisals Based on Security Best Practices for Software Configurations. *Proceedings of the Fourth Latin-American Symposium on Dependable Computing (LADC'09)*. Pages 57-64.
8. Afonso Araújo Neto, Marco Vieira. 2009. A Trust-Based Benchmark for DBMS Configurations. *Proceedings of the Pacific Rim Dependable Computing Conference (PRDC 2009).* Pages 143-150.
9. Afonso Araújo Neto, Marco Vieira and Henrique Madeira. An Appraisal to Assess the Security of Database Configurations. 2009. *Proceedings of Second International Conference on Dependability (DEPEND'09).* Pages 73-80.
10. Afonso Araújo Neto, Marco Vieira. 2008.Towards assessing the security of DBMS configurations. Proceedings of the IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN 2008). Pages 90-95.
11. Naaliel Mendes, Afonso Araújo Neto, João Durães, Marco Vieira, Henrique Madeira. 2008. Assessing and Comparing Security of Web Servers. *Proceedings of the Pacific Rim Dependable Computing Conference (PRDC 2008).* Pages 313-322

Table of Contents

1 Introduction 1
1.1 Benchmarking Security 4
1.2 Main Contributions of the Thesis 6
1.3 Structure of the Thesis 10
2 Background and Related Work 13
2.1 Overview of Computer Security Aspects 13
2.2 Security Evaluation 19
2.2.1 The Common Criteria 20
2.2.2 The OCTAVE method 24
2.2.3 The Center for Internet Security benchmarks 26
2.2.4 Additional Security Evaluation and Risk Analysis Methodologies ... 28
2.2.5 Security Characteristics Identification Techniques 29
2.3 Threat Modelling 30
2.4 Benchmarking 36
2.4.1 Performance Benchmarking 37
2.4.2 Dependability and Resilience Benchmarking 38
2.4.3 Security Benchmarking 39
2.5 Security Benchmarking as an Open Problem 41
2.5.1 Dependability Benchmarking *vs* Security Benchmarking 41
2.5.2 Benchmarking Trust 45
2.6 Conclusion 47
3 A Framework for Security Benchmarking 49
3.1 Threat Vectors as Basis for Benchmarking Security 54
3.2 Security Benchmarking Framework 56
3.2.1 Security Qualification 60
3.2.2 Trustworthiness Benchmarking 64
3.2.3 Instantiating the framework 67
3.3 Transactional Systems: the Case Study 70
3.3.1 Elements of a Transactional System 70
3.3.2 Security Benchmarking of Transactional Systems 72
3.4 Conclusion 74
4 Security Benchmarking of Transactional Systems Infrastructures ...75
4.1 Base Scenario 77
4.2 Security Qualification 80

4.3 Trustworthiness Benchmarking 81
4.3.1 Threat Vectors 84
4.3.2 Security Recommendations 91
4.3.3 Pessimistic Scenarios 101
4.3.4 Benchmark Procedure 108
4.3.5 Benchmark Metrics 111
4.4 Case Study 116
4.4.1 Systems Under Testing 117
4.4.2 Analysis of the Results of the Tests 118
4.4.3 Trustworthiness Assessment 121
4.5 Conclusion 125

5 Trustworthiness Benchmarking of Web Applications 127
5.1 Web Applications from a Security Perspective 130
5.2 Benchmarking the Trustworthiness of Web Applications using Static Code Analysis 134
5.2.1 Trustworthiness Metrics 135
5.2.2 Empirical Analysis of the Metrics 141
5.2.3 Experimental Evaluation 146
5.2.4 Lessons Learned 154
5.3 Towards a General Approach for Trustworthiness Benchmarking of Web Applications 156
5.3.1 Web Applications Code Threat Vectors 157
5.3.2 Security Precautions in Web Applications 158
5.3.3 Accounting for Secure Coding Practices 161
5.3.4 Trustworthiness Metrics 164
5.3.5 Preliminary Experimental Evaluation 167
5.4 Conclusion 169

6 Selecting Software for Transactional Systems Infrastructures 171
6.1 Identifying Security Mechanisms 174
6.2 Establishing the Impact of Security Mechanisms 179
6.3 Benchmark Metric and Execution 181
6.4 Experimental Evaluation 182
6.4.1 Software Packages Assessed 182
6.4.2 Comparing the Software Packages 182
6.4.3 Software Packages Gap Analysis 184
6.5 Conclusion 192

7 Conclusions and Future Work 195

References 203

Annex A Security Recommendations Tests, Weights and Analytical Results 221

Annex B Pessimistic Scenarios 233

List of Figures

FIGURE 2.1 OVERVIEW OF THE OCTAVE METHOD PHASES. 25
FIGURE 2.2 DEPENDABILITY *VS* PERFORMANCE BENCHMARKING 42
FIGURE 3.1 HIGH LEVEL VISION OF THE BENCHMARKING PROCESS 57
FIGURE 3.2 A TYPICAL TRANSACTIONAL SYSTEM ARCHITECTURE. 71
FIGURE 4.1 GENERAL UNTRUSTWORTHINESS FOR EACH SCENARIO.................. 122
FIGURE 4.2 UNTRUSTWORTHINESS FOR EACH THREAT, GROUPED BY CASE 123
FIGURE 4.3 ALTERNATIVE PRESENTATIONS FOR UNTRUSTWORTHINESS COMPARISON BETWEEN CASES... 123
FIGURE 4.4 FINE GRAIN ANALYSIS OF UNTRUSTWORTHINESS, FOR EACH CASE. 124
FIGURE 4.5 UNTRUSTWORTHINESS COMPUTATION FOR THE INTERACTION CLASSES 125
FIGURE 5.1 BENCHMARK RESULTS OF OUR CONTROLLED TPC-APP VERSIONS . 143
FIGURE 5.2 COMPONENT LEVEL EVALUATION OF RAW-NVR........................... 144
FIGURE 5.3 RAW-NVR EVOLUTION IN 16 VERSIONS OF 3 DIFFERENT SERVICES, RANGING FROM 0 TO 4 VULNERABILITIES 145
FIGURE 5.4 CALIBRATED METRIC ANALYSIS FOR THE 16 VERSIONS OF EACH SERVICE.. 146
FIGURE 5.5 OVERALL BENCHMARK RESULTS 168
FIGURE 6.1: MECHANISMS BY COMPONENT OF THE ANALYZED PACKAGES....... 185
FIGURE 6.2. AVAILABILITY OF MECHANISMS 185
FIGURE 6.3: NUMBER OF MECHANISMS AVAILABLE ACROSS PACKAGES. 186

List of Tables

TABLE 4.1 POTENTIAL THREAT VECTORS IN DBMS INFRASTRUCTURES 87
TABLE 4.2 DBMS CONFIGURATION SECURITY BEST PRACTICES DEVISED FROM THE ANALYSIS OF THE CIS DOCUMENTS 93
TABLE 4.3 COMPLEMENTARY DOD BEST PRACTICES 98
TABLE 4.4 BEST PRACTICE IMPACT KEY 100
TABLE 4.5 BEST PRACTICES ORDERED BY RELATIVE WEIGHTS 100
TABLE 4.6 PESSIMISTIC SCENARIOS ASSOCIATED WITH NOT FOLLOWING SECURITY RECOMMENDATIONS. 106
TABLE 4.7 SET OF ATTACKS CORRELATING THE PESSIMISTIC SCENARIOS AND THE THREATS 106
TABLE 4.8 MAPPING FOR THE FOURTEEN MOST IMPORTANT SECURITY RECOMMENDATIONS 108
TABLE 4.9 BENCHMARK SECURITY TESTS (SAMPLE) 109
TABLE 4.10 INFRASTRUCTURES DETAILS 117
TABLE 4.11 CASE 1, ORACLE 10G INSTALLATION 119
TABLE 4.12 CASE 2, SQLSERVER 2005 INSTALLATION 119
TABLE 4.13 CASE 3, MYSQL 5.0 INSTALLATION 119
TABLE 4.14 CASE 4, POSTGRESQL 8.1 INSTALLATION 119
TABLE 4.15 MOST IMPORTANT BEST PRACTICES YET TO BE IMPLEMENTED 120
TABLE 4.16 TESTS WITH UNANIMOUS RESULTS IN ALL FOUR CASES 121
TABLE 5.1 WEB FORUMS RANKED BY TRUSTWORTHINESS (TM) 148
TABLE 5.2 EXPERTS' RANKINGS 150
TABLE 6.1 CLASSIFICATION OF DATABASES SECURITY BEST PRACTICES IN REGARD TO THEIR REQUIREMENTS 176
TABLE 6.2 EXAMPLES OF THE MAPPING BETWEEN SECURITY BEST PRACTICES, SYSTEM STATE GOALS AND MECHANISMS GOALS. 177
TABLE 6.3 MOST IMPORTANT SECURITY MECHANISMS IDENTIFIED 180
TABLE 6.4. OVERALL RESULTS OF THE EXPERIMENTAL EVALUATION OF THE 7 DIFFERENT SOFTWARE PACKAGES. 183
TABLE 6.5 LIST OF MECHANISMS AVAILABLE IN ALL PACKAGES 187
TABLE 6.6 LIST OF MECHANISMS NOT AVAILABLE IN ANY OF THE PACKAGES .. 189
TABLE 6.7 LIST OF MECHANISMS AVAILABLE IN SOME OF THE PACKAGES (X MEANS THAT THE MECHANISM IS AVAILABLE IN THE CORRESPONDING PACKAGE) 190
TABLE 6.8 MECHANISMS AVAILABLE ONLY IN SPECIFIC SETS OF PACKAGES 192

TABLE A.1 SECURITY RECOMMENDATIONS DEVISED FROM THE ANALYSIS OF THE CIS DOCUMENTS 221
TABLE A.2 COMPLEMENTARY DOD CONFIGURATION BEST PRACTICES 223
TABLE A.3 BEST PRACTICES WEIGHTS 223
TABLE A.4 COMPLETE LIST OF TESTS 225
TABLE A.5 ANALYTICAL RESULTS OF THE INFRASTRUCTURES EVALUATED 230
TABLE B.1 COMPLETE LIST OF PESSIMISTIC SCENARIOS 233

1 Introduction

There is no disagreement nowadays about the importance of security in computer systems. The need for considering security as one of the pillars of any software architecture or implementation fills the pages of many popular newspapers and magazines, and the extensively documented consequences of security breaches range from public embarrassment to the loss of time, credibility and money.

Security of computer systems is a flourishing field with several distinct but complementary branches of research. Starting from pure theoretical aspects, like cryptography, security considerations are so wide that ultimately reach the complexity of the human factors that are inherently involved (Patrick 2003). Secure software design, development and configuration, attack mitigation and tolerance technologies, vulnerability discovery, analysis and prevention, are just some examples of research topics that are currently discussed in top security conferences. All these topics are applicable, generally or specifically, to almost all other branches of computer science. As a matter of fact, security research can be seen as a layer of concern that spreads in parallel with applied computer science research.

A key particularity of computer security is the central role played by human factors. In practice, there are two concepts that are fundamentally important in any security context: *capabilities* and *intention*. When analyzing a computer system from a security perspective, these concepts lead to questions that in other contexts are not usually taken into consideration. Take as example the following deliberation about a given System A: "*can any part of System A offer any advantage to any third party not considered in its specifications?*". The security implications of the potential answers to this question are clear when we instantiate System A as a system running in a bank, designed to manage bank accounts, in contrast to instantiating System A as a simple program that makes calculations (i.e. a software calculator). This exercise quickly leads to the following conclusion: the reason why the developer of the calculator program could, to a certain point, disregard security aspects, is that

breaking the specifications of the application does not pose any significant advantage (or disadvantage) to anyone, while that is clearly not the case for the system running the bank operations, at least in the majority of the contexts where these systems are used.

Although the *intention* of breaking a system's specification is usually related to the value that this action would provide to a given person/entity (which could, or not, be the *attacker* that attempts the breaking), it is important to realize that the intention is not the only condition to trigger an attack attempt: attacking a system also requires the attacker to have the means to do it. In that sense, the security of a system is not related only with the possibility of facing attacks, but also with the amount of resources – in a very broad sense – that are needed to break the system. Here, we are talking about the *capabilities* of the attacker.

In a simplistic view, the amount of effort required to break a system represents its *level of security*, which is a property independent from the motivations of the potential attackers. In fact, the amount of effort is a relationship between the technical capabilities available to the attackers versus the amount of barriers, or *security mechanisms*, which are put in place to disable those capabilities. If the mechanisms available allow nullifying completely the capabilities of an attacker, then the system can be classified as *secure* with respect to this particular attacker. In practice, when it comes to finding ways to effectively secure a computer system, the real challenge is answering the following questions:

Is the system already secure?

How far is the system from ideal security?

How do we modify the system in order to make it more secure?

Identifying all the potential attackers of a system is an impossible task, as this involves knowing exactly the persons to which the system has some kind of value (and as the system evolves and societies change, this becomes an endless, ever changing task). Given the pervasive nature and interconnectedness of computer systems, the only sensible approach is to assume that the system will be (sooner or later) attacked and that the attackers will have a considerable amount of resources available to accomplish the task. Due to this lack of precise knowledge, the approach followed by most organizations nowadays is to implement the highest number of security mechanisms they can afford, mostly following expert advice and intuitions about how much these mechanisms actually help.

An *ad hoc* approach to security, while usually helping in some way, has several clear disadvantages. Without a systematic method to properly assess the security of the system, the blind implementation of security mechanisms ends up being much more costly and less effective than it should. For instance, after adding a new security mechanism to the system, the inability to check if the security state really improved leaves the administrator with no clue whether the mechanism helped in any way. At the same time, if a highly secure state is achieved, the administrator cannot appreciate whether implementing any further mechanisms will be a waste of resources or not. Furthermore, while additional security mechanisms may effectively help in some way, the system administrators are left with no way to identify additional problems in the security barrier that may have been introduced by those same mechanisms. This *ad hoc* approach also leads to another unfortunate consequence: having a large number of costly mechanisms in place tends to transmit an unfounded sense of security. As the mere volume of security mechanisms never guarantees that all details are accounted for, the system may potentially be left with problems whereas the administrators think their security goals have already been accomplished.

Without a deterministic, representative and simple enough approach to evaluate security, it is utterly impossible for administrators to understand the security impact of systems' structural or functional changes. Administrators are also unable to make informed decisions concerning security aspects when it comes to tasks such as choosing between alternative software packages during the process of installing new software systems to support the organization activities. Means for reliably supporting the evaluation of the security level of computer systems are thus indisputably important.

The process of comparing systems in a standard, representative and accepted manner is called *benchmarking* (Gray 1993). In particular, a *security benchmark* is a method that is expected to support, at least, the following two tasks:

a) **Compare the level of security of a same computer system in two distinct points in its lifetime**. This process allows understanding how the security of the system varies when it is subjected to modifications, be these modifications of the system itself or of the environment where it is integrated in;

b) **Compare the level of security of alternative systems aimed at implementing the same task.** This allows making informed decisions regarding the selection of alternative software solutions taking into account existing organizational needs.

The computer industry already holds a reputed infrastructure for performance evaluation, where the Transaction Processing Performance Council (TPC) (TPC 2012) and the Standard Performance Evaluation Consortium (SPEC) (SPEC 2012) benchmarks are recognized as the two most successful benchmarking initiatives ever pursued. Furthermore, the concept of dependability benchmarking has gained ground in the last few years, having already led to the proposal of dependability benchmarks for several domains, including: operating systems, web servers, and databases and transactional systems in general (Kanoun and Spainhower 2007). Security, however, has been largely absent from previous efforts, in a clear disparity to performance and dependability benchmarking (Kanoun 2001, Vieira 2009). Researching alternative solutions for security benchmarking is precisely the goal of this thesis.

1.1 Benchmarking Security

Security evaluation methodologies have been proposed in several forms. One of the most popular security evaluation frameworks available is the Common Criteria standard (CC 1999) supported by the ISO/IEC group. While marginally allowing the comparison of systems, the Common Criteria is not considered a much successful approach due to several reasons, including its high complexity and emphasis in the analysis of specification documents instead of real implementations (Jackson 2007). Another important methodology for security evaluation is the OCTAVE® (Operationally Critical Threat, Asset, and Vulnerability Evaluation^{SM}) approach (Alberts et al. 2002) from the Software Engineering Institute (SEI) of the Carnegie Mellon University, which is a risk-based strategic assessment and planning technique for security. The OCTAVE approach is actually a set of security evaluation guidelines that support the process of self-evaluation within an organization. However, it is quite difficult to use this approach to compare different environments, to understand the impact of security decisions, or to evaluate the security of software alternatives, as the methodology is designed for the organization as a whole. Furthermore, risk based approaches require understanding the potential damage that attackers may cause in the assessed system, which is an extremely hard task due to the lack of historical data (Jaquith 2007).

Approaches like the ones introduced above can be classified as security evaluation methodologies, and indeed help organizations improving computer systems security when applied correctly. However, they are not suitable for supporting the tasks that a security benchmarking methodology is expected to support, as they are either too complex to be used by average system administrators or they require external expert analysis to be carried out (as is the case of Common Criteria). Expert analysis, in particular, is problematic in benchmarking contexts mainly

because *people change*. Being a standard approach, benchmarking requires the measurements performed in distinct points in time to be done using absolutely the same criteria, which is difficult to guarantee when we rely on what a person (or group of people) exactly knows. This nullifies the possibility to accomplish task a) (self-comparison in two points in time) mentioned previously, as we cannot be certain whether variations on the metrics are due to system changes or due to changes in the knowledge of the person running the benchmark. Benchmarking should rely on metrics that are standard and precise enough, so that evaluations in different points in time or of distinct targets are as little biased by external variables as possible (Gray 1993).

When trying to establish the security level of a computer system, in the terms mentioned previously, we find two distinct key perspectives. The first is related to actually finding real characteristics that can be exploited by attackers to cause some damage to the system or its owners. Those characteristics are usually called *vulnerabilities* or *weaknesses* and, depending on the system in question, may come from different aspects (Lyu 1996). For instance, when evaluating a web page, a typical vulnerability would be a software bug allowing attackers to apply input modifications capable of changing the pre-defined behavior of the application. Another example would be either *buffer overflow* vulnerabilities, which are coding mistakes that may allow injection of commands directly on the operating system, or *configuration vulnerabilities*, which arise from configuration inconsistencies or errors that may allow malicious users to obtain privileges they should not have (and therefore can be abused). Nowadays, the scientific community is putting a very significant effort in techniques and tools to find all sorts of vulnerabilities in all kinds of systems (e.g. penetration testing, static analysis, code inspections, etc.) (Livshits 2005, Long 2007, Antunes 2009). The vulnerabilities detected in a system may be corrected or not, depending on several contextual factors, but *finding them* is the main the goal of evaluation methodologies.

The second perspective for assessing the security level of a system is related to the fact that the entire set of vulnerability detection methods available nowadays is not enough to guarantee that systems are secure (as detectors typically suffer from coverage limitations) (Antunes and Vieira 2010). After trying to actually find existing vulnerabilities, we must consider the *probability* of the system still having hidden, hard to detect vulnerabilities, and that certain characteristics of the system may be used as leverage to facilitate attacks (e.g. an improper file system configuration may allow an attacker that has already gained access to the system to obtain even more information, or the fact that a server is not physically protected allows for alternate ways of gaining access the system). Such properties are much

more complex to find and evaluate as, by definition, they cannot be identified as vulnerabilities that either exist or not, but rather as characteristics that can raise the probability of occurrence of security incidents.

The type of analysis involved in *comparing* alternative systems in terms of their level of security is much more than simply trying to find actual vulnerabilities (Bondavalli 2009). In fact, when we are comparing two different systems, it is important to understand the following: even after applying a large amount of effort into finding vulnerabilities in two alternative systems, the fact that they both show zero obvious vulnerabilities does not mean that they are equally secure. This is mainly due to our inability to assure that no other vulnerabilities exist. This way, distinguishing the security level of two systems with no obvious vulnerabilities is still an open problem, which we thoroughly discuss in this work.

In this thesis we propose a security benchmarking framework that takes into consideration the issues and difficulties just presented. The fundamental assumption of our proposal is that to achieve fair comparison, security benchmarking must necessarily consider the two perspectives mentioned previously: the active search for vulnerabilities and security problems, and the propensity for other hidden or unidentified problems to exist. This is crucial, especially because each of these perspectives arise from different systems characteristics and may lead to different considerations when such information is used to support decision making processes.

1.2 Main Contributions of the Thesis

In this thesis we study the problems involved in performing security benchmarking, and show the type of concerns and characteristics that such benchmarks should have in order to attain their goals (i.e. allow comparing alternative solutions from a security point-of-view). To the best of our knowledge, we propose the first generic framework that is designed to support practical, representative, and useful security benchmarks.

The proposed security benchmarking process is divided in two key steps: *security qualification* and *trustworthiness benchmarking*. The first step is where the System Under Test (SUT) is evaluated to have a minimum level of security in order to be considered acceptable for use in a given application domain. The goal of this step is to actively try to find vulnerabilities in the system and also evaluate the security mechanisms it provides. A SUT that fails this step is automatically classified as insecure, with security level equal to zero. In a comparison process, where two or

more SUTs are being compared, this step is *qualificatory*, in the sense that the systems with vulnerabilities are immediately disqualified for practical use.

The second step of the benchmarking process, which makes sense only for systems that pass the first one (and therefore have no obvious vulnerabilities), is based on *trustworthiness benchmarking* concepts. This step is designed to provide, to a certain extent and given some premises, a relative level of probability that the SUT may be compromised when facing attacks that try to accomplish certain malicious effects. In a way, trustworthiness benchmarking provides the level of trust that a user can justifiably have when it comes to the ability of the system in avoiding a specific set of threats. In other words, the goal is to identify the system characteristics that entitle it to be *trustworthier* in face of uncertainties.

The proposed framework is a guide for the definition of concrete security benchmarks for specific application domains. In this thesis we present and discuss thoroughly the framework, devoting particular attention to the reason why a security benchmarking process should be divided and structured in such a way. Understanding the motivations for this benchmarking approach allows identifying its properties and, in particular, its limitations, which are also extremely important.

As a case study and proof of concept, we apply the proposed framework to design and run security benchmarks in the context of *transactional systems*, also referred to as On-Line Transaction Processing (OLTP) systems (Vieira 2003). These systems are characterized by having a central Database Management System (DBMS) and several remote clients running one or more applications that define the business rules of the data that are stored in the database. In this context, we divide a transactional system in two main parts (the *transactional system infrastructure* and the *business applications* that use the infrastructure) and proposed a specific benchmark approach for each of them. By applying our framework to both complex and simple realistic scenarios, we aim to demonstrate its generality and practical viability.

The focus on transactional systems is justified by the fact that this kind of systems are used to support the business operations of almost all organizations, making them a very representative use case (Sawyer 1993). Additionally, managing a transactional system is a complex task that many times is performed by people with very little security knowledge. This is a key concern as the security of such systems is absolutely vital for the success of a company's business. Therefore, a way to systematically evaluate and compare the security of transactional systems without complex trainings or requirements is of utmost importance.

In summary, the main contributions of this thesis are:

- *A survey on the state of the art on security evaluation and computer systems benchmarking.* The first important contribution of this thesis is the systematization of the work that has been done in the security evaluation and benchmarking domains. We discuss some of the existing approaches and identify the major aspects and difficulties that should be considered when devising generic security benchmarking approaches.
- *A security benchmarking framework composed of two steps (qualification and trustworthiness benchmarking) based on a reference domain and representative threat vectors for that domain.* Considering the difficulties identified before, we develop a framework aimed at overcoming those difficulties. The framework breaks the problem in two parts, each one providing a particular semantic outcome: the first is related to what we can clearly evaluate about systems security, and the second is related to the aspects that we can only estimate. The reasoning behind this approach and the goals of each step of the benchmarking process are discussed in detail.
- *The application of the proposed security benchmarking framework to the domain of transactional systems*, in order to study and understand its effectiveness and viability. The first consequence of the framework instantiation is the need for dividing the transactional system in two parts: the *transactional system infrastructure*, and the *business applications* based on that same infrastructure. This results from the fact that the security goals of these two parts are essentially different, and the framework automatically forces the benchmark to have a consistent view of them. The instantiation of the framework to each of these parts resulted in several complementary contributions, as presented next.
- *A security benchmark for transactional systems infrastructures*, which resulted in the following detailed contributions:
    - A representative *set of security recommendations for transactional systems infrastructures*, which can be used to support other assessments and security evaluation methodologies besides the proposed benchmark.
    - A *set of representative threats* that should be of knowledge of any database administrator, and a *set of security tests* that can be used for understanding the security problems that may arise in transactional system infrastructures.

  - A *complete trustworthiness benchmarking methodology and implementation for transactional system's infrastructures*, which allows understanding, from a high level perspective, the biggest security concerns that may manifest in the infrastructures under benchmarking. To demonstrate its effectiveness, the proposed trustworthiness benchmark was applied to four different real transactional systems infrastructures.
  - *The development of a tool to assist on the selection of the software components* (e.g.. DBMS engine and operating system) that best fit the security requirements of the transactional system infrastructure. This tool was used to assess seven representative distinct software packages (i.e. a combination of several DBMS engines and operating systems), which allowed evaluating them from the point-of-view of the existing security mechanisms.
- A study on the implementation of the framework in the context of web-based business applications, which resulted in the following contributions:
  - *A detailed discussion on alternatives for conducting trustworthiness benchmarking of business applications*, taking into account the security characteristics of the code of the applications under benchmarking. The study was done focusing on web technologies, which are the technology of choice nowadays, and whose security is largely dependent on the correct design.
  - *A detailed study*, including *a complete validation cycle, on the use of static code analyzers* as reliable and effective tools for the automated computation of trustworthiness metrics in web applications. In detail, we considered a representative use case, where a user would have to choose the most secure among seven existing software alternatives (in this case, seven web forums), and compared the automated benchmark proposal with the evaluation conducted by six security experts. The comparison of the results allowed the validation of the effectiveness of our proposal, along with the identification of its most important advantages and limitations.
  - *The proposal of a generic approach for the definition of trustworthiness benchmarks for web applications*, based on the findings of the previous study. In this case, we focused on the design of a tool that does not depend on the characteristics of static code analyzers, which could eventually change due to a diversity

of factors. Even though we did not implement a real tool based on this generic approach, we demonstrate it by manually computing and interpreting the metrics in a small-scale scenario.

It is important to emphasize that all the studies, proposals and methodologies are accompanied with detailed *justifications and discussions about their limitations*, particularly about why and how they could fail their objectives. In fact, it is probable that the most important contribution of this thesis are not the tools or studies presented, but rather ***a consistent view on how to correctly rationalize security aspects when the goal is fair comparison***.

1.3 Structure of the Thesis

This thesis is divided in seven chapters, as described in the following paragraphs.

Chapter 1 introduces the problem of security benchmarking and describes the main contributions of the thesis.

Chapter 2 presents the background and existing work related with this thesis. Section 2.1 presents an introductory view to security of computer systems. Section 2.2 presents several security evaluation frameworks and methodologies, focusing on the few that are more important in the context of our work. Section 2.3 presents techniques and approaches for threat modeling, which is an important aspect of security evaluation. Section 2.4 presents a description of the evolution of benchmarking, from performance to dependability benchmarking. Section 2.5 presents a discussion about the main difficulties of security benchmarking (particularly in contrast to the dependability benchmarking model), and presents a discussion about the idea of benchmarking trust and how the concept could be related with security benchmarking.

Chapter 3 presents the security benchmarking framework. Section 3.1 discusses the aspects that have to be considered when benchmarking security. Section 3.2 presents the concept of *threat vectors*, why they are needed and how to understand them. Section 3.3 describes the framework, starting with a general view, and then detailing the qualification and trustworthiness benchmarking phases. Section 3.4 presents a decomposition of transactional systems needed to apply the framework, justifying why, this has to be done.

Chapter 4 describes the application of the framework to the context of transactional systems infrastructures. Section 4.1 describes the base scenario used as a frame of reference for the whole benchmark, justifying its characteristics and representativeness. Section 4.2 put forward some ideas about the security

qualification step. Section 4.3 describes our approach for the evaluation of trustworthiness benchmarking of transactional systems infrastructures, including the threats vectors, the list of security elements, the *pessimistic scenarios*, the actual benchmarking tool, and the metrics. Section 4.4 is about the application of the benchmark to four distinct real infrastructures.

Chapter 5 presents the study of trustworthiness benchmarking approaches in the context of business applications, using as case web applications. Section 5.1 presents a general discussion of the security of web applications. Section 5.2 describes the set of experiments we conducted to evaluate the plausibility of using static code analysis tools to accomplish trustworthiness benchmarking. Section 5.3 draws from the limitations identified in the previous experiments, and proposes a general targeted approach for trustworthiness benchmarking of web applications. In both section 5.2 and 5.3, several experiments are presented.

Chapter 6 discusses the problem of security qualification when applied to transactional systems infrastructures, proposing a tool that can help in the selection of the software components needed to support that infrastructure. Section 6.1 describes how to identify security mechanisms from a set of security recommendations. Section 6.2 discusses the identification of the impact of such mechanisms. Section 6.3 discusses the metrics that are computed by the tool. Section 6.4 presents an experimental evaluation of the tool, where we used it to assess seven distinct software packages, consisting of multiple database management systems and operating systems.

Chapter 7 presents generic conclusions and a general overview of the main lessons of this thesis, also putting forward future work that is directly related to the achievements of this thesis.

2

Background and Related Work

This chapter presents the fundamental concepts and overviews the state-of-the-art on techniques related to security evaluation and benchmarking. We start by revising the most important concepts regarding security, and then discuss existing approaches for security evaluation, introduce the concepts behind benchmarking in general, and discuss the main difficulties related to security benchmarking. Even though security aspects are vast and can be rationalized from a series of perspectives (from the technical aspects to the human factors and their relation with security in general), we introduced these topics from the perspective of their relevance to the approaches, techniques and tools proposed in the rest of the thesis.

This chapter is organized as follows. Section 2.1 introduces basic computer security concepts. Section 2.2 presents an overview of relevant security evaluation methodologies and techniques. Section 2.3 addresses threat modeling and Section 2.4 presents related work on benchmarking in general, and on the approaches that are being applied for security benchmarking. In Section 2.5, we discuss the main motivation for this thesis, putting it in contrast to the current state-of-the-art, and also discuss the idea of benchmarking trust and how this concept can be related with security attributes. Finally, Section 2.6 concludes the chapter.

2.1 Overview of Computer Security Aspects

Before addressing more specialized topics, it is necessary to define some aspects and characteristics of the terminology related to computer security. The term *computer security,* which is actually the idea of *information security* applied to computers, is an integrative concept that includes all aspects related to the preservation of the several different properties that can be attributed to a specific information asset (Russell 2011). However, to deeply understand the relevance of

these properties, it is necessary to define beforehand the elements over which they apply.

First of all, security only makes sense when there is something to be *secured*. It is important to understand that the goal of computer security has nothing to do with security of hardware or people, even though it might involve these in certain cases. What computer security is concerned with is the security of the *information* that is generated, accessed and stored by computer systems (Siponen 2007). More specifically, within a given environment, the information always suffers a set of actions that might generate more information or trigger more actions. The rules that define the transformations that can or cannot be applied to the information are called *business rules* and represent essentially the principles that the system must follow to fulfill its objective. Each transformation defines not only the outcome, but also the allowed executors (usually the persons or other transformations that are allowed to trigger it). In that sense, computer security is related to ensuring that the information within the system will follow the business rules despite *anything else*, even assuming a very intelligent malicious person (or group of people) with an unpredictable amount of resources, trying to break *any* of the rules.

Given a certain system, there is a multitude of ways through which the business rules can be violated. However, not all of these ways are security concerns, as some cannot cause any type of damage or loss to any of the people involved or affected by the system. For example, a typical operator error (like a mistype) is not usually a security concern but might be an example of a business rule violation. A breach of a rule that is related to security is usually called a *security incident* or simply an *attack* (Russel 2001). The methods and techniques used to execute the attacks are referred to as *attack vectors* or *attack methods* and the set of all attack vectors present in a system defines its *attack surface.*

The issues that historically are considered security concerns are related to the violation of the following information properties (Parker 2002):

- **Confidentiality** – property that guarantees that the information is not accessed, used, copied, or disclosed by anyone except the authorized individuals.
- **Integrity** – property that guarantees that the information is not created, changed, or deleted by individuals without proper authorization.
- **Availability** – property that guarantees that the information is timely and correctly available to authorized individuals.

For a long time, confidentiality, integrity and availability were considered the core properties of information security. In fact, these three properties were considered complete enough to be the only properties that the security mechanisms would be in charge of preserving. However, Donn B. Parker (Parker 2002) pointed out some small deficiencies in the original set of properties, and showed that some particular types of very important attacks could not be specified by the loss of any of these properties. He introduced three other security properties of information:

- **Authenticity** – this property refers to the guarantee that the information is correctly labeled and that it is in fact what is said about it. This property is distinct from integrity because the information might not have been altered or deleted, but still be understood in a different way from what it was meant to. Fraudulent information is an example of non-authentic information that is correct from the point of view of its authorized creator. The security problem is that this information is not what its creator said it is.
- **Possession or Control** – the information can be out of the control of the rightful owner, possibly being transferred to someone else or used in a non-authorized way. This property is distinct from confidentiality because an attacker can violate it without violating confidentiality and vice versa (e.g. when the attacker takes control of a machine but does nothing with this control). Another important kind of breach is when one makes an unauthorized copy of a copyrighted intellectual work (like a movie). Notice that in this case there is no breach of confidentiality (the owner is authorized to see the movie), no breach of integrity and the information is available to its rightful owner.
- **Utility** – probably the most controversial "complementary property", utility is related to guaranteeing that the information can still be used for its original purpose. The most common example for a breach of utility is when a user encrypts some data and then loses the encryption key. The idea is that the data is still confidential, available (it is there), integral (it is correct), under control and authentic, but cannot be used anymore because of a transformation that cannot be undone. Unauthorized source code obfuscation sometimes is also used as an example, as the code still compiles and generates the corresponding executable code, but can hardly be modified anymore (without an effort that would not be necessary with the original code). The critics, on the other hand, say that utility can always be understood as one of the other properties. In the first example, the data is actually not available anymore exactly as it would not be in the case of a hard drive that cannot be turned on (but with no damage to the magnetic data). In the second example, the source code is not integral anymore because it has lost the semantics that was present only in the original source code.

Most of the security properties of information are defined in terms of the figure of an *authorized person*. Although, usually, most actions in a system are executed by real people, sometimes the actions might be also triggered by other systems (for which *authorized agent* would probably be a more accurate term). *Authorization* in this context is directly defined by the business rules of the system and specifies the set of actions that each agent within the system has the right to execute and the set of actions that it cannot execute. Usually, there is also a default policy for all actions not explicitly defined, which could be "all else is authorized" or "all else is denied", and also depends on the purpose of the system. The mechanisms through which authorization is actually implemented in a system might vary a lot (*privileges* or *access controls lists* are two common examples). However, the most complex security issue involved is related to identifying precisely who should have which authorization. This is called an *authentication* procedure (Daswani 2007) and is the process of assuring, to the desired level of certainty, that someone really is who he claims to be.

Another security property, which can be considered as a special case of authenticity, but is, frequently, considered separately, is **non-repudiation** (Stallings 2010). This property is related to guaranteeing that if someone performs an action then that action cannot be denied in a later future. For example, the idea of *digital signatures* only works when the system is built with non-repudiation in its core, meaning that it has the same properties of *undeniability* of a traditional signature. Although not necessary in every context, several other scenarios might require the preservation of this property (possibly not in a so strong form). For instance, if a legitimate system operator excludes some information then it is important that the system registers, in a reliable way, *who* performed the exclusion, generating evidence that cannot be hidden. This kind of *auditing* preserves this property not as strongly as a digital signature (that no one should be able to forge), as the system administrator may be able to alter the evidence in some way. However, the property holds because the operator does not have the same privileges that of the administrator and that is sufficient for this purpose. In this case, the system administrator is expected to have the power to view or modify data in the system, and therefore that is not a security breach. The fact is that he is supposed to do it only according to what are his authorized assignments.

A malicious administrator (which is an example of an *insider threat*, (Martinez-Moyano 2006)) eliminating evidence or using its privileges to abuse the system in some way provides an example of the *abuse of trust* (Bishop 2008). This is a problem that circumvents any computer system and does not have a definitive solution. The biggest difficulty, in this case, is that it is mostly a human aspect and

not a technical one (Whittaker 2003). In any system, some amount of trust is posed upon all people involved, being it the administrator who bears a very large amount of trust, or a simple end user that has a very limited, but non-negligible, amount of privileges. The problem is inevitable because whenever the system poses any amount of authorization to a given individual, it is opening the possibility for someone to find ways to abuse it and break the rules. Actually, it is a known fact that the most access a person has to the system the higher is the number of combinations of actions that it can perform. Some of these actions can readily be used to cause a security breach and avoiding it is, in most scenarios, completely unfeasible. The *principle of least privilege*, which is to always place the least amount of privileges possible to any element within a system, is one of the most important and recognized principles of authorization distribution. This principle has been proposed more than 30 years ago, and has been proven right since it was first discussed in (Denning 1976).

Computer security research is done not only to understand security aspects but also to develop *security mechanisms* designed to fulfill several goals. Mechanisms for the preservation of the security properties, mechanisms to allow reliable authentication and authorization and mechanisms to lower or eliminate the possibility of abuses of trust are just some examples. These security mechanisms are commonly known as *security controls* and can be classified in several forms. When a security control is active in a system and a security incident is about to happen, there are three moments in which the control may act (Bowen 2006):

- It may act before the occurrence of the incident (or its completion), effectively avoiding its occurrence. In this case it is called a *preventive control*. An authentication mechanism is an example of a preventive control.
- It may act during the incident by trying to identify its occurrence and, when possible, activate an alert so the person responsible can act accordingly. This is called a *detective control* and auditing and logs are examples of this type of control.
- It may act after the incident, possibly reducing or eliminating the consequences of the attack. These are called *corrective controls*. Backups and redundant servers are examples of it.

Security controls can also be classified in regard to their nature. They may be classified in one of the following four categories (Bowen 2006):

- *Physical controls, e.g.* fences, doors, locks and fire extinguishers;

- *Procedural controls, e.g.* incident response processes, management oversight, security awareness and training;
- *Technical controls, e.g.* user authentication and access controls, antivirus software, firewalls;
- *Legal and regulatory or compliance controls, e.g.* privacy laws, policies and clauses.

In theory, a system implementation together with its environment and appropriate security controls are expected to not be susceptible to attacks (as that is the goal of the security controls). However, in practice, it is impossible to have a completely secure system, especially if one considers an insider threat. The *weaknesses* that the system still presents and can be used as attack vectors, despite the security controls in place, are called *vulnerabilities* (McGraw 2006). Examples of classical vulnerabilities are software bugs or incorrectness (e.g., a buffer overflow and SQL injection attacks (Daswani 2007)), authentication weaknesses (e.g., the existence of weak passwords (Blackwell 2000)), configuration problems (e.g., a poorly configured firewall (Wool 2004)), or even a physical security problem (e.g., leaving the database server stationed in an uncontrolled room full of unauthorized people).

Instead of being cases of exception, more and more the computer science research community is learning that vulnerabilities cannot be completely eliminated, despite all efforts to avoid them (McGraw 2006). As a consequence, two important guidelines are frequently emphasized as key security practices that should be applied to any context: *security by design* and *defense-in-depth* (Howard 2002).

S*ecurity by design* means thinking about the security of a system while designing it, instead of considering security as a new layer of features. This turns out to be a much more successful approach because of a simple fact: when one adds security functionalities to an existing system, the number of inconsistencies (i.e. vulnerabilities) that can emerge from the combination of the original state (without security controls) with the state with the new functionalities (the security controls) is much higher than the number of defects in a system that was designed with these functionalities from scratch. In other words, the *attack surface* of a system designed with security in mind is always smaller than the *attack surface* of a system that has been secure by the later appliance of security controls.

Defense-in-depth, on the other hand, is the idea of always assuming that the security controls can be surpassed. In other words, instead of protecting a system with one huge barrier, always consider that each part of the system must be secured independently as if all other barriers were already defeated. The principle of least privilege, for instance, is an example of the application of the principle of defense-

in-depth. If some attacker takes control of some agent in a system (e.g. a process) the damage it can do is limited by the original purpose of the agent if this agent does not have more privileges than the ones it needed in the first place. Securing a network with a global firewall and still having local firewalls on the operating systems of the machines on that network is also another example of defense-in-depth.

2.2 Security Evaluation

Computer security evaluation, in some contexts referred to as *risk analysis* applied to computer systems, has been a concern for organizations and systems administrators for a long time. To decide if the security mechanisms present in an installation are enough or should be improved, first it is necessary to evaluate them. Security evaluation is the process of determining how well the security controls of a given system are working and how effective they are against known attacks and threats (Bowen 2006).

The challenge faced by systems administrators is that computer security evaluation is a task that requires a very specialized knowledge. To perform a reliable evaluation, the analyst must have the capability for understanding all factors at stake, the nature of the threats involved, and how the security controls in place work, and these topics are usually not part of the administrators' training. To solve this, the choices are either hiring outside help or learning and applying an appropriate security evaluation methodology.

The urge in proposing security evaluation methodologies was always historically so strong that several private and governmental organizations have invested a lot of time and money on it. For example, in the early 80's, the government of the United States through its Department of Defense started developing what later would be called the *Rainbow Book Series.* This is a series of standards designed for the evaluation of trusted systems, and describes the process to be used inside the US government. In particular, the *Trusted Computer System Evaluation Criteria* (DoD 1985), also known as the Orange Book, is a standard that sets basic requirements for assessing the effectiveness of computer security controls built into a computer system.

In 1999, the concepts in the Orange book were merged together with other related standards like the *Canadian Trusted Computer Product Evaluation Criteria* (Mate Bacic 1990) and the *Information Technology Security Evaluation Criteria* (Jahl 1991), giving rise to a new international standard that was supposed to be accepted worldwide. The *Common Criteria for Information Technology Security Evaluation*

(Common Criteria 1999), or simply the Common Criteria, became the standard ISO/IEC 15408 in a joint action of the International Organization for Standardization (ISO 2012) with the International Electrotechnical Commission (IEC 2012).

This section focuses on the main aspects of three of the most representative approaches: the Common Criteria framework, the OCTAVE method and the Center for Internet Security benchmarks. These methodologies were chosen because they provide very distinct approaches to security evaluation, and most others either resemble one of them or share characteristics. However, additional methodologies for security evaluation and risk analysis are introduced in Section 2.2.4.

2.2.1 The Common Criteria

The Common Criteria standard (Common Criteria 1999) is a security evaluation framework that defines a process where a computer system is evaluated against a set of security requirements. The evaluation results in a level of assurance, or Evaluation Assurance Level (EAL) and a certification from the Common Criteria. Essentially, the assurance level expresses the effort that was applied by the Common Criteria evaluators in order to be certain that the system has the security requirements that it claims to have. The first draft of the standard was published for comments in 1993, and finally became an official ISO standard in its version 2.0, in 1999. The main objective of the standard was to replace the security evaluation and processes used in different countries by a unified process that would be accepted by all of them. This would allow product evaluations conducted in one country to be accepted in other countries.

For a given *Target Of Evaluation (TOE)*, which is the product or system under assessment, the evaluation within the Common Criteria framework is based on a fundamental document that describes the characteristics of the TOE: the *Security Target*. The security target, on the other hand, may or may not reference another document called a *Protection Profile*. Both documents are structurally similar but have distinct purposes. However, understanding a protection profile allows to more easily understanding a security target.

The protection profile identifies the security requirements that the particular TOE must implement in order to be secure against an identified set of threats typically found in environments surrounding it. In other words, a protection profile is an implementation independent statement of security requirements that address threats in a specific environment. The most important elements that are part of a Protection Profile are:

- *Security Environment definition*: a high level description of the environment where the TOE typically operates.
- *Secure Usage Assumptions*: definitions about some important characteristics of fundamental elements of the environment. For example, some characteristics of the network, considerations about the kind of physical control that is assumed regarding the TOE or the characteristics of trustworthiness of the administrators. These assumptions are the basis over which the evaluation is valid.
- *Organizational Security Policies*: the policies that the organization must enforce in order for the product to effectively have the security stated.
- *Threats to security*: enumeration of the security threats that must be addressed by the implementation of the TOE in order to be considered secure in the sense of this Protection Profile.
- *Security Functional Requirements*: high level security elements that must be present in the TOE implementation and that should be employed to avoid the threats identified before. These elements are catalogued by the standard, and form eleven classes divided in 67 families, 138 components and 250 elements.
- *Security Assurance Requirements*: the evaluation requirements to be performed over the TOE as to be able to certify it with a specific Evaluation Assurance Level. The possible assurance requirements are also catalogued by the standard.

A protection profile is a document defined generically, meaning that it is *implementation independent*. In practice it defines a class of devices or scenarios working in a specific environment. For instance, it is possible to define a protection profile for a firewall in a particular scenario, or a smart card in another scenario. The definition of different protection profiles for the same class of devices is possible as well, with different security requisites for each one. Basically, the main purpose of protection profiles is to provide means for some person or organization to express the security requisites that are necessary for a given purpose. A government, for example, might require a particular product to be certified against a specific protection profile before considering its acquisition.

The security target, on the other hand, specifies the characteristics of the product or system that will undergo the certification process. It can be seen as an instantiation of what would be a generic protection profile relatively to a particular product, and is usually provided by the developer of the product. A security target typically includes all elements that are part of a traditional protection profile, but explains how they are applied to the product in question. It also includes a detailed description of the mechanisms that are implemented to satisfy the security

functional requirements. Although not required, usually it also mentions a list of protection profiles which the TOE might comply with. The TOE is then evaluated against all of them, and the certification states that.

The most important part of a security target or of a protection profile is the definition of the security functional requirements expected from the TOE. The standard defines the following eleven high level classes of functional requisites that a system or product might have:

- *Security Audit* – monitor, capture, store, analyze, and report information related to security event.
- *Communication* – Assure the identity of originators and recipients of transmitted information; non-repudiation.
- *Cryptographic Support* – Management and operational use of cryptographic keys.
- *User Data Protection* – Protect user data and the associated security attributes within a TOE and data that is imported, exported, and stored.
- *Identification & Authentication* – Ensure unambiguous identification of authorized users and the correct association of security attributes with users and subjects.
- *Security Management* – Management of security attributes, data, and functions and definitions of security roles.
- *Privacy* – Protect users against discovery and misuse of their identity.
- *Protection of the TOE Security Functions*– Maintain the integrity of the TSF management functions and data.
- *Resource Utilization* – Ensure availability of system resources through fault tolerance and the allocation of services by priority.
- *TOE Access* – Controlling user session establishment.
- *Trusted Path Channels*– Requirements for trusted paths and trusted channels.

The assurance requirements defined in the security target will set the level that the implementation of the TOE will be evaluated. In any certification process, the evaluation is done by the application of the *Common Methodology for Information Technology Security Evaluation* (CEM), also part of the standard. The evaluation process is done by a third party laboratory complying with the ISO/IEC 17025 (Honsa 2003), which certifies and states management and technical requirements for testing and calibration laboratories. A successful evaluation provides a

certification of the TOE within one of the seven possible levels of assurance, with the following corresponding rigorousness:

- *EAL 1* – the TOE is functionally tested and a minimum level of confidence in the correct operation of the security functions is guaranteed. This EAL is appropriate for environments where no serious security threats are anticipated.
- *EAL 2* - the TOE is structurally tested, and a low to moderate level of confidence in the correct operation of the security functions is guaranteed. This EAL is assigned to systems for which little documentation exists.
- *EAL 3* - the TOE is methodically tested and checked and a moderate level of confidence in the correct operation of the security functions is guaranteed. EAL 3 represents a thorough investigation of the TOE and its development, starting at the design phase. Testing and evaluation are conducted against functions, interfaces, and guidance documents.
- *EAL 4* - the TOE is methodically designed, tested, and reviewed and a moderate to high level of confidence in the correct operation of the security functions is guaranteed. EAL 4 is the highest level of assurance usually provided to commercial off-the-shelf software.
- *EAL 5* - the TOE is *semiformally* designed, tested, and reviewed, providing moderate to high level of confidence in the correct operation of the security functions. EAL 5 is appropriate in environments where resistance to attackers with a moderate attack potential is needed.
- *EAL 6* - the TOE is *semiformally* verified design and tested, and provides a high level of confidence in the correct operation of the security functions. To be evaluated as EAL 6, the software design requires the use of systematic security engineering practices and techniques.
- *EAL 7* - the TOE is formally verified design and tested, providing a very high level of confidence in the correct operation of the security functions. EAL 7 represents complete, independent white-box testing that employs formal methods, similar to those in use by the safety engineering community. EAL 7 is intended for use in extremely high-risk environments that must protect high-value assets.

Despite its popularity, the Common Criteria is not a standard unanimously accepted. The major criticism against the standard is that it tests almost only the design of the product, and not the implementation, even at the highest levels of evaluation. In the words of Alan Paller, director of research at the SANS Institute, "*You are not testing the product at all. You are testing the paperwork*" (Jackson 2007). As it is, a certificated product is a long distance from been considered secure, so the cost of certification (which is very significant) is actually not worth

it. One recurring example against the standard is the certification with EAL4 of the Windows 2000 operating system, which continuously had security corrections long after the certification (Baumhardt 2006). Jonathan Shapiro, assistant professor at Johns Hopkins University also puts it as not worth it: "*The evidence so far suggests that it is a waste of time and resources. I would be extremely happy to see evidence to the contrary, but it doesn't seem to be out there*" (Jackson 2007).

Another criticism against the common criteria is that the certification is valid only in regard to the security target document (and mentioned protection profiles). A certification, even with a correct implementation, means that the product is secure only with the configuration and environment defined in the document. For example, the configuration and environment defined in the Windows 2000 certification strips it of so much functionalities (for example, the *Internet Explorer browser* and the *Internet Information Services)* that sometimes it turns out to be almost a useless shell. In that sense, practically all installations of this operating system running today invalidate the certification.

2.2.2 The OCTAVE method

The Operationally Critical Threat, Asset, and Vulnerability Evaluation method (Alberts 2002) was developed in 2003 by the Software Engineering Institute (SEI) at Carnegie Mellon University on behalf of the Department of Defense of the United States government. It is a self-directed risk assessment methodology, suited for small teams of people from the operational and the IT departments of an organization.

A fundamental difference of the OCTAVE approach comparing to most proposed risk assessment methodologies, is that it is driven mostly by operational risk and security practices instead of pure technology considerations. The design of the approach is aimed at allowing an organization to:

- Perform self assessments without outside requirements;
- Identify risks that are particular to the organization business and operations;
- Identify and focus on the protection of the most important information assets of the organization;
- Raise the security awareness at all levels of the staff.

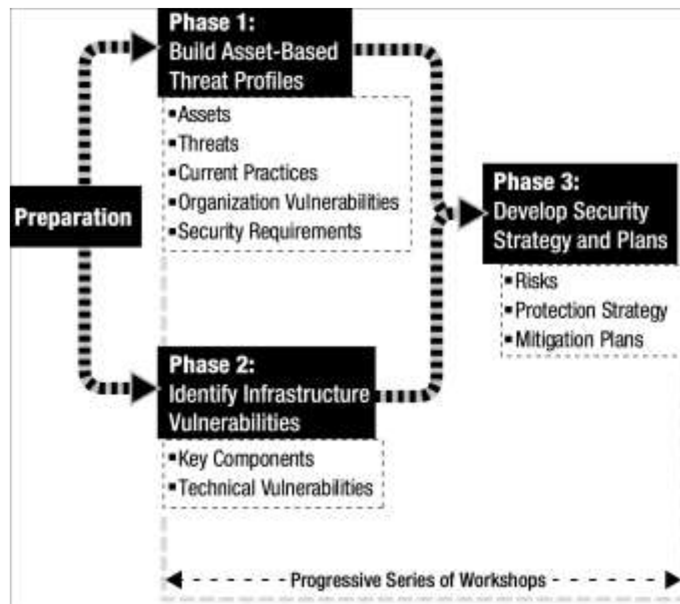


Figure 2.1 Overview of the OCTAVE method phases.
(Alberts 2002)

Figure 2.1 presents a high level view of the methodology steps. The OCTAVE method is based on three main phases that are further broken down into processes, and evolves through a series of workshops carried out by the analysis team. In Phase 1, the analysis team identifies important information-related assets and the current protection strategy for those assets. The team then determines which of the identified assets are most critical to the organization's success, documents their security requirements, and identifies threats that can interfere with meeting those requirements. In Phase 2, the analysis team performs an evaluation of the information infrastructure to complement the threat analysis performed in Phase 1 and to support mitigation decisions in Phase 3. Phase 3 includes risk identification activities and the definition of a risk mitigation plan for the critical assets (Alberts 2002).

All OCTAVE phases are supported by catalogues of information provided by the method, which are designed for teams without security expertise and without outside help. The main catalogues are the following:

- *Catalogue of practices* - a collection of best strategic and operational security practices;

- *Threat profile* - the range of threats that a typical organization needs to consider;
- *Catalogue of vulnerabilities* - a collection of vulnerabilities based on existing platform and applications, for consultation regarding technical aspects that should be considered.

The OCTAVE method was designed as a complete process for large organizations. As is, it is not suitable for small organizations, which created a gap that was covered later by two other methodologies. These alternative methodologies are the OCTAVE-S and the OCTAVE Allegro, and are derived from the original OCTAVE method. While the OCTAVE-S methodology is just an adaptation of the original OCTAVE to smaller organizations (Alberts 2005), the OCTAVE Allegro has a slightly different approach, built up from the experience gathered with years of application of the original method (Caralli 2007).

The idea of the Allegro approach is that when information assets are the focus of the information security assessment, all other assets can be easily brought into the process as containers where information assets are stored, transported, or processed (Stevens 2005). In this sense, a container can be a person (since people can store information as knowledge, transport information by communicating, or process information by thinking and acting), an object (e.g. a piece of paper), or a technology (e.g. a database). Thus, threats to information assets are identified and examined through the consideration of where they *live*, which effectively limits the number and types of assets brought into the process. Moreover, focusing on information assets effectively limits the amount of information that must be gathered, processed, organized, analyzed, and understood to perform a risk assessment.

2.2.3 The Center for Internet Security benchmarks

The Center for Internet Security (CIS) is a non-profit organization formed by several well-known academic, commercial, and governmental entities that has created a series of *security configuration benchmark* documents (CIS 2008). The documents, which in some cases are accompanied with tools that verify the compliance with the configurations suggested, cover specific brands of several kinds of very popular software. Most of the software for which CIS benchmarks were developed are *fundamental pieces of software* that are the basis of most information systems in use today: operating systems, database management systems and network devices. Although fundamental, it is known that this kind of basic software is usually complex and does not come with good security configurations by default (Schweitzer 2006). Building information system's infrastructures over insecurely configured software results in systems that are

insecure in all their levels. Also, by being widespread, they are prime targets for attacks and knowledge regarding security vulnerabilities (especially coming from insecure default options) is very likely to spread fast.

These CIS benchmarks are developed and maintained by the public and private members of the organization. Building from personal experiences, each document is created through discussions and consensus regarding the most secure configuration options applicable. They are based on best practices for deployment, configuration, and operation in networked systems. In essence, each document contains explicitly all relevant security configuration options that are considered important in the usual environments that they are found. The configurations are divided in two different levels of security:

- **Level 1 – prudent minimum due care**. This is the set of configuration options that are considered the minimum level of security an organization should enforce. The suggestions are chosen to be simple, in a way that any system manager can understand and apply them, and are unlikely to cause any kind of disruption or degradation of the services they provide.
- **Level 2 - prudent security beyond the minimum level**. This is the set of configuration options that are necessary for systems demanding high security. Also, these configurations might cause impact in the operation of the system, so a system's manager with a reasonable level of security knowledge might be necessary to understand and apply them correctly.

Even though most of the benchmarks do not take into account the actual business rules of the environment where the software is being used, the approach from CIS has a significant number of advantages over other security evaluation approaches. One important characteristic of the approach is that it separates the security knowledge from the technical knowledge necessary to apply it, making the suggestions much more accessible than other methodologies. Another relevant advantage is that it is widely accepted, as the documents are the product of extensive analysis and consensus of several distinct representatives from public and private sectors. Also, as they are based in field experience, the threat model that supports them has the advantage of already being put in practice, being perhaps a form of validation of the security ideas behind it.

Despite the advantages, the CIS documents also have some noticeable drawbacks:

- The documents are not designed and written in a single standard way, and are actually overlapping in some areas. This implies that when more than one document is used in a single installation (e.g. hardening an operating

system and then the DBMS installed on it), difficulties might arise if similar things are stated in different forms in each document.

- Each document focuses specific software of a specific version. New versions of the software, even similar versions, cannot use the document without incurring in the risk of existing a significant difference that hinders the original settings as insecure;
- The documents are focused only on the specific configuration options available in the particular software being configured. In some cases, this causes that major security principles are not even mentioned. If the administrator is not warned that some important security control is missing from the software he is using, he cannot evaluate if it is important enough that he replaces it or implement the control in an alternative way;
- Even though some rationale is provided in some cases, the major security principles behind the choices are frequently not provided. Not mixing the security justifications with the actual configurations they provide is good from a practical sense, but the security principles behind them are necessary for several reasons: a) the administrator should be able to understand what are the risks he is facing when he is not able to comply with a recommendation (which might happen frequently in production environments); b) the administrator should be given the choice of coming out with alternative solutions to the security concern behind each suggestion (something he cannot do if he does not know what the suggestion's goal is).

Overall, the CIS approach is very interesting, very practical and is important in several ways. However, it is clear that there is room for improvement. In particular, because of some of these drawbacks, they cannot actually be considered representative benchmarks, as is discussed in Section 2.4.

2.2.4 Additional Security Evaluation and Risk Analysis Methodologies

While the Common Criteria standard presents a product oriented approach for security evaluation, the OCTAVE method appears as a self-evaluation process that takes in consideration as much aspects and particularities of the organization as possible. Even though they present complementary perspectives to security evaluation, several other frameworks, approaches and methods have been proposed and lie somewhere in between. Some relevant proposals that can be found in the literature:

- MEHARI (CLUSIF 2004): a risk management methodology developed by the CLUSIF (Club de la sécurité de l'Information Français) and built on

the top of two other methods: MARION and MELISA not maintained anymore.

- CRAMM (Siemens 2003): the *CCTA Risk Analysis and Management Method* is a risk management method from UK originally developed by CCTA3 in 1985 and currently maintained by Insight Consulting.
- CORAS (Vraalsen 2007): CORAS (*Risk Assessment of Security Critical Systems*) was a European project developing a tool-supported framework based on UML, exploiting methods for risk analysis and risk assessment of security critical systems.
- ISRAM (Karabacak 2005): methodology developed in December 2003 at the National Research Institute of Electronics and Cryptology and the Gebze Institute of Technology in Turkey. It is a survey-based model with a quantitative approach to risk analysis that allows for the participation of the manager and staff of the organization.
- NIST SP 800-30 (Stoneburner 2002): The *Risk Management Guide for Information Technology Systems* was developed by the National Institute of Standards and Technology as a recommendation for use by all federal agencies of the US. The process is subdivided in several steps: system characterization, threat identification, control analysis, likelihood determination, impact analysis, risk determination, control recommendations and result documentation. Like the OCTAVE method, a small knowledgebase of common threats is provided to help the assessment.

2.2.5 Security Characteristics Identification Techniques

The security evaluation frameworks previously described are essentially aimed at evaluating systems security from a high level perspective, including very large classes of threats simultaneously. However, security evaluation can also be done in smaller scales, looking for known kinds of security problems which when corrected may indeed increase the overall level of security, even if they cannot express by how much.

Static code analysis (Livshits 2005) is a technique where a program is used to analyze the source code of a program in order to find vulnerabilities in the source code. They usually are based on the search of coding patterns that normally can be attributed to vulnerabilities (Chess 2007). Several static code analysis tools are used in a series of experiments in Chapter 5.

Vulnerability scanners (Shahriar 2012) are programs designed to test systems against a list of known vulnerabilities, listing the ones that are found and therefore allowing them to be corrected. They are highly dependent on vulnerability

databases, and therefore their effectiveness depends on them being constantly updated. Vulnerability scanners are tools that can be employed as integrative parts in the implementation of our benchmarking framework.

Penetration testing tools or *fuzzers* (McClure 2009) are tools also designed for searching vulnerabilities. However, instead of being based on databases of known vulnerabilities, they interact with the system by submitting series of random or maliciously crafted input values in order to verify if the system has some kind of input validation failure. Most of these validation failures can be used to attack the system, and therefore can be considered vulnerabilities. Penetration testing is a technique that can also be done manually, in which a security expert will study and try to violate the input validation of the system (Long 2007). In this context, it is usually called *manual code inspection*.

A whole set of alternative techniques for identifying vulnerabilities also exist, ranging from direct attack injection approaches (Fonseca 2009, Antunes et al 2010), software testing (Antunes and Neves 2012) to robustness testing approaches (Saad-Khorchef 2007, Oliveira 2011). Most of these tools also can be used as components in our framework, and security benchmarking would not be possible without such capabilities. Nevertheless, their results and contributions in the context of benchmarking and selection have to be considered carefully.

2.3 Threat Modelling

Threat modeling is a technique that naturally appears as part of any kind of security and risk evaluation process, and started to take a formal shape in the last years. The idea behind threat modeling is to identify what are the potential threats against a particular scenario, and based on them determine what are the procedures or security controls necessary to mitigate these threats (Shahriar 2012). This kind of technique can be useful in the context of existing environments that must be further secured, but is especially useful when applied during the design phase of a system.

In most security evaluation methodologies, identifying threats is always posed as the process of *brainstorming* about the potential attacks and vulnerabilities that the system might be susceptible to. Although the formal approach to this task also requires some inventiveness to try to cover as much threats as possible, threat modeling is currently evolving in the direction of being a methodology that helps to exploit the threat space even further.

Formal approaches to threat modeling start to become considerably relevant with the STRIDE approach proposed in (Howard 2002) and supported by Microsoft (Swiderski 2004) as an important step to secure software design. STRIDE is

actually an acronym that stands for a threat classification method based on six different (possibly overlapping) ways of breaking the information properties. At the same time that it is used as a threat classification, it also forces the analyst to think about the ways that an attacker could implement each of the breakings. They are the following:

- **Spoofing –** threats that involve an entity using an identity that is not its own. Examples: stealing and using authentication information, pretending to be a legitimate part of the system and feed bogus information to another part.
- **Tampering –** threats that involve modifying data or another part of the system. Examples: modifying an unauthorized file or replacing the code of a particular function that is trusted (e.g. a DLL or an input validation function).
- **Repudiation** – threats involving the denial of performing an action. Examples: the exclusion of data and consequent denial of such action or denial of performing a digital signature.
- **Information disclosure** – threats involving the exposition of information to an unauthorized entity. Examples: reading other system user private files, eavesdropping the communication of a remote connection or reading the environment variables values of another operating system process.
- **Denial of service** – threats involving that a particular service becomes not available to its legitimate users. Examples: defacement of a remote web server, exceeding the processing capabilities of an application device or changing the authorization rights of the users of an application.
- **Elevation of privileges** – threats involving an entity obtaining more privileges than it was originally supposed to have. Examples: a regular user obtaining administrative rights or an application executing operating system commands.

Threat modeling in the STRIDE approach is performed as follows. First, it is necessary to identify the assets that must be protected. To protect the information properties within a system, it is necessary to protect the devices that carry the information, the mechanisms that transmit the information and the means that are used to access it (e.g. the network). For these to become more visible to the analysts, it is suggested that a *Data Flow Diagram* (Stevens 1974) of the system being analyzed is drawn (or alternatively an *UML deployment diagram* (Booch 2005)). Further documentation about the scenario or application being analyzed is always welcome, and the most decomposed it is, easier will be the analysis done over it.

With all data flows exposed, the analysis proceeds by trying to envision ways that each of the STRIDE threats can be applied to each of the data flows and elements involved, even the most improbable ones. When all threats are documented, before starting to address them, they are usually ranked regarding their overall risk, as to address first the most relevant ones. To do this, another acronym for five different aspects that can be related to a threat is used, called the DREAD score, which then allows computing an overall risk value for the threat. Each of the DREAD components is assigned a rating value ranging from 1 to 10, which extremes can be interpreted roughly as follows:

- **Damage Potential**: if an attack realized this threat, what is the consequent damage?
  - 1 = Disclosure of irrelevant information
  - 10 = Complete system and data destruction
- **Reliability**: does the exploitation of the threat always cause damage?
  - 1 = It will cause damage only under the most improbable conditions
  - 10 = It will always cause the most possible damage
- **Exploitability**: how easy is it for an attacker to exploit it?
  - 1 = It requires advanced programming and networking knowledge and physical access to a protected area of the organization.
  - 10 = Just a web browser and internet connection
- **Affected Users**: How many users potentially can be affected by it?
  - 1 = Just one user which mostly does not use the system
  - 10 = All users
- **Discoverability**: how easy is it to discover this threat?
  - 1 = Finding out about it requires knowledge of the inner workings of several closed source components and access to confidential parts of the system
  - 10 = The threat is available in public domain and fairly obvious

The overall risk of a threat is computed as the average of the scores of all components, and the most risky ones are considered first. After that, it is possible to analyze each threat and evaluate if there is already a mechanism in the system that prevents it from occurring or not. Any threat that does not have a mitigation mechanism is considered a vulnerability of the system, which can be severe or not. Evaluating if threats are or not already mitigated in a system can be a challenging task on its own.

Despite the method of analysis used, techniques that deal with threats usually fall within one of the following four different categories (Dorfman 2007):

- *Avoidance* – when a particular method is employed to completely eliminate the possibility of someone exploring the threat;
- *Reduction* – when the probability of exploring the threat is diminished instead of eliminated, what in some cases is the only alternative;
- *Transference* – when the risk is actually transferred for another party to solve. For instance, insurance is an example of transference of risk.
- *Retention* – the idea of simply accepting the risk and deal with the attack if and when it occurs.

Although the STRIDE approach does not explicitly provide formal ways to evaluate the threats and their mitigations individually, several alternatives exist in the literature. One popular method is using *attack trees*, which was suggested by Bruce Schneier in (Schneier 1999) and resembles the use of *fault trees* (Roberts 1981) for the analysis of system dependability.

The process of threat modeling using attack trees starts by the definition of a set of attack goals, which are considered the final objective of an attacker. An attack goal could be, for instance, reading an encrypted email, executing software in a particular remote machine or making a machine become unresponsive. The instantiation of the threats identified in a STRIDE approach might be considered as attack goals. The root of an attack tree is the attack goal, and the analysis start by identifying all methods that can be used by an attacker to accomplish the goal, which are state as children nodes. There might be several alternative ways of achieving goals or there may be necessary combined steps, which defines OR and AND nodes. A goal (or sub-goal in the case of a children node) is achieved if all AND nodes are achieved or if any OR node is achieved. The process continues recursively for the sub-goals, expanding the tree until the leafs are steps considered simple enough to be evaluated.

A complete and correctly designed attack tree can be used for several different security analyses. It shows all ways that an attack can be accomplished and particularly any path from the root node to a plausible leaf can be considered a vulnerability of the system. The tree also helps in the sense that the attack can be avoided at any step of the path, providing different mitigations strategies. An advantage of the method is that when a goal depends on an intermediary step that has several possible ways of being achieved, mitigating this particular intermediary step avoids several different vulnerabilities simultaneously.

Although complete, expressing attacks as trees might not be the most flexible approach, meaning that a complex attack tree can be very difficult to analyze. One way to allow the approach to be more flexible is to, instead of trees, use Petri nets (or place/transition nets) that are directed graphs used to model transitions with pre and post conditions, as suggested in (McDermott 2000). Coloured Petri nets, which use coloured nodes as an additional formalization expression, are also proposed as a way to extend the formalization even further (Helmer 2007).

Misuse cases (Alexander 2003) and *abuse cases* (McDermott 1999, 2001) are two other formal ways of expressing threats that can be useful to help understanding and identifying threats within a system. These methods, which are very similar with minor distinctions, are based on *use cases*, which are part of the UML language, being suitable to complement a system specification that already uses this language. Diallo in (Diallo 2006) presented and compared misuse and abuse cases with attack trees and the common criteria specification language, pointing out the advantages and disadvantages of formalizing threats in each of the approaches. Not surprisingly, this work shows that they are actually complementary, neither of them being the optimal solution for all perspectives.

In all approaches for threat modeling, despite the formality of each one, a significant amount of security knowledge is still required. This happens because it is always necessary some creativity to be able to identify all the possible ways the system can be attacked, and the most reliable way to achieve this creativity is through security experience. To help with this problem, another branch of investigation is becoming more and more popular, which is the study of *attack patterns* (Hoglund 2004). An attack pattern is an abstract mechanism for describing how a type of observed attack is executed and providing a description of the context where it is applicable. A formal study of repeating attack patterns used to break software was first presented in (Hoglund 2004), and clearly is an approach that can be applied to any kind of attack. Although fairly new, there is already a considerable amount of investigation regarding ways of expressing attack patterns (Pauli 2008) and using them (Gegick 2005, Gegick 2007).

The Common Attack Pattern Enumeration and Classification (CAPEC) sponsored by the Department of Homeland Security of the United States (National Cyber Security Division 2008) is an initiative that has as goal to try to build together with the community a comprehensive attack pattern database. The main idea behind the project is that such a database could be used to support any kind of security analysis process and evaluation, as it will provide an extensive attacker perspective (Barnum 2007).

Possibly the most comprehensive threat modeling approach already proposed is the Trike methodology (Saitta 2005). Trike is a framework for threat modeling built from the experiences gathered from all other methods that were already proposed. It is a formal approach designed to be complete and had two main goals as motivation. First, it is known that extensive threat modeling is a very long process that demands lots of documentation and careful analysis. Trike is designed to allow the automation of the biggest most portion of the process possible, meaning that the analysts can focus where it is really needed. Second, identifying all threats within a particular system usually demands very extensive security knowledge. By using a base attack library (provided by the framework), Trike defines a process to generate all possible threats against the described system in an automated manner, as to miss the least possible number of threats. To achieve both these goals, the Trike framework was proposed together with the implementation of a tool that implements the methodology, but this is still under alpha stage development (Saitta 2007).

Trike differs from other threat modeling technologies from a number of ways. Instead of using an attacker perspective, Trike models the threats from a defensive perspective meaning that instead of considering attacks, it considers actions that should not happen. The basic elements of Trike are actors, assets and actions. The analyst identifies and models the actions that the actors are supposed to do over the assets and from these modeling, two types of threats can be exhaustively enumerated: 1) all actions that are not supposed to happen are considered *elevation of privileges* threats and 2) actions prevented from happening are considered *denial of service* threats. This automatic threat generation is possible because the methodology is based on the principle that all actions can be decomposed in smaller actions that ultimately are "create", "read", "update" and "delete" actions over assets. This way, for a consistent description of the systems intended behavior, the complementary action space can be systematically identified, which cannot be done in other more *ad hoc* methodologies.

Trikes main advantages can also be considered its main disadvantages. To allow for automatic threat generation, the description of the system must be absolutely accurate, which can take a lot of work. Any missing details will cause either for the enumeration of non-existing threats or the failure of identifying important ones. Another problem is that the number of threats generated tends to grow exponentially with the number of assets within the system, which causes a serious scalability problem for analyzing complex systems. Also, threats involving elements outside the system boundaries are also missed in the algorithms, and must be considered in a traditional manner. For these and other reasons, the authors state

that the framework is still under development and advise care in the application of the methodology and of the supporting software.

2.4 Benchmarking

A computer benchmark is a standard procedure that allows assessing and comparing systems or components according to specific characteristics (Grey 1993). Historically, the most common goal of benchmarking of computer systems was the evaluation of performance. In particular, methods for evaluating the cost/performance trade-off were much required as new computer architectures and systems were being designed (Steen 1989). Nevertheless, the idea of assessing computers, software and processes in a way that allows comparison between different solutions can be applied to any aspect that can ultimately be labeled as "good" or "bad".

A useful characteristic of performance benchmarking is that it is easy to come up with quantitative *metrics* capable of expressing the speed *in which a system executes tasks*. When this is possible, a good/bad comparison can be trivially done, only by numbers comparison. However, not all aspects are easily translatable to quantitative metrics, and security is one example (Torgerson 2007). The problem is that, even though it is easy to picture a scenario that can be labeled unanimously as "very good" and another one that can be unanimously labeled "very bad", the ones in between are open for subjective interpretation. As an example, it is not rare to find magazines inventing benchmarks (e.g. performance, usability, security, etc.) and applying them to off-the-shelf software, authoritatively labeling them as good or bad. The problem is that this kind of benchmarking depends exclusively on the opinion of the evaluator and it is fairly easy to disagree with the results.

To be useful, a benchmark must be reliable in a sense that its methodology and results should not be open for alternative interpretations. In particular, Gray suggests that a good benchmark must meet four different criteria (Gray 1993):

- **Relevance** – it must be representative of the most typical operations within the problem domain. A benchmark that applies only too small subset of the problem domain is not useful as it allows limited comparison.
- **Portability** – being portable amplifies the benchmark usefulness by allowing comparison of a wider range of different systems and architectures.
- **Scalability** – the benchmark should be scalable in the sense that it should not depend on the size of the system being evaluated.

- **Simplicity** – the benchmark should be easy to understand, otherwise it will lack credibility.

Vieira states six properties for a useful benchmark, restating and complementing the previous four (Vieira 2005a). The rationale is that, if carefully validated, having these properties will more easily demonstrate the benchmark usefulness and allow its acceptance by a larger number of users. These properties are representativeness, portability, repeatability, scalability, non-intrusiveness and simplicity of use. Portability and scalability bears the same definition as in (Gray 1993) and representativeness can be understood exactly as relevance.

Repeatability is related to the ability of a benchmark to always produce the same overall results for the same system (in non deterministic systems, repeatability should be seen in statistical terms), no matter the number of times it is executed and by whom. This property is extremely important for the credibility of the benchmark; otherwise its results could always be disputed.

Non-intrusiveness is related to the quality of requiring minimum changes in the system being evaluated (or none at all). The reasoning is that if the benchmark process requires significant changes in the system, then one is not benchmarking the original system anymore, but rather the modified one. This property is a big concern for automated benchmarks because they usually imply installing and executing some benchmarking software. The software will inevitably consume system resources, and these should be taken into account in the results. The installation of this software should not require system modifications for the same reason.

At last, simplicity of use is related not only to the benchmark being easy to understand, but also easy to apply. A complex benchmark would never appeal to a large number of users, and therefore its usefulness would be compromised.

2.4.1 Performance Benchmarking

Benchmarking performance was historically so relevant that it is possible to find a large number of organizations proposing these types of benchmarks for several distinct domains. The Transaction Processing Performance Council (TPC 2012) is a consortium of vendors defining benchmarks for transaction processing and database domains. The System Performance Evaluation Cooperative (SPEC 2012) is a consortium that defines benchmarks for scientific and workstation domains. The Perfect Club (Cybenko 1990) is a consortium of vendors and universities that define benchmarks for the scientific domain, with particular emphasis on parallel or exotic computer architectures. The EuroBen group (Steen 1993) established a

series of benchmarks for the evaluation of high-performance scientific computers. The Parallel Benchmarking Working Group (Dunlop 1994), today the PARKBENCH committee, is a joint initiative for benchmarking parallel systems.

Performance benchmarks (including the ones previously mentioned) typically fit in a general profile that includes three particular components:

- *Workload* – a representative set of work that must be executed in the system being evaluated during the benchmark run. Work, in this sense, depends on what the benchmark is supposed to evaluate. In practice, the workload represents what would be required from the system in a typical real scenario, and the most representative it is the better.
- *Metrics* – a set of performance metrics that must be extracted from the system as to characterize the effect of the workload on it. The set of measures will depend on the kind of workload being executed and, most importantly, on what are the factors that the benchmark is designed to evaluate.
- *Procedures and rules* – the rules and procedures defining the steps that must be followed during the benchmark run. This set of rules establishes how the workload is executed, how the measures are collected and how the final benchmark results are computed. They must be clear, complete and unambiguous in order to allow the benchmark to be repeatable.

2.4.2 Dependability and Resilience Benchmarking

Although most performance benchmarks fit within the profile above, benchmarking other qualities of a system might require different approaches. The DBench project (Kanoun 2001; DBench 2000) was an initiative by several universities and organizations to develop *dependability benchmarks*. The justification for such project is that performance benchmarks are significant only in controlled environments, where the system suffers no adverse effects. Dependability benchmarks, on the other hand, would provide reliable indicatives of how a system degrades under the occurrence of faults and how is its capability to recover from them. Being able to evaluate systems from a dependability point-of-view is a very important because in the real world, faults are expected. For example, no one would choose a high performance system that simply crashes in the event of a simple fault. Thus, a way to reliably identify how different systems behave under the presence of the most common faults is extremely relevant.

A multitude of dependability benchmarks can be found in the literature for a very large diversity of domains (see (DBench 2000)), and a key characteristic of dependability benchmarking is the addition of a *faultload,* which represents the set

of typical faults that the systems in a particular domain are subjected to and a set of a dependability metrics, that aim at evaluating the degradation of the system performance and the efficiency of the dependability mechanisms. In Section 2.5.1 we present a deeper discussion about the way dependability benchmarking works.

With the evolution of computer systems, the dependability mechanisms they had also evolved and today we have the emergence of adaptation mechanisms (Almeida 2011). Basically, instead of simply coping with a set of faults, now the systems can adapt to a wider range of environmental changes in order to keep the performance as high as possible given any imposed conditions. The evolution of the dependability mechanisms again created another set of difficulties to benchmarking of systems in general because now measuring the performance degradation due to faults is not enough anymore, as the systems adapt to the imposed environmental stresses of all sorts a wider range of conditions that are not only limited to faults have to be considered, and particularly the ability of evaluating the overall efficiency of such adaptation mechanisms becomes a crucial problem, as we have to account for the degradation imposed by the same additional algorithms and modules required by them. In the literature, the concept of *faultload* evolves into the concept of *changeload* (Almeida 2012a, Almeida 2012b) that is designed to model all the stressful conditions that the system being evaluated will be subjected to under real conditions.

2.4.3 Security Benchmarking

A very initial attempt to devise a security benchmark that could hold up to scientific standards can be found in (Vieira 2005b). This work proposes a methodology for benchmarking the security mechanisms of database engines, which is done through a set of classes. The benchmark defines a set of tests that are used to characterize the mechanisms, and from the results of these tests a class is assigned to the engine. The test set is generic in the sense that any relational DBMS can be evaluated, and the approach is applied to two engines, Oracle 9i e PostgreSQL 7.3. Although very limited in scope, the approach appears to have everything that is required for a useful benchmark.

The security benchmarks proposed by CIS (presented in the Section 2.2.3), on the other hand lack several of the properties that are expected from a benchmark. Unlike the security benchmark of (Vieira 2005b), they are too specific for each version of the software for which they are designed. The problem is that, as benchmarks, their results are unreliable. First, even when a system follows all the configuration suggestions proposed, stating that it is more secure is problematic because security depends also on the way the system is used and on the

characteristics of the surrounding environment. Also, stating that a system is more secure because it follows more suggestions might be misleading, because sometimes some specific suggestions might have no influence on this particular environment. Moreover, all of these applications have security limitations, and these are never accounted for. Nevertheless, this is not to say that these suggestions are not useful, but that certainly means that they are hardly benchmarks.

The recently finished Amber project (Assessing, Measuring and Benchmarking Resilience) (FP-7 2010), funded by the European Union under the FP7 program, gathered the experience and expertise in benchmarking from an international group of researchers, and successfully raised awareness of the lack of security benchmarks proposals. In the Research Roadmap that resulted from this project (Bondavalli 2009) the authors identify several research goals and suggest a strategy aimed at eventually achieving research mass able to accomplish the definition of security benchmarks. Their proposal is based in the expansion of the extremely succesful model used for dependability benchmarking, in which fault injection techniques are used to evaluate the behavior of the system under faults (Bondavalli 2009). Their assumption is that devising a *representative attackload* and proper *security metrics* allow the specification of a security benchmark following the same approach.

The literature already shows a number of research works based on attackloads. In most cases, the main approach is to model attacks in a similar way to faults, using attack injection techniques in an attempt to evaluate security aspects of systems. In (Friginal 2011) the authors model a few attack techniques in order to complement the analysis of COTS under the specification of ISO/IEC 25045 standard. In (Friginal 2009, 2010) we find attack injection techniques being used to assess *ad hoc* networks. It is important to emphasize that such approaches are extremely useful and interesting, but are distant from the goal of a dedicated security benchmark that is capable of *measuring security level* of the evaluated system. Instead, the techniques obtain information about the dependability of the systems, the impact on performance of the system and of the security mechanisms and are also able to identify which systems can be breached and which cannot. However, selecting *the most secure system* is something that is extremely risky to do using only the results of such techniques, as we explore in the next section.

A much bolder attempt at actually measuring the security level of systems can be found in (Mendes 2011), where the authors take a database of known vulnerabilities and use it to rank the evaluated systems in terms of the risk that these vulnerabilities incur in the system. Although this approach is useful, and could be an integrative part of a security benchmark, selecting components based on this approach can also

be misleading, particularly because vulnerabilities can be patched, and after they are patched they give no real clue of the real security of the remaining system.

Attack injection (Antunes et al 2012, Fonseca 2009) and vulnerability finding (Shahriar 2012) are techniques that discover actual attack paths that can be used by attacker to harm systems, and this is important. However, we have to be extremely careful when interpreting what an existing vulnerability of attack path means to the security of a system, or else you incur in the real risk of expressing things that are actually not true, as we discuss in the next section.

2.5 Security Benchmarking as an Open Problem

Security benchmarking is still an open problem. Even though the community is clearly trying to move forward in the proposition of alternatives to devise solutions for this problem, the reality is that the path that security research is taking on this matter leads to several difficulties that will be extremely hard to overcome, due to the particularities of security that are not being taken into account. In the next sections we will discuss such difficulties, and begin the discussion of the measurement of trust, as an alternative to current approaches.

2.5.1 Dependability Benchmarking *vs* Security Benchmarking

The most common dependability benchmarking model in use today (Kanoun 2008), and which is slowly becoming an accepted standard as part of the TPC benchmarks (TPC 2012), is based on the definition of the following set of elements:

- A representative *workload*, which should represent the average stress and environment conditions that the system under test will be subjected to.
- A representative *faultload*, which includes typical faults that the system may face in the field.
- Performance and dependability *metrics*.
- *Guidelines* and procedures to run the benchmark and collect the metrics.

The dependability benchmarking model is built upon already established performance benchmarks, as discussed before. The transition is depicted in Figure 2.2. Typically, the benchmark execution is divided in two experiments: the *golden* run, where performance metrics are collected during the application of the workload, and a subsequent run where the system is subjected to the *faultload* concurrently with the workload (Kanoun 2001). Besides collecting dependability metrics relative to the fault tolerance of the system, the main goal of the second run is to obtain performance metrics under faulty conditions, which, when compared

with the performance during the *golden* run, allow the evaluation of the overall system degradation.

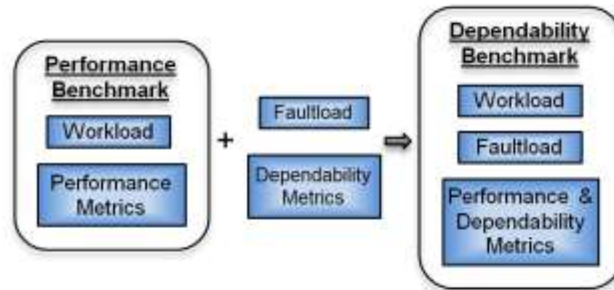


Figure 2.2 Dependability *vs* performance benchmarking

Possibly driven by the undisputed success of the dependability benchmarking model, the scientific community has shown a general feeling that such model could be successfully expanded and applied to the security field. For example, the Amber research roadmap (Bondavalli 2009) makes the following suggestions as short-term goals (should be accomplished in 3 years' time frame):

> *"<u>Reference attackloads and injection tools to be used in the development of security benchmarks</u>: Finding whether representative types of attack patterns and security vulnerabilities exist through field studies and analysis of information available; Definition and validation of reference attackloads for different security benchmarking domains and classes of targets; Development of tools to inject reference attackloads in different classes of benchmark target systems."(Bondavalli 2009)*

It is clear that the Amber consortium feels that the dependability benchmarking model may work for security benchmarking, as long as representative attackloads are defined (in the same lines as representative faultloads), and that the community is able to design a representative set of security metrics that allow characterizing the system regarding its ability to prevent the attacks (or their effects) contained in the attackload.

Assuming that there exists a set of security metrics with the above capabilities, the problem with the approach begins with the definition of what is a *representative attackload*. Obtaining a representative faultload is already a very complex problem (Arlat 2002). For example, should we consider a flooded room as a representative "fault"? It surely depends on where the system is, and how critical is the service it provides. But assuredly a faultload that does not include a flood as a potential fault

(despite the domain benchmarked) will not considerably hinder the benchmark representativeness. The fact is that every single fault included in the benchmark cumulatively renders it more representative, and any potential fault to which the system is tested against provides valuable information to the system owner. We may say that there is some *fuzziness* in the border dividing a representative faultload from an unrepresentative one, and that fuzziness does not have to be fully cleared for the benchmark to be useful.

In the security domain, things are more complex. Suppose, for instance, that it is possible to determine a representative *attackload* for a particular domain. Using the dependability benchmark model enhanced with an attackload and security metrics, we apply the benchmark to choose the most secure of two systems. After evaluating the behavior of the two systems subjected to the attackload, any conceivable set of security metrics is expected to provide, at the very least, one kind of security information (even if able to express more): *either the systems are completely immune to the attackload, or the systems are breached*. What can we learn from each of these results? If a system is breached, the disclosure of the report of the benchmark run would make that information public, and it would become available to anyone with knowledge of the attackload. No sane person would choose a system that has a known security vulnerability, particularly because knowing that the system can be attacked is equivalent of saying that the system is essentially insecure. If confidentiality was lost, integrity was lost or the system became unavailable (to limit our discussion to these basic security properties) then the system was successfully breached and attributing a “level” of security when one cannot maintain the security properties makes no sense. On the other hand, if the properties were not breached, then the attack was not successful, and metrics that represent the degradation of the system due to the interference of the attacks are either performance or dependability metrics, *but they are not security metrics*, and this is true even if what we are measuring is the degradation of the security mechanisms themselves. In the end, if both systems are vulnerable, the benchmark user is left with little options, even if one of them is “slightly less attackable”, whatever that may mean. But then, what if both systems are immune to the attackload? Are both systems 100% secure, or the attackload is unrepresentative? Which is more likely? In fact, both answers have limited usefulness.

The previous discussion assumes that it is possible to find a representative attackload. But when security is the issue, the dimness between a representative attackload and an unrepresentative one may not be tolerable. A single missed attack is enough to turn the most secure system in the world into the easiest one to break. Furthermore, a representative attackload would need to take into consideration the

attacker's perspective, and thus the capabilities of most probable attackers (at least the technical capabilities, even if we ignore the financial ones), which usually are impossible to predict beforehand. Predicting how people will think and act in the future is simply too complex, and odds are that *the exact achievement of a representative attackload automatically renders it unrepresentative*.

Perspectives regarding the set of security metrics are also not promising. In (Littlewood 1993) the authors proposed trying to measure *the effort-to-breach* a system, which would appear interesting when allied with a representative attackload. This kind of metric assumes that there is a value that varies between *zero effort* and a *full breach effort*, and would be somewhat useful to administrators and developers. The idea was likely borrowed from the cryptographic community, where encryption algorithms strength is evaluated based on the effort that the attacker has to do to break it. What is generally missed is that at any time, and given a particular technological situation, any cryptographic algorithm is assigned by the research community a binary status: either it is broken or it is not broken. This holds even for algorithms that have theoretical shortcomings that allow finding its solution faster than brute force, as the encryption algorithm AES (Nikolić 2009) (which has some theoretical shortcomings, but is not broken), and the cryptographic hash function SHA1 (Wang 2005) (which is considered broken even though no real break was computed yet).

From a benchmarking perspective, the most information you can get from an attack - and therefore an attackload - is *whether it works or not*, which amounts to the fact that the system has a vulnerability that is not covered by a compensating defense mechanism. In other words, if a target is submitted to an attack that is successful, the most important usable information that you get is the fact that it is vulnerable to this attack. In a benchmarking context, whenever a benchmarked target is found to be susceptible to an attack, the likely procedure will be to correct or compensate the vulnerability, something that in the end will alter the benchmarked target. So, any security metric based on the amount of attacks that are successful is *misleading* because the actual system that will be used in the field will be the corrected version of the benchmarked target, and not the flawed one. If we assume that we could fix one system, then we have to assume that we could fix all benchmarked targets. The problem is that, now, we end up with a set of systems that is resistant to all attacks contained in our attackload, and the metric will result in the same value for all these targets, leaving the problem of security comparison unsolved. Moreover, attack effects will vary depending on the system usage and goal, and even if we could measure these, they most likely have no relation to the probability of the system being vulnerable to that specific attack, so they will not help solving the problem.

In contrast, this model works for dependability benchmarking because the injection of faults affords an opportunity to gain knowledge on the behavior of the system if and when those faults occur in the field and, if faults occur, the metrics will characterize their overall effects. For instance, the benchmark characterizes what happens after a disk effectively fails. In security, the goal is more to identify if we are secure during future attack events, and less to minimize the amount of damage resulting from a known successful attack. Comparing to dependability benchmarking, it would be like trying to understand the ability of the system in never allowing a disk to fail in the future. We know that disks will fail, and there is no correction/improvement on the system that may prevent, ever, a disk from failing. Vulnerabilities, on the other hand, when found *should* be corrected or circumvented.

In other words, the dependability benchmarking model does not seem to be the most adequate for security, mostly because the goal of a useful security benchmark is slightly different from the one of a dependability benchmark. Although the knowledge of known attacks that are able to breach a given system is extremely valuable, allowing to correct flawed systems, this information does not help to select between candidates because if we allow all the candidates to be corrected according to the knowledge our attacks, we end up with several systems that measure equally concerning attackload based metrics. In the end, we are still left with the problem of choosing the system that will behave better when subjected to active, ingenious and malicious minds that have beforehand the entire knowledge about any existing security benchmark. To solve this, we need procedures and metrics that are not based only on known attacks and vulnerabilities, but that relate to the probability of the existence of unknown vulnerabilities and the ability of the system to resist to unknown attacks.

2.5.2 Benchmarking Trust

It is interesting that back in 1993, Littlewood (Littlewood 1993) cites the Orange Book (DoD 1985) levels as "*represent(ing) levels of trust, an unquantified belief about security*", toning it as a downside of the approach, while at the same time proposing *effort-to-breach* as a useful quantifiable metric. Although the Orange Book levels are far away from being useful for benchmarking and comparison, maybe they were more on the right track than realized.

The security community already noticed that the words *trust* and *security* have been more and more used interchangeably (Marsh 2005), as if a trusted system was a secure system, and security necessarily implied trust. This is a problem, as this use

of terminology is mixing up concepts that are necessarily different and actually complementary.

A secure state is the state of "not being able to be attacked" or "not being vulnerable". Although it is possible to come up with "levels" of how vulnerable the system is (i.e. levels of security), the definition of each of these levels is likely to be static, and the state will either be one or the other. The fact that a certain attack is possible effectively means one is less secure, even though it is hard to include the notion of future unknown attacks and unknown vulnerabilities in the concept of a definitive security state.

Trust, which is an assumed reliance on something or someone (McKnight 2006, Sullivan 2010), on the other hand, can be thought of as a continuous concept, which can be increased and decreased in a variable amount, depending on the circumstances and events. This makes it *automatically suitable as a metric for comparison* and, therefore, for benchmarking. Also, the work done in trust quantification is far ahead than the research in security metrics, and approaches for measuring trust can be already found in the literature (Ray 2004). We believe that trust is a concept that more naturally accommodates probabilities and uncertainties related to unknown factors.

Using a real life analogy, we happen to trust more someone the more evidence he/she provides that he/she can be trusted. Also, the degree of increased trust varies with each situation. For instance, if you pass by a person in the street and he does not steal from you, your trust in that person may increase a little bit. If the same person saves your life from being run over by a car, it may increase more. Both levels of trust are useful, as in one case it gives you the liberty of not being particularly afraid in a subsequent encounter, and in the other you may consider depending your life upon the person. Nevertheless, in neither case this trust guarantees that the person will not hit you with an axe when you turn your back away a next time. Notice that trust allows one to make *informed decisions*, but without providing any *guarantees*. This is probably the main concern of (Littlewood 1993) when considering trust levels as a downside of the Orange Book. However, the security community is already comfortable with the idea that there is no 100% secure system, so this may not be an unbearable problem if this issue is dealt with correctly, which is something that our benchmarking framework does.

Regarding benchmarking, when we shift the focus from measuring security to measuring trustworthiness, several differences are evident. The most important and controversial one is exactly the fact that trust does not necessarily imply security, even though it may suggest it. So how could it be considered an alternative? First,

it is unlikely that we will ever be able to provide definitive guarantees against future unknown attacks. Also, vulnerabilities are things that are not definitive, and the simple advancement of security knowledge creates vulnerabilities where before there was none. Basically, certain characteristics start being vulnerabilities once someone finds out how to exploit them in an attack scenario. In this sense, it may be impossible to have more than trust in our systems. Second, and more important, aside from guarantees, a trustworthiness benchmark accomplishes all goals that would be required from a security benchmark in an easier manner. On average, a more trustworthy system will be more robust to attacks and less likely to be attacked than a less trustworthy one. Much like in dependability benchmarking, averages are the best possible predictions we can make about the future conditions under which the system will operate.

By accepting these fundamental limitations in security evaluation, we find out that a trustworthiness metric should be based on the *amount of evidence available that the system is secure.* More evidence of security mechanisms, processes, configurations, procedures and behaviors (we may call each of these *security elements*) results in a more trustworthy system. Also, the more widespread or more narrow the protection provided by the existing elements, the more the degree of trustworthiness varies. In a way, we would be measuring how wide the umbrella of security elements of the system is, or its *defensive surface* (in contrast to the *attack surface* concept (Manadhata 2007) suggested by an attackload). The larger the umbrella, the less likely there is a hole somewhere, and more trust one can justifiably put in the system.

One important characteristic of this approach is that the amount of trustworthiness of each security element has to be correct only in a relative way (i.e. the exact values are unimportant). For each security element, trustworthiness may be added to the system in the amount of known attack paths that it covers. It may also be weighted by its own constituent trustworthiness (e.g. *does the element usually works as expected and has proven to prevent real attacks?*). Here, probability of failure of security elements may play a part, and a whole lot of ideas can be incorporated to the concept, which is more deeply discussed in the next chapter.

2.6 Conclusion

This chapter presented an overview of the state of the art of several topics related with the rest of the thesis, ranging from security evaluation frameworks and methodologies to the state of the art and the evolution of benchmarking. We devoted particular attention to the reasons why current approaches to dependability

benchmarking do not fit the requirements of security benchmarking, which was the main motivation for the framework we propose in the next chapter.

The security evaluation techniques covered are closely related with benchmarking, as they are also methods for assessing high level security aspects, and may be used in situations where benchmarking is not completely necessary (or not applicable). The chapter focused on three relevant methodologies, which can be viewed as complementary ways for evaluating security: the Common Criteria, the OCTAVE method and the Center for Internet Security Benchmarking approach. Two of those contributed to some aspects of our benchmark implementations (namely the Common Criteria and the CIS benchmarks). Other complementary security evaluation methodologies and frameworks are based on variations of these were mentioned for completeness. A few specific techniques, also important to our work, were presented. These techniques are not full security evaluation frameworks, but are used to evaluate more specific security aspects and play important roles in security benchmarking, such as vulnerability finding techniques, static code analysis and penetration testing.

Another key topic covered was threat modeling. Even though we do not apply directly any specific threat modeling technique in our work, we do partially include one in the process of creating a list of threats for transactional systems infrastructures (presented in Chapter 4).

As the goal is to provide a security benchmarking framework, this chapter also included a detailed discussion on benchmarking topics. We described what is traditionally expected from a benchmark, including some hints related to the evolution of the concept over the years. Essentially, benchmarking as a scientific research topic started with the goal of evaluating and comparing performance. However, in the last decade, the concept evolved towards the evaluation of dependability attributes of computer systems. This work appears exactly in a moment where the research community is beginning to extrapolate the methodologies, lessons and achievements from dependability benchmarking research to other aspects, including security. However, as discussed, the problem of benchmarking security is quite different from benchmarking dependability attributes. Finally, the chapter discussed the concept of benchmarking trust, namely on how it can be related with security aspects, which is an idea that we explore extensively in our framework.

3

A Framework for Security Benchmarking

The set of *metrics* is the central and indispensable component of a benchmark. Conceiving a security benchmark would be a trivial problem if the definition and collection of security metrics were easy tasks, which is not the case. In fact, Enterprise-Level Security Metrics were included in the *2005 Hard Problems List* prepared by the INFOSEC Research Council, which identifies key research problems related to information security (INFOSEC 2005). Although there are many proposals of security metrics for computer systems, so far no *consensual* general security metric has been defined (Jansen 2009).

Ideally, a security metric should portray the degree to which security goals are met in the *System Under Test* (SUT) (Payne 2006). The expectation is that the comparison of the result of measurements performed on two distinct systems – or the same system in distinct states or circumstances – provides enough security information to allow the system administrator/owner to make informed decisions regarding the selection of alternatives or necessary improvements. Furthermore, although the exact kind of output we expect from a security benchmark depends on the goals of the SUT and on the context in which it is (or will be) used, that output should always include information about the kind of security problems the system may have, and should allow the identification of the parts of the system that are more prone to security breaches (and therefore deserve more attention).

One of the biggest difficulties in designing such a generic security metric is related to the fact that the security level of a system is highly dependent on what is *unknown about the system* (Torgerson 2007). For example, vulnerabilities that exist in an application, but that nonetheless are not perceived by the developer/administrator, are the ones that (ideally) should influence the security metric the most; otherwise the metric will be of reduced usefulness, as decisions based on it will not take those vulnerabilities into account, thus leading to erroneous or misleading conclusions.

This issue becomes even more challenging when we consider complex scenarios, with many devices, software and people involved, and where security vulnerabilities may exist not only because of faulty elements, but also due to the combination of the characteristics of these elements, including the environment around and the existing interactions (e.g. a database accessed by several applications and users). Given these factors, it is extremely hard to devise a numeric value that correctly expresses the actual *security level* of a computer system in a way that allows making meaningful and safe comparisons.

Insecurity metrics based on risk try to cope with the uncertainty associated with measuring the security level of a system by incorporating the *probability of attacks* (Jelen 1998). Risk is usually defined as the product of the likelihood of an attack and the damage expected if it happens. In principle, this metric can be used to decide if the hazards to which the system is exposed are acceptable or not, and also to help selecting the ones that should be mitigated first. The problem with this approach, in addition to the already hard problem of compiling an exhaustive enough list of possible attacks, is that it is very easy to underestimate or overestimate the two values (the probability and the damage), exactly for the same reasons that a general security metric is hard to define and compute: again, these values are highly dependent on what is unknown about the system This is, obviously, a major problem when risks are used for supporting security related decisions.

An additional problem of risk-based assessments is the fact that they rely too much on external information. Basically, the probability of attacks is directly related with "*the probability of an external agent having some interest in attacking the system to begin with*", and the potential damage is biased by the possible interests of the attacker, which certainly varies wildly. Even if one manages to get accurate values in a certain point in time, the context evolves and changes depending on factors that have absolutely *nothing* to do with the system that is being assessed (Grey 1993).

In essence, traditional security and insecurity metrics are hard to define and compute (Torgerson, 2007), as they involve making isolated estimations about the ability of an unknown individual (e.g. a hacker) to discover and maliciously exploit an unknown system characteristic (e.g. a vulnerability). Moreover, these metrics are often expected to depend only on information about the system itself, while at the same time incorporating the capabilities, behaviors and intentions of potential attackers, as if the information about the system could be enough to define the behavior of a potential attacker. In other words, this perspective starts from the assumption that a security metric *can* be made universal, in the sense that it will have the same value when seen from different perspectives (e.g. the administrators' versus the attackers' points of view). This will never be true as it is virtually impossible to know all attackers' capabilities, and the number of ways a system can interact with its environment is practically infinite. We start the definition of our framework by assuming that this approach is unfeasible, and therefore we have to redefine the whole idea of benchmarking when it comes to security aspects.

When pondering over security benchmarking, we have to be careful to never lose sight of some fundamental aspects. One of those aspects is that we do not want the portrayed level of security to vary depending on external variables, or else two distinct measurements will not be comparable. To illustrate how easy it is to miss this point let's discuss the case of two "common" incident metrics found in organizations, and that are very frequently misinterpreted as "security metrics". One we call NVD, the *number of viruses detected* in all computers of an organization, and the other is NSD, the *number of spams/phishing detected* in the overall bulk of email that circulates in the network (Kumaraguru 2007). Let's assume that these numbers are collected with some predefined periodicity that allows us to compare two measures separated by one period (e.g. one month).

NVD and NSD are interesting administrative metrics that can be used in practice to help in the security activities of an organization. For instance, if NVD or NSD numbers are high, this may lead to the decision of buying or implementing more security precautions against spam and viruses, allocating money for that task. In this case, such decision is justified by the simple fact that the number of incidents is high. In general, thresholds can be defined and used to raise awareness within the organization, in order to improve the attention of the employees to the problem and help find and mitigate potential causes. For benchmarking purposes, however, those numbers can be extremely misleading. To understand why, we have to consider the two main goals of security benchmarking: *self-comparison over time* (to evaluate improvement or degradation) and *comparison of distinct software (*for selecting the best alternatives).

Starting from the self-comparison goal, a key question can be stated as follows: *if NVD and NSD rise dramatically over time, is the security of the organization getting worse in any sense*? The answer is that it depends on why the numbers raise, which sometimes is not easy to know. In some situations, a simple rule modification or antivirus definitions update may trigger the detection of several infections that were already there, but were previously unknown (meaning that the overall security situation is improving, as the viruses that were there are now being eliminated). It may also be the case that targeted attacks are occurring at the present moment, and they are successfully being identified and blocked by the filters and antivirus. In this case, we may say that the situation is getting worse, in the sense that the organization is being attacked, but on the other hand it is good to verify that the tools are working as they are supposed to (even though we have no idea if they are solving the problem completely).

This reasoning can get even trickier. Suppose that NVD raises and NSD keeps stable: this would probably turn the administrator attention to the antivirus, trying to understand why the metric changed. But this would lead nowhere if the case was, for example, that the users inside the organization were being victims of phishing attacks (e.g. clicking in malicious links in emails that were not caught by the spam filter, and infecting the machines with viruses). In such situation the problem would have nothing to do with the antivirus, but with the spam filter and with the lack of understanding of the employees about the problem. Alternatively, we may see both numbers going down. What could be the course of action in that case? Could it be because the security countermeasures lost effectiveness, or because the number of attacks just decreased? Should one be concerned or reassured if the number of viruses detected suddenly decreases by 50%?

The main conclusion that has to be drawn from this illustrative discussion is that such numbers express information that can never be used to understand the status of the security level of the organization. Even though they portray some relation between the security level of the organization and the events that are occurring in real time (attacks, or lack of attacks), *it is not possible to extrapolate the actual security level from this relation*.

Considering now the goal of comparison of software alternatives, the usefulness of such numbers for ranking is even worse. If an organization changes an antivirus or anti-spam solution to an alternative one, and the numbers for NSD and NVD go up, does that mean that the new solution is better? Again, following the same type of reasoning, we can conclude that those new solutions may very well be worse than the old ones, and that it is impossible to justifiably and confidently decide either way based on the values for NVD and NVD. It is important to remember that the

pattern of change of such numbers strongly depends on external factors that cannot be controlled. In practice, this kind of metrics cannot be used for benchmarking, as they can be significantly misleading.

Security benchmarking must be a process that **consistently and systematically identifies the actual security characteristics of the evaluated targets despite environmental influences**, and conclusions must not vary for a single target even in the presence of new attacks or attackers, or this may invalidate the measurements. One key mantra that should not be forgotten is that *we are measuring the system, not the attackers.* In fact, whenever new attacks become relevant to the point of making a benchmark invalid, the solution is to define a new benchmark specification, and deem the old one as obsolete. As far as possible, under the same benchmarking specification, the security assessment of a target should be *deterministic* and not change with time or due to variations on the attackers' capabilities. Given all these restrictions, it becomes understandable why security benchmarking is an extremely hard problem and why no effective model has been proposed so far.

Another key aspect that needs to be emphasized is that security benchmarking will never be able to express more about security knowledge than what the current body of knowledge on security can provide. People should not expect security benchmarking to miraculously bring forth information that was invisible to everyone beforehand. In other words, security benchmarking should be perceived as a procedure able to extract, analyze, organize and summarize information related to the security level of a benchmarked target in such a way that this information can be used confidently for relative comparison and decision-making. From this perspective, the security characteristics of the assessed target are much more relevant than the capabilities of the attackers, which will serve only as a *frame of reference* for the threats that systems are expected to be protected from. One key idea that we try to convey in this work is that **in security benchmarking we should model the attackers' capabilities as the *effects* that they may cause in the system, independently of their actual capabilities or intents**.

The outline of this chapter is as follows. Section 3.1 we discuss the idea *o threat vectors* and what they are a good starting point for trustworthiness benchmarking. Section 3.2 we present our benchmarking framework. Section 3.3 we present the system that will be used as a case study of our framework. Section 3.4 concludes the chapter.

3.1 Threat Vectors as Basis for Benchmarking Security

The main reason why computer security is important is the existence of threats. If there were no threats, we would not have to be concerned with security. Therefore, in a way, threats are the component that drives almost all security analysis approaches (Schmidt 2010).

Even though we all understand the idea intuitively, in security research works the term “threat” (Im 2005) is frequently associated with different formal definitions. Particularly, the exact concepts that have to be present for a specific threat to be defined vary from one author to another. A commonly used definition is that a threat is the specification of *whom*, *how* and *in what circumstance* a given *action* will accomplish some *undesirable effect (result)* (Johnston 2010). For instance, a threat defined this way could be stated as follows:

Terrorists may detonate a bomb in a bus causing it to explode.

Improving the security of a scenario where this threat is assumed to be possible would require implementing measures that prevent it from being accomplished whenever there is an attempt. Notice that the threat specification already contains a lot of information. For example, the attackers are terrorists, not college students. They will use a bomb, not a missile or a biological weapon, and the event would involve a bus. Such definition also allows us to quickly understand the intended effects of the attack attempt. Even though the immediate effect is that people on the bus will die or be hurt, the main goal is to cause panic, first in the region where the explosion happens and then in the general population (relying on the helping hand of the automatic media exposure). The final goal is to cause general fear and, ideally, mass panic and a variety of damages in all levels of society.

A way to improve security on this scenario would be to raise the awareness of the people that use buses for transportation, and to investigate manually suspicious buses and abandoned packages (if possible, without disregarding the side effects of such measures, such as the hindrance and delay imposed by such procedures). To consider a more general approach and broaden the security measures needed, we can change a few of the elements in the definition: for instance, let’s assume that also taxis may explode, and that college students and old ladies may also be recruited by terrorists. This clearly shows that the number of possible threats may increase exponentially if several such variations of the elements of the initial threat are considered, making the goal of “preventing all threats” impossible to achieve.

Computer systems are extremely complex, and exactly due to that, they can be attacked from an almost infinite number of angles with different approaches, causing a myriad of distinct effects (Chapman 2011). When we generically talk about the security of a computer system, we want to be broad, and therefore should include all those angles simultaneously. However, exactly like in the bombing threat we discussed previously, it is not feasible to enumerate all the possible threats we have to take into attention when securing the system, and as the systems evolve, so do the techniques used to accomplish the attacks. Our goal, however, does not change: we want to reduce the probability of the system being successfully attacked considering the set of all possible ways to do that. Theoretically, security benchmarking should help in driving the system modifications in a way that improves the probability of successfully stopping any possible attack. The key question is: *how do we even start achieving such goal?*

In an evaluation context like security benchmarking, when we look at threats like the preceding example (i.e. the terrorist attack), it is not hard to notice that too many elements are fixed, and that this is not an adequate approach if the goal is to be broad. For instance, if we focus on *buses*, we are forgetting about *trucks* and *cars*. If we focus on *bombs* we are not considering *biological weapons*. If we are going to vouch that something is more secure than another, we better do it taking the widest angle possible, or else our assertion may be wrong in a huge number of scenarios. Furthermore, the “who/how/when” of attacks in computer contexts varies so much and changes so fast that we believe it makes little sense to try to focus on specific details of these variables.

Another important aspect that cannot be forgotten is that accomplishing security benchmarking requires considering only the characteristics of the system in the assessment, avoiding the dependency on external factors. So, although the benchmark driver is the concern of preventing external threats, what we should look at and take into consideration are the *characteristics of the system*, and not the characteristics of the attacker or the attack itself. In fact, as these are the elements over which we can act (we cannot change the attackers; we can only change the system), we have to consider threats from an alternative perspective.

In this work, *threat vectors* are defined as sets of characteristics of a system that are related to threats that accomplish certain specific effects. In the example above, *mass panic* would be a threat vector, which would be defined as *the set of characteristics of the environment that lead to an increased probability of the occurrence of mass panic*. In this case, we could extrapolate that certain agglomerations of people do favor the creation of mass panic, even if this is not the only requirement. The goal is to help discarding the information regarding specific

attackers and attacks and to focus on the characteristics of the system that have some relation with the probability that certain bad effects may occur. Also, while focusing on the effects, it becomes easier to identify alternative attack situations that may not be obvious from the start. For instance, could *sound based weapons* be effective to cause mass panic situations? What would be the precautions required in that case? More importantly, if we are concerned with panic, then we are open to techniques that act on the people that may suffer from that panic, fighting the effect instead of the cause.

This definition of threat vector widens the way we look to security aspects, while at the same time maintains the focus on the system instead of on the attackers. As we are looking at *systems' characteristics* that have to do with the possibility of certain *bad effects*, we can then aggregate these characteristics and translate them into probabilities of the effects being accomplished even without taking into account the attacker's related details. Note that, we use the expression *bad effects* instead of, alternatively, *malicious effects*, as the later would usually assume intentions behind them. As we are focusing on the system, it is not necessary to consider someone with any kind of intention; what we are concerned with is that the *effects,* which by definition are unwanted, do not manifest themselves.

The main challenge, therefore, is to determine, for a given domain, what are the threat vectors that are important to consider and, more importantly, what are the systems' characteristics involved in accomplishing the related effects. In our framework, this definition is what provides, in the form of trustworthiness benchmarking metrics, comparison capabilities to a security benchmark.

3.2 Security Benchmarking Framework

The assumption that *security has a lot to do with what we do not known about the system* requires us to investigate how to include in a comparison framework (i.e. a benchmark) information about what we know and about what we do not know about a system. In the course of our research, we came to the conclusion that the most effective way to correctly tackle this problem is by explicitly separating the benchmark in two parts: first, the benchmark should evaluate the explicit security mechanisms and visible defects that the system has, and second, it should assess the possibility of the system still having unknown security problems. This way, the proposed security benchmarking framework requires two distinct evaluations to be carried out, namely: *security qualification* and *trustworthiness benchmarking* (see Figure 3.1).

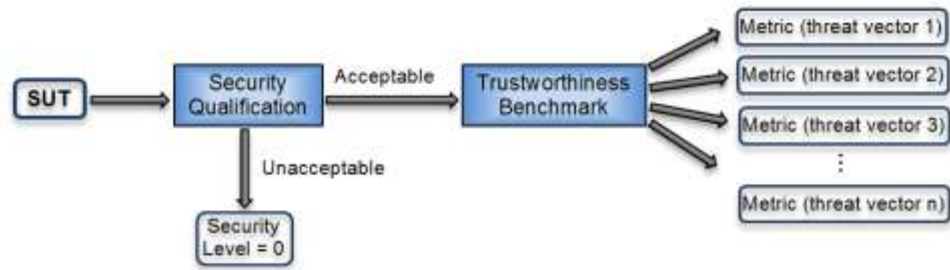


Figure 3.1 High level vision of the benchmarking process

Security qualification is related to the actual, tangible characteristics of the system, and their effectiveness on complying with a pre-defined level of security specified for a given application domain (i.e. the domain of the systems to which the benchmark should apply). Today, most domains have a minimum level of security that is required so that a system can be considered *acceptable*. For instance, the minimum absolute level of security that we would expect for a car is that it must require a key to be opened and to be turned on, and that the key would be only in the possession of the owner of the car. A car without a key would not be acceptable for most people, as that car could be easily vandalized or stolen. The same reasoning can be made for a bank account that does not require any authentication protocol for withdrawals. These examples (one car without keys, a bank account without authentication) are simply not acceptable for use in most domains, and a security benchmark would fail completely if it did not take into consideration these types of requirements.

In computer systems, a qualification step of a security benchmark could require the software being benchmarked to not have any obvious vulnerabilities detectable by static code analysis tools or penetration testing tools (or both) and/or to have a certain type of construction pattern (e.g. it could require the application to employ specific algorithms, libraries or access methods in its programming). Another possibility for the qualification step would be to require the system to provide certain configuration options or security mechanisms (e.g. encryption capabilities, enforcement of certain policies or specific methods of authentication and access controls). These aspects are domain dependent and *qualificatory*, in the sense that a system is not considered acceptable for use if it fails these requirements.

An example of a qualification requirement for an *operating system security benchmark* could be as follows: *the system is disqualified if it does not ask for authentication before allowing any kind of user interaction*. This requirement is quite intuitive and it is very easy to imagine situations where this is a fundamental

security requirement for an operating system (e.g. the operating system used in private workstation in an organization). At the same time, we can also find several other situations where this is clearly not a requirement (e.g. a public kiosk designed to show repeating slides), and therefore, a security benchmark for such case would not include a qualification requirement like this. The details and justifications that lead to the inclusion or not of each requirement are part of the definition of the domain, which is indeed a crucial part of the benchmark. The issues regarding the definition of the domain are presented in more detail in Section 3.3.3, when discussing some aspects related to the instantiation of the framework. For now, we may understand a domain as a particular *use-case* of some class of applications (e.g. *operating systems for typical desktop home-users*, and *operating systems for web servers* are two examples of use-cases for operating systems).

Notice that the simple existence of an authentication mechanism in a qualified operating system provides very little information on how reliable that mechanism is; the qualification is simply stating that a system is not acceptable if it does not have it at all (i.e. at the very least, the mechanism *must* work and not allow an unauthenticated person to interact with the operating system easily). Other possible qualification requirements could be: to require the operating system software to not present any vulnerability during an automated source code analysis (possibly using a specific tool defined by the benchmark), or to require the operating system to employ one of a specific set of authentication protocols.

In a general perspective, security qualification comes from the observation that it makes little sense to assign a security level to a system that has obvious ways of being attacked (be it due to the inexistence of a security mechanism or due to the existence of an obvious vulnerability). The main assumption is that, if one knows how to successfully attack the system, then the security is defined as zero and the SUT fails (i.e. is not acceptable for use). Obviously, the details and specificities of the qualification step depend not only on the particular application domain as discussed above, but also on how effective the benchmark will require the targets to be. For instance, the qualification could require the SUT to implement a two-factor authentication by default, or, alternatively, express the existence of a simple pre-shared key setting to be enough. A more detailed discussion on security qualification is presented in Section 3.2.1.

The systems that pass the first step are considered equally secure up to this point, and are therefore assigned for *trustworthiness benchmarking*, which is a *quantitative evaluation* that allows some kind of security comparison. The trustworthiness benchmarking step is designed to account for the security characteristics that cannot be expressed simply as *have* or *don't have* verifications,

and is therefore intrinsically different from the qualification requirements discussed before. The main idea is to analyze and express a general level of trust that can be put on the SUT characteristics according to a set of plausible assumptions (which are based on the set of *threat vectors* relevant in the context of the application domain).

Procedures for accomplishing trustworthiness benchmarking should enumerate and aggregate the systems characteristics that increase or decrease the probability of the effects defined by the threat vectors to manifest themselves, based on information on how this is usually accomplished in the field for each threat. For instance, in the context of a *security benchmark for web applications* let's consider SQL Injection attacks as a threat vector: a trustworthiness benchmarking algorithm could look for evidences (e.g. patterns) showing that the code of the application has some probability of having errors that may lead to SQL Injection vulnerabilities (Amirtahmasebi 2009).

An important aspect about trustworthiness benchmarking is that this kind of evaluation should be done only after verifying that *no obvious ways of attacking the system exist.* In the web applications security benchmark example, we would execute trustworthiness benchmarking only after trying to find actual SQL injection vulnerabilities (e.g. by using automated tools during the qualification step). This is a critical requirement of the approach, as the trustworthiness benchmarking algorithm will not look for actual vulnerabilities, but for the preponderance that hidden vulnerabilities may still exist within the assessed application or system. A more detailed discussion on the properties and justifications for such definition of trustworthiness benchmarking are presented in Section 3.2.2.

In summary, the proposed security benchmarking framework includes a two-step procedure, as depicted in Figure 3.1. First, the systems under testing undertake the set of tests defined in the *qualification step*. The result states whether the SUT is acceptable for use or not (i.e. this step decides if the target has security level zero or more than zero). Qualified systems are subjected to trustworthiness benchmarking, which computes a metric (or set of metrics) that represents how trustworthy the system is in respect to the benchmark threat vectors, while considering the set of characteristics that increase or decrease the probability of the occurrence of the corresponding bad effects. By design, this probability does not take into account the intentions or capabilities of attackers, but only system's characteristics, which are the ones that the system administrator is able to influence. The values are comparable among threat vectors, but not across threat vectors, as the measurement units may differ. For instance, if we have a SQL Injection threat

vector and a Denial of Service threat vector, the comparison of one against the other may or may not be meaningful depending on the way the values are computed.

3.2.1 Security Qualification

The security qualification step within our framework is related to the identifiable characteristics and properties that are considered, in a sense or another, *security requirements* for the target systems to have a security level higher than zero. Basically, in a given domain, the framework assumes that a system has security level zero if it **does not** comply with one of the following assertions:

1) The system provides the set of mechanisms required for securely accomplishing tasks in the specified domain;
2) The set of procedures specified by the benchmark are unable to detect a characteristic (e.g. a vulnerability) that guarantees that a malicious attacker can accomplish a certain *effect* that is either unwanted or violates the business rules of the system.

The **first assertion** is related to the fact that some security mechanisms are naturally expected in certain domains. For example, access controls are expected in database engines, authentication is expected in operating systems, but neither of those are necessarily required for all types of software systems, and might even be optional for those same applications in certain specific use cases. The concrete list of security mechanisms that compose the qualification step definition is highly dependent on the benchmarking domain and on the list of security tasks and activities required in that domain (Section 3.2.3 discusses in detail the problem of the domain definition in the context of the identification of the domain that serves as the main use case in this thesis).

Another example is *disk data encryption*, which is not a universally required security mechanism for database engines, even though for certain usages it could be a requirement (e.g. databases that hold private medical data) (Weber-Jahnke 2007). Encryption of *data in transit*, on the other hand, is more frequently considered a requirement, unless the data that is transmitted is already of public access (Harbitter 2002). Notice, however, that this assertion is related with the capabilities of the target systems, and not with how these capabilities are used in practice. Also, the definition of the set of mechanisms for this step should take into account the fact that the lack of certain mechanisms may be compensated by the existence of others (Howard 2002). For instance, although *encryption of data on the disk* is not supported by some database engines, that can always be implemented by encrypting the same data at the application level - even though it could be harder

to do it correctly and securely. Nevertheless, there are certain mechanisms that are extremely difficult to compensate for and therefore urge for a qualification step. One example is the lack of authentication at the operating system level, which would be an extremely complex security flaw to compensate successfully.

One possible argument against this first assumption is the fact that the security level of a target is not directly related to the security mechanisms it provides, exactly because often they can be compensated during use. Our assumption, however, is that security mechanisms being designed and implemented directly in the target system are always a better choice than adding them later as additional complementary procedures. In other words, when security features are considered from the design of the system instead of being included later as extraneous features, they are not only more efficient, but also provide more capabilities (McGraw 2006). As a simple example, suppose that a database engine providing an intrusion detection system for malicious SQL injection is required for a given scenario. One could argue that a *network sniffing* based solution (e.g. the one proposed in (Fonseca 2008)) using an external software would be more than enough to support this capability, making it pointless to include the existence if such a mechanisms as a qualification requirement. However, while a sniffing based solution can provide detection capabilities, it does not support *prevention* capabilities - by denying the execution of a malicious command - which can be certainly done if the intrusion detection system is designed within the database engine. And even if we could achieve the exact same capabilities with a network sniffing solution through the use of a complex set of communication processes and tools, this solution would, without a doubt, increase considerably the complexity of the architecture, raising the probability of configuration and interaction vulnerabilities and also the overall maintenance effort. Nevertheless, requiring the inclusion (or not) of each security mechanism as part of the qualification step for a given domain should always be based on appropriate reasoning.

The **second assertion** is related to the existence of actual security flaws (i.e. failures of compliance with the defined design) on the SUT that are detectable by current security analysis methodologies. Nowadays there are several distinct techniques that can be used to automatically or manually detect different types of vulnerabilities in all types of systems, and a significant research effort is applied continuously to improve these capabilities. For instance, the effectiveness of static code analysis tools and penetration testing tools are already good enough so that using them to make an initial security evaluation of the target systems is actually worthwhile (Schulte 2012). Also, we have to consider the fact that using these tools is so easy that if the users/managers of a system do not take advantage of them, the

attackers might. Furthermore, even if the detection of such vulnerabilities depends partially on obtaining information that is not public (e.g. the source code of the software might not be open), this does not guarantee that attackers also cannot obtain it. This leads to the inevitable conclusion that we have to assume that *every security flaw that an automated mechanism can detect should be considered of public knowledge*, and therefore this should be the bare minimum analysis that a system should pass before being put into operation.

Obviously, it is arguable whether this procedure should be part of a qualification step or not, in the sense that it may also contribute to the computation of the final metrics in the trustworthiness benchmarking step conducted later. This is an important issue that should be clearly examined. The argument boils down to the fact that the number of flaws detected (by such automated tools) in each system may differ greatly, and they can, to a certain extent, be translated into different degrees of security, allowing to *compare* them instead of simply disqualifying them as we are proposing.

Let's analyze this in the context of an example: a campaign for benchmarking two systems, A and B, and the benchmark specification states that, for qualifying, the systems should pass a static code analysis with a particular tool. Now, let's assume that system A presents one vulnerability and system B presents ten vulnerabilities during this analysis. Consider the following question: *why shouldn't we define system A as more secure than system B?* The reasons why **we should not do so** are actually many, and are summarized in the following points:

- The visibility of a vulnerability has no relation with the total number of vulnerabilities in the system. It may be easier for attackers to find and exploit a unique vulnerability in system A than finding any of the ten vulnerabilities in system B.
- If all vulnerabilities of systems A and B have the same visibility, it may be the case that the damage an attacker is capable of accomplishing in both systems (independently of the total number of vulnerabilities in each) is exactly equal. Therefore, using each system poses the exact same risk to the user.
- It may be the case that one vulnerability in system A is more dangerous than all the others in system B, depending on the systems' internal architecture.
- More importantly, even if both systems had exactly the same number and types of vulnerabilities, the actual damage an attacker can cause depends

on the way the system is used and the value that the system has to the attacker, which is external information that, by definition, should not be part of a security benchmark specification, and therefore should not contribute to the security degree computation.

The worse problem, however, is that the *number of vulnerabilities* - even a number weighted considering the severity of different vulnerabilities and their visibility and whatever else we could think of - is a *fundamentally misleading metric*, even if we could circumvent all the problems mentioned above. To understand this proposition, let's assume that given any two systems there is an algorithm capable of determining, beyond any doubts, that the set of vulnerabilities that exist in system B is more dangerous than the set of vulnerabilities of system A. The key question is: *what happens if we use this information to state that system A is more secure than B?* We believe that the answer is that the benchmark user is *encouraged to choose system A*. The real problem arises from the consequence of this encouragement. Knowing that system A has a certain number of vulnerabilities (now of *public knowledge*) would also motivate the user to not put it into production before correcting those same vulnerabilities, turning system A into a corrected version, without public vulnerabilities, which we may call system A'. But the same can also be done for system B, in this case turning system B into the corrected version B'. Although the initial decision to select system A was based on the fact that A had less severe vulnerabilities, the decision was misleading because using the same rationale to compare systems A' and B' would result in a different conclusion: both systems A' and B' have zero known vulnerabilities and therefore have the same degree of security if we rank them from the perspective of the *severity of the known vulnerabilities*.

This way, we have to assume that whatever flaws and vulnerabilities the qualification step of the benchmark discloses, those will not be present during the use of the system (unless they are harmless, and therefore are not actually *vulnerabilities* in the sense that they do not "allow a malicious attacker to accomplish a certain *effect"*). The reality is that the benchmark user has two choices: either he corrects the vulnerabilities (i.e. patches them), generating a second version of the system, or the vulnerabilities are not corrected and the system is not put into use (thus disqualified, as it has security equal zero, meaning that at least one possible way of attacking the system is of public knowledge). Obviously, in the first case (i.e. if the user patches the target systems), he will end up having a *draw* among all the SUTs (i.e. the corrected ones will not have known vulnerabilities), thus distinguishing the security of those applications cannot be done using information regarding known vulnerabilities. In our framework, this is be the task of *trustworthiness benchmarking*.

As a summary, we would emphasize the following:

> *The actual publicly known flaws or security deficiencies of systems should never be used as official and standard benchmarking metrics in any way, because in a real situation they will likely not be present when the system is put into production. Instead, actual flaws should disqualify systems for use or point the fixes the system needs in order to be acceptable for use. Trustworthiness benchmarking, or the task of evaluating the propensity to unknown or hard to detect security problems, is the only kind of metric that can put one system before the other when nothing can be said about the actual existence of security flaws.*

We believe that this is one of the most important lessons of our thesis discussion and our framework is fundamentally based on this idea.

3.2.2 Trustworthiness Benchmarking

Trustworthiness benchmarking is a process ultimately based on a very intuitive reasoning: *the system that should be trusted the most is the one that demonstrates more evidence of including trustable characteristics*. For any particular domain, trustworthiness benchmarking is the formalization of this intuitive perspective in the form of algorithms able to compute quantitative attributes representing the tendency of the system for having good or bad security. As explained before, even though trustworthiness benchmarking should be applied to systems that do not present obvious security problems (i.e. that passed the qualification phase), this does not exempt them from having characteristics that are related with better or worse security characteristics in general.

Based on the identification of the threat vectors selected for a given domain, the trustworthiness benchmark should identify and group the set of characteristics of the system related to each vector, and should allow a quantification of trust based on their presence, their absence and/or their effectiveness. Although such a benchmark will depend on the domain specification and on the threat vectors being considered, it should express how frequently one could find evidences that allow understanding the probability of the bad effects defined for each vector to manifest. In other words, given some predefined characteristics related to the threats, the process computes the prevalence of such characteristics and their manifestation *density*, based on a predefined expression of the size of the system under testing.

As an example, consider a coding pattern (i.e. a programming style) that is in general known to be a bad programming practice in terms of security. A trustworthiness benchmarking algorithm could be based on counting the number of

times this pattern appears in the source code of the systems being benchmarked, normalized by the size of each system, thus providing a *manifestation density* of such practice. In a higher-level, where the source code is only a small part of the problem, the approach would start from a list of security recommendations that are consensually recommended in the context of the target application domain and compute a compliance level of the system against that list; the main challenge in this case is to understand the consequences of implementing or not such recommendations. In this thesis we explore both these approaches in the context of transactional systems, investigating both of them from their conceptual and fundamental propositions up to their application and validation, evaluating at the same time the limitations of such algorithms and approaches.

A fundamental part of our trustworthiness benchmarking approach is the idea that the characteristics being evaluated and aggregated must be related to the threat vectors *without actually being vulnerabilities themselves*. In other words, those characteristics can be identified as potentially contributing to security or insecurity without being decisive to the existence of security flaws, which dictates the main difference between the qualification and the trustworthiness evaluation. By definition, the characteristics to be considered in this case are usually not enough to allow attacks, but instead they are either partially related with known attack scenarios or they are related with a higher probability of the appearance of vulnerabilities (even if we cannot be sure that any vulnerability really exists). Defined in this way, the system with the higher density of characteristics related with the accomplishment of the effects of threat vectors should be ranked as the least trustworthy one.

The concept of trustworthiness benchmarking is one of the biggest challenges of this work and, in our opinion, the second most important contribution, in addition to the security benchmarking framework as a whole. As explained in the previous section, the result of the qualification step is a system (or set of systems) that has no obvious flaws and vulnerabilities - to the extent of the procedures defined in the benchmark specification - and that are considered acceptable for use. The goal of trustworthiness benchmarking is then to provide the relative level of confidence that the benchmark user can justifiably put into each system when it comes to its ability to avoid the bad effects defined by the threats vectors identified for the domain. This confidence, or trust, may be interpreted as the relative probability of attackers to be successful when trying to attack the system, even though this interpretation is not required. In other words, while the first step of the framework (qualification) provides some guarantees that the system can be put into work, this second step (trustworthiness benchmarking) provides an index that distinguishes

the qualified systems using an estimative of how robust the systems are expected to be in the long run.

While there is plenty of information in the literature that can help in the specification of qualification steps within our security benchmark framework, trustworthiness benchmarking in this particular form is a new proposal, and very little work can be found in the literature concerning the concept (Yang 2011, Toma 2010, Gefen 2002). This way, we devote two entire chapters to the concept, in order to show that the idea of trustworthiness benchmarking is sound and does in fact correlate with security aspects in practice. However, it is important to understand that this kind of evaluation can be seen as a generalization of concepts and practical ideas that are already being used in several areas computer science. Two techniques that are based on the same premises as our proposal are described next.

A long-standing procedure used in the field of computer systems dependability that is based on the same principle we are proposing is called *defect seeding* (Sherriff 2006). Defect seeding is the process of purposefully injecting random bugs in a piece of source code that will be later submitted to manual review for the identification of general bugs (i.e. programming errors). After the review, the ratio between the number of injected bugs that were found and the total number of bugs injected is used to compute an estimative of the number of real bugs that could not be found. The procedure takes advantage of the following assumption: if programming errors are not intentional (and therefore *random)*, they present a normal distribution, and therefore the difficulty of finding errors will be the same for the injected errors and the real ones (assuming that the injected ones also have a normal distribution). This is also true for the distribution of security flaws and vulnerabilities, exactly because we know that, as general bugs, they are also not intentional, and can be viewed as a subset all the bugs of an application. This way, trustworthiness benchmarking will take advantage of the following relation: if a given characteristic of the system can sometimes lead, or be related to, a certain security flaw, then the number of hidden security flaws will tend to be proportional to the number of security characteristics that lead to it. This way, we connect trustworthiness benchmarking with real security characteristics.

Another interesting work that is based on the exact same assumptions as trustworthiness benchmarking is the work on attack surface identification from Pratyusa K. Manadhata (Manadhata 2007). In this work the author demonstrates that a higher number of alternative entry points in a software system increases the probability of one of them being found and ultimately exploited by attackers. In other words, the work demonstrates that the insecurity of software can be correlated

with a higher or lower number of entry points, which are functions of the system that are primarily linked with the possibility of attacks.

Trustworthiness benchmarking extrapolates the ideas presented in the examples above and generally assumes that a system with more evidences for insecurity will in the long run be less secure (or, from another perspective, the one with more evidences for security should be the one we trust more). The reasoning in which we base our approach, however, highlights one important limitation of trustworthiness benchmarking: *evidences for security are no guarantee of security*. For that reason, in our framework, any security guarantees that can be obtained without any doubt should be obtained in the qualification step, while the second step only deals with relative probabilities and unknown factors. For example, if a system has a hidden vulnerability that no static code analyzer, penetration tester or even manual analysis can detect, it is unrealistic to expect any trustworthiness benchmarking procedure to take it into account.

What we should expect is that if some hidden vulnerability is the result of detectable insecurity patterns in the construction or characteristics of the system, then this system will be ranked lower than another system that do not possess any kind of insecurity pattern, or at least has less insecurity patterns, correctly inducing the user to choose the system that is more likely to be secure. For instance, taking the example of the attack surface concept, if a vulnerability in one entry point is a direct result of the existence of too many entry points (i.e. more entry points increase the probability that one of them will have a programming error) then the attack surface metric will be a probabilistic expression of the hidden vulnerabilities. This is exactly trustworthiness benchmarking.

3.2.3 Instantiating the framework

The instantiation of the framework into a concrete benchmark instance is not simple, and several a-priori definitions have to be made. In the following paragraphs we discuss the main aspects that have to be considered, the reasoning behind those requirements and why they are important. We finish with a detailed summary of all the steps involved in the definition of a concrete benchmark.

The actual specification of a benchmark instance starts from the definition and careful study of the *domain* in which we want to apply the benchmark. The term domain usually refers to the specification of some particular application area, like, for instance, operating systems, web servers, databases, etc. However, as we discussed before, security aspects have direct relation with value and utility both for the attacker and for the victim, and applications taken without any context

simply cannot have their security correctly evaluated, as one cannot identify the bad effects that might occur. For example, it is impossible to determine if the lack of authentication in an operating system is a security problem without any assumption of where and how it will be used. For this reason, any security benchmark has to start by assuming some kind of system usage and, depending on the case, the existence of roles of interaction with different security properties. This description of application use-case may be very detailed or extremely brief, depending on the specific situation and the objectives in terms of the representativeness of the results of the benchmark. In this thesis, we refer to this use-case as *base scenario,* which is also the main foundation from where we identify the threat vectors mentioned earlier. This way, without a precise definition of the base scenario it is not possible to make any kind of security judgment.

Ideally, the base scenario description should provide information regarding the expected interactions and roles regarding the potential target systems usage. However, in a security context, more information is required for defining a benchmark. In particular, two very important aspects have to be specified carefully: the benchmark goals and the benchmark user, which actually are definitions that are basically intertwined. Even though the reasoning behind the need for the base scenario is quite intuitive, the reasons for these extra definitions have to be carefully understood.

As already explained, benchmarking can be used to *compare alternatives* or *to evaluate the evolution of a single system in time*. However, these tasks presuppose that something in the domain has alternatives, or that something in the domain may evolve. But this may not be true for all systems at all times, and must be clearly expressed in the benchmark definition. To clarify the difficulties mentioned before, let's consider the case of a DBMS, which is the core of most transactional systems, and let's assume that we want to build a benchmark to compare alternative DBMS engines. A benchmark with such goal would be applied before the installation of the engine, while still being able to point the best engine to use. In this case, there is no environment to correlate with potential security problems, so the benchmark must be capable of taking into account, realistically and in a useful way, the conditions of the future use of the database, and not only the software engine in an isolated way. After the deployment of a chosen DBMS engine, the situation becomes considerably different. In this stage, it is not reasonable to assume anymore that the DBA will keep changing the engine even if a more secure one is found. In most cases, the step of choosing one DBMS engine is a commitment for the life of the system, as the effort to change it is quite significant (involving changing not only the DBMS, but also the applications that use it). Therefore, after

deployment, what we have is a specific engine operating within an environment that evolves. The type of benchmarks that make sense at this point is necessarily different from the ones that were used to select an engine in the first place *even if we consider that the threats might be the same*. In this case, what the administrator needs is a tool that allows understanding the potential security issues of the environment.

The issues discussed above have direct impact in the utility of a benchmark, or, in other words, *how useful the results of the benchmark will be for its user.* Basically what this means is that certain results are only useful in particular moments of the lifetime of the target systems, and a security benchmark that does not take this into consideration might be practically useless despite its correctness. For instance, stating that a particular software is insecure (or is less secure than an alternative) in a context where the user is obliged to use it and cannot replace it is not a useful outcome. On the other hand, stating that one of the characteristics the user has control over (e.g. the configuration of the software or its environment) should be changed to a more secure state is a more useful result.

These reasons show why clearly specifying the user of the benchmark is also an important aspect. In practice, the role of the user of the benchmark defines what makes sense to express in the benchmark and what does not make sense, which will influence the base scenario definition, the qualification and the trustworthiness benchmarking specifications. For instance, a DBA and an application developer are two distinct roles that have different capabilities and assignments, even under the same domain, and therefore would require two distinct security benchmarks. While the first may be interested in securing the DBMS engine against insecure software that connects to it, the second one should be more interested in making the software that connects to the DBMS more secure, even if both systems are part of the same transactional system architecture.

In summary, the definition of a useful benchmark requires the following steps to be previously conducted:

1. Definition of the high-level application domain (i.e. the base definition of the potential benchmark target systems).
2. Specification of the application use-case, the *base scenario*. If applicable, this includes the specification of the main roles that are expected to exist in the use-case in terms of their interaction with the system.
3. Specification of the benchmark user, or who will use the results of the benchmark.

4. Definition of what kind of guidance the benchmark user could expect from the benchmark (i.e. the benchmark goal).

The definitions above allow then the actual development of the benchmark, which is done by the performing three steps:

1. Identification of the *threat vectors* relevant to the security of the base scenario. These threat vectors specification should lead to a strategy for identifying the characteristics of the architecture that are related with the unwanted effects being considered.
2. *Qualification specification*: definition of the set of procedures that allow identifying the systems that are not acceptable for use in the field (i.e. have security 0), given the base scenario specifications. Qualification requires the system under test to meet two requirements:
    a. Have the minimum set of security mechanisms needed to carry out the security tasks identified for the domain and taking into account the base scenario.
    b. Pass the minimum set of evaluations, automated or not, that show that there are no *publicly known ways of attacking the system.*
3. *Trustworthiness benchmarking specification*: definition of the set of algorithms that compute the index that represents a relative amount of trust that can be put in systems, in terms of its characteristics to prevent the manifestation of the undesirable effects determined by the considered threat vectors.

3.3 Transactional Systems: the Case Study

In the previous sections we described transactional systems from a very high level perspective, assuming the reader to have a very basic knowledge of the domain. This section describes in more detail that domain, as it is the focus of the benchmarks presented in the next chapters. First, we describe what exactly a transactional system is and what distinct parts it has. Then, we present the parts of that domain were we made progress and what we actually study in detail throughout our work. Transactional systems were chosen because not only they are a very representative system, which is used by almost all organizations today, but also their complexity is high enough for it to be an interesting evaluation challenge.

3.3.1 Elements of a Transactional System

Transactional systems, as the name suggests, are systems that process data or perform actions through series of *transactions* (Reuter 2008). A transaction is usually defined as a set of elementary steps that should be considered as a unity,

and should not cause any effects if interrupted during mid-execution; either all of them are successful or none is to be executed. Another behavior usually expected from a transaction is that none of its effects are *visible* during mid-execution to other transactions (making concurrent transactions to not perceive the effects of each other before they are completely finished). These expected behaviors are usually referred to as the ACID properties (Atomicity, Consistency, Isolation and Durability (Gray 1992)).

The utility of transactions is that they allow defining complex operations based on more basic commands. Transactions, therefore, endow systems with a trustworthy mechanism capable of composing simple commands into more complex ones, allowing for the system state changes to be as complex as required by the domain. In other words, a set of data can confidently be changed from one consistent state to another consistent state, even if the transformation from one state to the other requires several different processing steps that may fail independently (Gray 1992).

A transactional system is usually based on an architecture designed to help and support some *business domain*, where a set of users want to use computing capabilities with the goal of supporting a specific set of business tasks (Zsifkov 2004). The business domain is usually some specification of a real live enterprise process (or set of processes). In practice, the transactional system *helps* accomplishing the goals of the business, and is typically is composed by three distinct elements namely a database, a Database Management System (DBMS) engine, and one or more client applications. A general and very common setup is depicted in Figure 3.2.

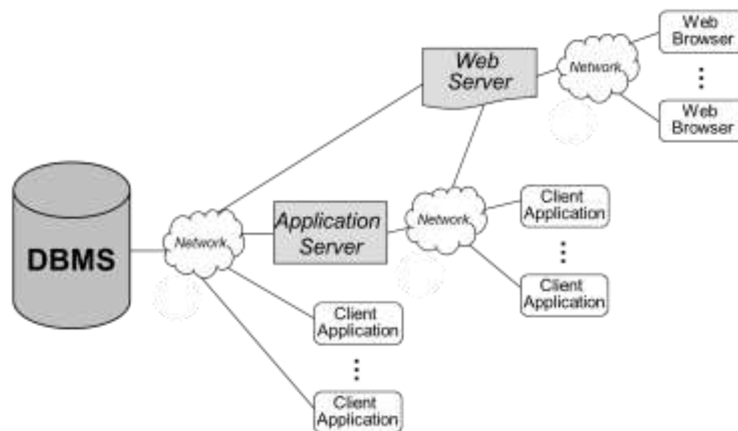


Figure 3.2 A typical transactional system architecture.

The database is the logical implementation of a data model, which is used to shape, store and maintain data regarding the *business domain* that the transactional system is designed to support (Gray 1992). The DBMS engine, on the other hand, is the software system that maintains the database and provides interfaces for interacting with it, allowing for the modification of the data model, as well as the provision of commands that allow the insertion, modification and removal of data from the database. In other words, the database data model is the vision that the DBMS provides to the users of the data they input into it. The DBMS is also responsible for maintaining and enforcing the series of access rules that ultimately define what is permitted and what is forbidden within the database (Gray 1992).

The client applications are the software implementation of the rules of the business domain, even if some part of those rules sometimes are actually implemented inside the DBMS engine (Eisenberg 1996). For example, a client application may include the algorithms that decide what different types of data are to be stored together, the consistency requirements for this data to be valid, the computations that are permitted over this data, the methods and views through which the data is accessible and, also, what are the transactions that are required for maintaining the business rules of the domain. Transactions appear in this context as the sets of transformations that will keep the database in the consistent states defined by the business rules.

One important characteristic of transactional systems is that most of them implement a client server architecture (in a two, three or multitier architectures (Ram 1999), meaning that the client applications and the DBMS engine communicate through a network of some sort. This leads to the reality that most clients applications run in an environment completely different from the DBMS engine, and, at the very least, the information that the user inputs may come from insecure environment and devices. This leads to a whole sort of security complications and characteristics that make transactional systems a very attractive domain for our study.

3.3.2 Security Benchmarking of Transactional Systems

To consider transactional systems as our case study, we need to examine the question of *what do we expect* from a security benchmark in this domain. The first aspect one notices from the description in the previous section is that even though we defined a transactional system has having 3 elements (i.e. the database, the DBMS engine, and the client applications), when we realize that the database itself is only a piece of data that is inside the scope of the DBMS engine, we conclude that actually there are preferable two levels of software to be considered: the DBMS

and the clients. Both of these have not only completely different operational environments, but also completely different security goals, as we examine next.

The DBMS engine is a piece of software that must ensure the integrity of the databases it holds (Allard 2010). As being mainly software that receives *orders and commands* from the applications, it must provide guarantees to the clients that data changes do not occur except under authorized conditions. In other words, the primary goal of the DBMS is not prevent the data it holds from suffering any other manipulation besides the ones submitted by a correctly authenticated application. Such guarantees, however, have to take into account all the ways the potential attackers can use to interact with the database engine, which are not only associated with the engine itself, but extend also to the underlying operating system, hardware, network, etc., and all the personnel that is in charge of maintaining the correct functioning of the system, from the Database Administrators (DBAs), to the developers that interact directly with the DBMS and the maintenance crew in charge of backups. All these variables have to be accounted for if one wants to have some kind of clue about the security level of the DBMS engine.

Throughout this thesis, we will refer to the DBMS engine plus the entire underlying environment as a *transactional system infrastructure*, in the sense that it is the part of the system that gives the fundamental support for the definition and implementation of end users business rules. Additional security characteristics of transactional systems infrastructures are discussed in Chapter 4, where we study the application of our framework to this part of a system and put forward ways for characterizing in a practical and comparable manner real live installations. It is important to notice, however, that given such complexity, the selection of the DBMS engine itself is also a very important, yet very difficult thing to do correctly from a security perspective. This particular problem is revisited in Chapter 6.

The security goals of the client applications, on the other hand, are somewhat different. As they are the part of the system that defines and implements the *business* rules of the domain, the most important thing that we want to be sure is that such business rules are correctly implemented and cannot be broken. Usually, functional testing of the software that is being developed tries to identify if the defined business rules are correctly implemented, and if the system actually does correctly what the users require it to do (Zsifkov 2004). However, doing what it should is not the same thing as not doing what it should not do. Robustness and security testing (Shahriar 2012) are techniques that can be used to test the system conformity and correctness from the perspective of either unexpected interactions or malicious interactions (i.e. interactions where an attacker may use any mean available to lead the system to break some business rule). In practice, when

selecting an application one wants to know how likely it is for an attacker to be successful if he tries to force the system to execute an illegal action. We discuss these problems in detail in Chapter 5, where we identify and validate ways for supporting such selection process under the assumptions of our framework.

3.4 Conclusion

This chapter presented a security benchmarking framework, which is divided in two parts: security qualification and trustworthiness benchmarking. We thoroughly discussed the framework and the reasoning behind it. The main idea is that the correct classification of systems concerning security attributes has to be done by means of separating the processes used to evaluate the knowable security aspects of the target (e.g. the search of existing vulnerabilities) from the evaluation of the aspects we can only estimate (e.g. the probability of a system having hidden vulnerabilities.).

Security qualification is the process designed to deal with the tangible security characteristics of the system being evaluated, and the main result of it is the identification of the systems that are acceptable for the domain. Identifiable security vulnerabilities and the lack of fundamental security mechanisms necessary for the accomplishment of the required security tasks in the domain are the primary reasons for disqualifying alternatives, which are then considered as having security level equal zero.

Trustworthiness benchmarking should then be applied to the systems that are considered acceptable. This process, therefore, estimates the amount of trust that we can justifiably have that the system will not bring security problems in the future due to undetectable vulnerabilities or the lack of proper security precautions.

The chapter ended with an introduction to the particular domain that will serve as use case for the instantiation of the framework on concrete benchmarks, i.e. transactional systems. From a security point-of-view, we divided transactional systems in two parts, the transactional systems system infrastructure, which is addressed in Chapter 4, and the business applications, which are studied in Chapter 5.

4

Security Benchmarking of Transactional Systems Infrastructures

Database-centered transactional systems are typically designed following a client-server architecture (Ram 1999). As such, they can be divided in two main parts: the database server infrastructure, which is centered on the DBMS engine and its related software and hardware appliances; and the business applications that implement the business logic and provide the end user interfaces. Although these two parts are highly tied, they have completely different characteristics, thus requiring different approaches in a security benchmarking context.

The database server infrastructure is usually maintained by a small group of Database Administrators (DBAs). Its security characteristics are strongly influenced by the large number of configuration alternatives provided both by the DBMS engine and the network and server configurations that relate with it (for example, in most cases the operating system configuration directly affects the

DBMS security features). The problem here is that, for an average DBA[1], it is extremely hard to keep track of all the details that may influence the security of a database installation. Furthermore, a key aspect regarding the security of the database infrastructure is that the damage that an attacker can cause strongly varies depending on the characteristic he exploits. This effect can be clarified by a simple example: consider an attacker that exploits a weakness in a backup system in order to obtain confidential information; this form of attack prevents him (on the majority of the scenarios) from modifying the information, which would otherwise be possible if he somehow obtained access directly to the DBMS engine.

The business applications, on the other hand, are usually well specified and the security problems that they may have can be narrowed down to a much smaller list of possible variations and bad effects, making them much easier to understand (Russel 1991). In fact, as the goal of these applications is to enforce the business rules of the service they are built to support, security risks typically consist of failing in enforcing such rules. This is normally related with programming mistakes that allow following execution paths that were not originally intended (e.g. a SQL Injection vulnerability that permits a data change that should not be allowed). Although such mistakes are hard to detect and prevent during development, once exposed they are quite easy to analyze and correct (Shahriar 2012).

In this chapter we apply the framework proposed in Chapter 3 to build a security benchmark for database-centric transactional system infrastructures (security benchmarking of business applications is addressed in Chapter 5). First, we define a generic scenario (the *Base Scenario*), in which we specify the boundaries of what we are considering to be a transactional system infrastructure. Then we discuss the approaches for security qualification and trustworthiness benchmarking of DBMS infrastructures, applying the abstract concepts defined in Chapter 3 to the concrete base scenario. To demonstrate the proposed benchmark, Section 4.4 presents a case study, where the benchmark has been applied to compare four real installations using four distinct DBMS engines. Finally, Section 4.5 concludes the chapter.

---

[1] Defining an "average" DBA is not trivial. In this context, we consider an "average" DBA as someone that is not an absolute expert in every single system involved in the database installation (or installations) he is in charge of. Practice shows that most DBAs in small and medium size organizations are not security experts and do not hold extensive knowledge about all the possible configuration options of the infrastructure elements, including the OS, network elements, etc.

4.1 Base Scenario

In order to be able to make decisions regarding the benchmark definition, we need to make some background assumptions regarding a few characteristics of the benchmarking domain (i.e. the environment for which the benchmark is being designed). As explained before, security is both related to value (something that the attacker may obtain or the victim might lose), and resources or capabilities needed to gain that value. Both of these require tangible properties to be considered, or else it is impossible to reason about security aspects in a practical manner. This way, two key restrictions are considered in this base scenario: first, the scenario is as generic as possible, to allow the benchmark to be applicable to the largest possible number of real installations; second, the scenario is specified in order to be representative of security concerns in real applications.

In fact, even if personal database applications (e.g. an application for storing and managing personal notes) may have security implications, a security benchmark for such a domain would clearly have very limited interest. Much more relevant are situations where critical personal and business data are at stake, and/or where security problems may affect a very large number of individuals simultaneously. Furthermore, as we are targeting the transactional system infrastructure, the particularities of the business applications can be abstracted, focusing only on the classes of users that interact with the database (i.e. the virtual identities that relate with the system). The idea is that the database infrastructure should protect itself against exploitations of characteristics of the environment and vulnerabilities of the business applications; therefore, benchmarking the security of the infrastructure should not be constrained by the business applications specificities.

The following points detail the key assumptions and characteristics of the proposed base scenario (their representativeness is discussed later in this chapter), which largely shape the benchmarking domain and provide the boundaries for the definition of the benchmark components:

1. The infrastructure is composed of a relational DBMS engine on top of an operating system (OS) running on a single physical computer. Although this is a simple configuration, practice shows that it is representative of the large majority of database installation in the field.

2. The platform is connected to a local area network (LAN), and the DBMS may be accessed locally (from the console) or through that network (from client hosts or application servers). The LAN may have a connection to the Internet. However, Internet users do not connect directly to the DBMS

(prevented via common network configurations), but only indirectly through web applications hosted by application servers.

3. The DBA is the overall administrator of the environment, and either there is only one DBA or several DBAs that act as a single entity (by making consensual choices for the system configuration, which is a typical management approach in the context of complex installations).

4. Threats are always associated to individuals, which might (or might not) have a legitimate relation with the system. The only trusted individual is the DBA. All others are assumed to be potentially untrustworthy (i.e. a pessimistic approach is followed when it comes to security issues), and thus may try to compromise the system in some way.

5. Individuals always interact with the system through *userids*, which are virtual identities assigned to each individual (or set of individuals) depending on the relation he has with the system. *Userids* are verified by an authentication procedure and belong to one of the following *interaction classes*:

> a. *Application userid*: users that authenticate and interact with the database system using a business application (e.g. a web-based application), and whose actions are restricted by the application's rules;
>
> b. *Operating System (OS) userid*: users that authenticate directly to the OS and whose actions are restricted by the configuration of the OS environment;
>
> c. *DBMS userid*: users that authenticate using the DBMS authentication mechanisms and whose actions are restricted by the DBMS configuration and environment.

6. Real individuals have roles that entitle them for one or more *userids*. For example, end-users have only an application *userid* and developers may have a DBMS *userid* and also an OS *userid*. The DBA may hold the three types of *userids*, while maintenance staff typically has only an OS *userid*. Real individuals that are not users of the system do not have a legitimate *userid*, and they may interact with the system only through interfaces that have an anonymous network access (e.g. an authentication web page that

may be open to the Internet, or the operating system network layer that responds to ICMP (Stallings 2010) requests coming from the LAN).

7. Custom business application code (implemented by developers and restricted by the database administration policies that dictate how applications may connect to the DBMS) may run on a local web server, on top of remote application servers, on remote client hosts inside the LAN, or inside the DBMS as stored procedures.

A key characteristic of the scenario proposed above is related to the interaction classes (item 5). The definition of interaction classes assumes that a real individual either has its relation with the system defined by one (or more) of the three interaction classes (i.e. application, operating system, or DBMS *userids*) or has no official relation with the system. Although in practice the relationships may be much more complex than that, this approach simplifies the analysis of the system security, as a potential attacker must always act on the system through one of these classes. Each of the four relationships defines a distinct *environment container*, with a predefined set of privileges associated with it, which must be taken into consideration when analyzing the security of the system. This is particularly relevant in infrastructures as complex as a database installation, especially when inside threats are being considered Actually, we should emphasize that in database environments, inside threats must be seen as as relevant as anonymous Internet attacks: insiders may even be more hazardous, as they frequently have pre-established security privileges within the system (Bishop 2008). Thus, small vulnerabilities may be more risky when facing an insider attack than when facing an unknown Internet hacker.

In the case of applications that are publicly available to the Internet, we assume that all users have an application *userid* that grants them privileges to access the publicly available parts of the existing business applications. In our benchmark, all those cases (from the insider threats to the anonymous Internet users) are taken into account by evaluating each threat from the point of view of all the different interaction classes.

It is important to realize that the definitions presented above are representative of a very large number of real DBMS infrastructures. Even though such definitions are quite complete (i.e. they include the most relevant aspects, from a security point-of-view), they are at the same time very flexible. For instance, although the scenario considers the existence of application developers, it is flexible enough to consider the situations where the DBA is the only developer and also the situations where there is a software development team (i.e. the number of developers is not a

constraint). Also, no specific structure is imposed on the local area network, as that would be extremely complex due to the large number of possible variants; the only assumption is that any connection to the Internet goes through a specific point of communication, and direct connections to the DBMS are not possible from outside the LAN. Obviously, we could also consider other scenarios with alternative assumptions, including: DBMS replicas, applications using more than one DBMS engine, three tier architectures, multiple DBAs and operating systems running inside virtual machines, etc. However, as we will show later, it would be quite straightforward to consider such cases during the benchmark definition. In practice, we decided not to overcomplicate the base scenario, as our main goal is to show the validity of the framework, and not to propose an universal benchmark. Benchmarking should be a joint initiative, taking input from several parties (Bondavalli 2009).

A final relevant aspect is that the complexity of the environments that fit the assumptions above makes them highly prone to the appearance of vulnerabilities (Russel 1991). The benchmark must help the administrator understanding the threats to which a configuration is more exposed, allowing him to make educated decisions and address primarily the most critical problems from his own perspective.

4.2 Security Qualification

As defined by the base setup presented before, a transactional system infrastructure (sometimes also referred to as *database infrastructure*) is a set of network, hardware and software elements that are configured in a way that provides the support for the business applications (which in fact implement the end users solutions). Without considering an enclosing environment, evaluating the security of a DBMS infrastructure is very hard, as no threats can be assumed beforehand. This happens because we cannot pinpoint what is valuable and should be protected, and what is not valuable and would never impose a loss to the system owner. At the same time, the security of any business application, which is the main reason for the existence of the infrastructure, depends ultimately on the correct configuration of the DBMS infrastructure. Therefore, we face the fact that the choices made before the deployment of a business application do have impact in the security of all the systems involved.

Security qualification is the step where we identify what is acceptable and what is not acceptable in terms of security within a domain. As explained in Chapter 3, this analysis is based on two key aspects: 1) the vulnerabilities that allow someone to attack the system, and 2) the security mechanisms that are required for a system in

that domain. The problem is that, without considering the business applications specificities it is not possible to reliably reason about the vulnerabilities that allow a transactional system infrastructure to be attacked, making security qualification based on the analysis of vulnerabilities misleading. This is due to two reasons: first, one cannot identify what is supposed to be protected (e.g. all resources may be public, thus some vulnerabilities are irrelevant); second, when protecting a scenario as complex as a transactional system, the system administrator usually follows a defense-in-depth approach (Howard 2002), meaning that any single vulnerability may be mitigated by an alternative security layer (thus, not actually being an exposed attacker entry point).

The second aspect regarding security qualification (the security mechanisms that are required for the system to be used in the benchmark domain), can however be addressed without considering the business applications. In fact, assuming that for tuning the security configuration of a live infrastructure the administrator effectively makes use of the set of available mechanisms to maximize the system defense surface (following a defense-in-depth approach), then it is possible to qualify the underlying software (i.e. the software elements that will support the DBMS infrastructure) taking into account the specific configuration the administrator intends to deploy. In other words, it is possible to qualify final products (e.g. DBMS engines, OS) in terms of the mechanisms they provide for the administrator to defend his infrastructure. This can be done by comparing the intended configuration with the set of mechanisms provided by the benchmarked software (mechanisms that are required but not provided by a given software package, may disqualify in the proposed benchmarking process).

This aspect is thoroughly discussed in Chapter 6, in which we propose a qualification benchmark that can be used by administrators to select software packages for transactional system infrastructures. Such qualification process is heavily based on the lessons learned when defining the trustworthiness benchmark presented in the next sections, and will allow answering a very specific question: *how can a DBA choose a DBMS engine (and the other supporting software) that ultimately allows easily securing a transactional system infrastructure*?

4.3 Trustworthiness Benchmarking

Within the framework proposed in Chapter 3, security comparison is given by performing trustworthiness benchmarking, where the goal is to provide some kind of estimation of the proneness of some security premise to be broken. As mentioned before, given an isolated DBMS infrastructure, it is not possible to reason about its security in terms of breaches of the business rules, as such rules are not

implemented yet: they will only exist when business applications are deployed (when considering a benchmark for a transactional system infrastructure, we are not considering the businesses applications, but the infrastructure that may be the backbone of a set of applications; we will discuss the design of such a benchmark in Chapter 5). However, as those business rules will eventually exist and will have to be enforced when a business application is deployed on top of the transactional system infrastructure, trustworthiness benchmarking can be used to:

> *assess and compare how much control the administrator has over his infrastructure* or, in other words, *how much certainty the administrator can have that his infrastructure will not be used to break the business rules without his consent.*

In practice, the benchmark should allow the administrator (i.e. the DBA, as specified in the base setup) to assess and compare the effectiveness of different configurations on *preventing attackers from using the infrastructure to break the restrictions imposed by the business applications*. This is the most useful point-of-view to take when benchmarking the security of a transactional system infrastructure, and therefore is the one chosen to guide the definition of our benchmark.

For the actual benchmark definition we propose four key steps. Although these steps intend to be generic (i.e. applicable for the definition of any trustworthiness benchmark for transactional systems infrastructures), they are based on the base setup described before, and may need to be adjusted when considering scenarios with different characteristics. The steps are as follows:

1) *Identify the threat vectors that are relevant in the context of the scenario (i.e. that are representative of real threats)*. The first piece of information needed for defining a trustworthiness benchmark is the definition of what are the *bad, undesirable or harmful effects* that are considered to be relevant security issues. In a transactional system infrastructure, the threat vectors should be consist of generic effects that may allow or facilitate attackers to compromise the security of the business applications that run on top of the infrastructure. For example, Denial of Service prevents the business applications of obtaining the data they need, even if the applications themselves are working. Other example would be obtaining access to private information through means that the business applications developers are not even aware that exist (Side-Channel Information Disclosure).

2) *Identify the system elements that influence the probability of one or more of the threat vectors being instantiated as attacks*. After devising the list of harmful effects, we need to enumerate as thoroughly as possibly the set of elements related with them. For example, encrypting communication channels allow preventing obtaining private information, and so does the encryption of backups. Also, small precautions like having the DBMS engine daemon with the least amount of privileges also prevent extended damage in the case of application's vulnerabilities (i.e. if privileges are correctly set, one application might not be able to affect another, improving the overall security of the infrastructure despite the vulnerabilities). Such elements may be directly extrapolated from the harmful effects, or identified based on other types of analysis and research, and may consist of security mechanisms, processes, configurations, procedures and behaviors associated with security in the benchmarking domain (in this case, the transactional system infrastructure). This list will serve as the base for the definition of what should be taken into account when evaluating trustworthiness aspects.

3) *Define how much each element influences the security of the infrastructure (in average)*. This is the most difficult and controversial part of the process, as it may depend on the characteristics of the environment being evaluated, something that should be avoided in a benchmark for portability reasons. At the same time, it is unrealistic to assume that all the security elements provide the same contribution to the security surface of a system, even from a generic point of view. For example, the security impact of having the DBMS daemon running with administrative privileges within the operating system cannot be the same as the impact of a complete lack of auditing or privilege management capabilities. This way we need to assign a level of influence to each element, even if in an approximate manner.

4) *Identify how the security elements relate with the threat vectors*. Even though it is clear that the security elements identified in step 2 are related with security, the threat vector (or vectors) they are related with is not always obvious. The goal of this step is to perform an analysis that allows such identification. An example is: the execution of the DBMS engine daemon with excessive privileges may lead to what security problems? This identification is particularly tricky, especially because the original security elements cannot be obtained through a methodical and/or formal method. We thus propose the concept of *pessimistic scenarios* to make the correlation between security elements and threat vectors. For example, obtaining physical access to the DBMS engine server may allow either to

have access to the operating system or directly obtaining the raw data stored in the physical hard drives, but such things are only possible "everything goes bad", which is more or less the idea of the pessimistic scenarios.

These four steps allow the design of a benchmarking procedure to guide the assessment process of any concrete environment that fits our base setup specification, which should allow the computation of the trustworthiness metrics. The next sections thoroughly present the actual process we used to define the benchmark, discuss the difficulties and decisions taken along the process, and the set of concrete steps that allowed us to build a trustworthiness measurement tool that can be used by DBAs. As it will become evident, most of the steps takes advantage of field research and practice in an attempt to make the benchmark as representative and realistic as possible. Although we realize that work based on field research has limits (which is why formal methodologies are often preferred), there are no formal methods available to accomplish our goals (and it seems extremely hard to even propose one). This way, we are left with the field experience of professionals. To better understand the problem, we also discuss and analyze the limitations that such approach imposes on each step of the proposed benchmarking methodology.

4.3.1 Threat Vectors

In the context of our framework, the first step towards the implementation of a trustworthiness benchmark is the identification of the effects or circumstances that are considered security violations, which we define as *threat vectors* (as proposed in Chapter 3). In the context of transactional system infrastructures analyzed without taking into consideration any business applications, the effects that we want to identify are the ones that are generically associated to security breaches in the presence of any set of conceivable business applications. In other words, the goal is to identify the effects or circumstances that may allow (or facilitate) attackers to circumvent one or more of the rules that the business applications will be in charge of enforcing.

Lists enumerating typical threats in the transactional systems domain are not obvious or easy to obtain, and a set based simply on the breach of CIA properties is too generic to be useful in practice (Parker 2002). A slightly more targeted approach could be based on the STRIDE threat modeling methodology (see Chapter 2 for more details), which proposes the following list of security threats:

- ***Spoofing**: threats that involve an entity using another identity that is not its own;*

- ***Tampering**: threats that involve unauthorized modification of data or another part of the system;*
- ***Repudiation**: threats involving the denial of someone performing an action;*
- ***Information disclosure**: threats involving the exposition of information to an unauthorized entity;*
- ***Denial of service**: threats that may lead a particular service to become unavailable to its users;*
- ***Elevation of privileges**: threats that may allow an entity to obtain more privileges than it was originally supposed to have.*

Although a relevant starting point, STRIDE is also too generic, and the actual semantics involved in the application of each of these threats in the context of transactional systems are too open for disagreements. To understand why, take, for instance, the *Information disclosure* threat. Whenever an end user provides some confidential information via a business application, this information immediately goes through the following workflow (or a variation of it): first it is processed by the user interface application, then it is transmitted through an arbitrarily complex network to a server (that may be the DBMS server directly or an intermediary application server), and finally it may be processed by this server or by the DBMS engine before being stored in the database files. The information may be temporarily stored in the server's memory and written to a permanent storage device, which may then be copied to another media for backup. This way, unauthorized access to this information can happen through several distinct means (Payton 2006), including:

- *Physical access to the server*: even if logical access to the server is protected through reliable authentication procedures, the data can be obtained from the memory footprints of RAM circuits, or even from the physical hard drives. Alternative system boot from optical or USB drives are also known to be possible.
- *Interception of traffic data*: network traffic can be collected not only on the intermediate network routing equipment, but also at the end-points, where privileged access to the operating system of the database server or to the client device would allow reading the information.
- *Interception of backup copies*: physically and/or logically unprotected backup devices can also be used to access unauthorized information (even if backups can hardly be used to modify data).

- *Insider threats*: several different persons may work together in the maintenance and administration of the infrastructure, thus having legitimate access to the information. Thus, it is important to account for the possibility of people taking advantage of their privileges in order to obtain information they should not have access to.

In practice, we need a list of threats like the ones defined by STRIDE, but that takes into account the characteristics of our base scenario. A possible approach that could be used to accomplish this would be to collect information concerning real instances of security breaches (i.e. real cases like the ones proposed above) and extrapolate the effects involved in each one, grouping them in large categories. The problem is that details concerning attacks to real installations are extremely hard to find. This is mainly due to the fact that administrators tend to follow a "*security through obscurity*" approach, thus hiding the occurrence of any successful attack events against their systems (Pavlovic 2011). Their reasoning is that disclosing such information could draw the attention to the existing weaknesses, opening the door for more attacks.

Information that can be more easily found is related to implementation bugs in real DBMS engines that turn out to be security vulnerabilities (Messmer 2012). In theory, these bugs could be analyzed in terms of the threats to systems in the field (even if it is not always possible to identify how they can be used to breach business rules of applications, as this would depend on how these applications are designed). However, a trustworthiness benchmark for our base scenario cannot be based on this kind of information (at least not completely), for two reasons: first, because software bugs do not account for all the security effects in a DBMS installation, and are not representative of the large number of issues caused by the myriad of possible configuration errors; second, because fixing software defects in a DBMS engine is not usually the DBA's responsibility, and therefore the effects of these bugs could hardly be avoidable in a real situation. The only measure the DBA can take is to install, as soon as possible, the existing patches that fix software defects. In other words, such benchmark would be of reduced usefulness for DBAs, in the majority of the cases, as it would simply provide information about something that the DBA is not able to change unless he replaces the whole infrastructure (which is unrealistic in most scenarios).

To identify the relevant threat vectors for DBMS infrastructures we conducted an extensive field search. We started from a wild range of documents and papers from a variety of sources, like white papers, manuals, research papers, etc., and analyzed them for the kind of information we needed (i.e., what should we prevent from happening within a transactional system infrastructure). As this specific kind of

information is too scattered and most documentation we found just touch these issues superficially (and we need at least some level of justification for each threat), we finally ended up reducing the initial sample to three main sources of information: the original six STRIDE threats, two protection profiles for databases from the Common Criteria evaluation methodology (Common Criteria 1998, 2000), and a popular white paper (Shoulman 2009) that presents a consensual "top 10" of database threats. The reasoning behind this decision is that each of these sources already includes a summary of the consensual threats identified by the groups that created them. Thus, the intersection of the four documents provides, in our opinion, a representative set of the threats.

We then analyzed the information contained in the chosen documents and rewrote the threat definitions that they present in the form required by our benchmark, which can be stated as: *what effects that we do not want to happen in our base scenario infrastructure.* Our analysis resulted in a set of eight infrastructure threats, which are representative of all the threat information that could be found in the four documents. We further validated and refined the list and the definitions regarding their completeness and correctness, by asking the opinion of a large set of experts, including database administrators (at least four administrators has more than 3 years of practical experience) and researchers (at scientific conferences). Table 4.1 presents the final threat vectors, their definition and some examples of security aspects related with each vector.

Orthogonality was a key aspect considered in the definition of the threat vectors. In fact, the vectors have to be as representative as possible of the real attack threats and malicious effects that may occur in the context of our generic infrastructure, but they also have to be as orthogonal as possible among themselves, allowing for a reduced overlap among different vectors. For instance, the white paper analyzed (Shoulman 2009) includes platform vulnerabilities as one of the top 10 database threats. However, from a DBMS configuration point of view, most platform vulnerabilities (like operating system vulnerabilities) are used as a way for *maliciously obtaining privileges*. Thus, such a vector clearly matches a very important threat defined by STRIDE: *privilege elevation*. These observations, together with a careful analysis of the documents mentioned before, allowed us to define what we believe to be the eight more relevant DBMS infrastructure configuration threat vectors. Nevertheless, it is important that the threat vectors provided here are periodically evaluated, and whenever necessary, the list should be adapted and improved.

Table 4.1 Potential threat vectors in DBMS infrastructures

Threat Vector	Description
Legitimate excessive privilege achievement (LegExPrA)	This threat is related to configuration characteristics that increase the probability of allowing a user to obtain more privileges than the ones he is supposed to have. These excessive privileges are a threat because, by definition, they allow the user to perform unauthorized actions. Examples of issues that may lead to legitimate excessive privilege achievement are: granting privileges with using open ended expressions (e.g. ALL and ANY keywords, which define the way privileges can be forwarded), not implementing views to hide unnecessary columns, and using an administrator OS *userid* to execute the DBMS engine daemon
Illegitimate privilege elevation (IllPrEl)	This threat is related to configuration characteristics that increase the probability of allowing a user to obtain an arbitrary privilege that he should not have in any circumstances. An attacker usually achieves illegitimate privileges by actively exploiting vulnerabilities at some level of the system. Examples of vulnerabilities that may lead to illegitimate privilege elevation are: not using a dedicated platform, not patching the DBMS or OS software, and not disabling unused protocols on the network stack
Denial of Service (DoS)	This threat is related to configuration characteristics that increase the probability of a user being denied timely access to some functionality or resource. Examples of issues that may lead to DoS are: not making and testing backups, storing log information in the OS partition, and not properly setting OS file system privileges of the DBMS data files
Communication Weakness (CommW)	This threat is related to configuration characteristics that increase the probability of a communication channel between a user and the DBMS to behave in an improper way. This threat includes sensitive information disclosure as well as traffic manipulation and diversion, and may be due to: not encrypting a remote connection, using a default or self signed certificate for a server, and placing production and development servers on the same network segment, etc.
Authentication Weakness (AuthW)	This threat is related to configuration characteristics that increase the probability of allowing an individual to become authenticated to the system as another individual. Examples of vulnerabilities that may lead to this are: storing password information in clear text, not forcing strong password policies, not excluding default *userids*, and using host based authentication

Side-Channel Data Exposure (SCDtEx)	This threat is related to configuration characteristics that increase the probability of sensitive information to be accessed through an alternative (i.e. illegitimate) access channel. Vulnerabilies that may lead to side-channel data exposure are: storing schema creation SQL files in the DBMS platform, not protecting backup files, not configuring access permissions of DBMS data files, etc.
Audit Trail Weakness (AudTW)	This threat is related to configuration characteristics that may result in a decreased ability to identify unexpected behavior (including its causes and possible suspects). It includes not only real audit functionalities, but also logging mechanisms and other tracking facilities. Problems include: not auditing sensitive information, not protecting log files, and not auditing application code changes
SQL Injection Enhancement (SQLI)	This threat is related to configuration characteristics that increase the probability of an SQL injection vulnerability to be exposed or enhanced. Examples of such characteristics are: not disabling DBMS extensions that allow file system operations, not implementing least privileges policies, and not protecting application code

The threat vectors (presented in Table 4.1), combined with the interaction classes defined for the base scenario, will ultimately serve for calculating the trustworthiness index of the evaluated systems. These two dimensions give the DBA the flexibility to focus on the areas of the system that are more important considering the particularities of his particular installation, and allow the use of the same benchmark in very distinct transactional systems architectures. To exemplify and demonstrate how a DBA can use these dimensions to tailor the benchmark results to his environment, lets consider two different, but quite common scenarios:

- ***Scenario A****: in this scenario, the DBMS used is a MySQL engine over a Linux system. The operating system also hosts an Apache web server, which runs a single application developed in PHP that connects directly to the database that runs in the same server. The system is maintained by a single person that is, at the same time, the DBA and the developer of the application (i.e. no other person has a valid userid on the DBMS or on the operating system). All users connect to the system using via a business application, whose interface runs over a web browser, and that communicate with the server using the https protocol for security.*
- ***Scenario B****: in this scenario, a dedicated machine hosts a SQL Server DBMS engine over a Windows 2003 operating system. The DBMS is accessed directly by several stand-alone applications developed using the*

> *Delphi language. Applications run on client hosts spread over a LAN and on another server machine that hosts an Internet Information Services web server and several ASP web applications that are accessed from the Internet. A large team of developers has DBMS userids with a variety of roles and privileges, and the maintenance staff executes backup procedures every night, using operating system and DBMS userids.*

Applying the benchmark consists of executing the procedure and computing the corresponding levels of trustworthiness (as discussed later) for each threat vector within each interaction class. Tailoring the results consists then of focusing the analysis on the values that make sense for each scenario. For example, in scenario A threats related to communications channels are very unlikely to be a concern because communication with the database occurs only locally from the web server process, which then communicates with the users using the https protocol, which is known to be secure. At the same time, the DBA does not have to worry about regular operating system users causing problems because he is the only one with an OS *userid*. This way, he might decide not to spend time fixing privileges on the file system, something that could be a problem in another context. He should be, however, very concerned with application bugs (e.g. SQL Injection vulnerabilities) that would allow for a non-system user to obtain private information. Additionally, he should also worry about the availability of the database application.

The concerns of the DBA in scenario B are quite different. With so many developers and applications he must not lose the control of privileges within the DBMS (which could lead to unintentionally granting someone excessive privileges). Also, the DBA is demanded to continuously collect and analyze reports about unusual behaviors and he must be able to pinpoint the suspects and the causes when attacks happen. At the same time, he should assume that several individuals that cannot be fully trusted (e.g. the developers) may run commands in the operating system, and therefore should apply measures to minimize the consequences in the case of a disgruntled maintenance staff. The DBA should also realize that the local network is very complex and insecure, and that connections between the remote clients and the database should be protected.

The dimensions to consider when analyzing the benchmark results are obviously different for the two scenarios (e.g. threats related to communications channels are more relevant in scenario B than in scenario A), and should allow the DBA to prioritize the areas that are of more relevance for security revision (i.e. dimensions for which the configuration is less trustworthy). This might also help the DBA justifying the need for replacing specific components of the infrastructure. For instance, if it is too hard to obtain auditing information in a particular DBMS engine and that is identified as a high priority for the specific environment (as is the case

of scenario B), then the DBA may consider replacing his DBMS engine (or add some external auditing feature that provides the same information). The same reasoning applies to an operating system that makes it difficult to keep file system permissions organized, or that has vulnerabilities being frequently disclosed and reported. Whenever a DBA justifiably distrusts such aspects (being supported by a systematic evaluation approach, like the proposed benchmark), then there is a good justification to engage in radical environment modifications like these.

4.3.2 Security Recommendations

Reliably securing a database infrastructure (like the one represented by our base scenario) requires the administrator to follow a Defense-in-Depth approach (Howard & LeBlanc, 2002). Defense-in-Depth can be seen as a *reasoning framework* in which one always assume that any security mechanism can fail, and therefore, security depends on several layers of mechanisms that compensate the failures of each other. For instance, no one would ever test thoroughly an application and assume that this precaution would compensate the installation of a database engine with default settings and empty passwords. At the same time, no one would install a firewall on the network and assume that no outside user would ever be able to gain control of internal servers.

Any level of acceptable security comes from the combination of several configurations which, in the end, allow a proper definition of *who* and *when* the access and modification of each piece of information is authorized. To accomplish such level of security in our base scenario, the DBA is expected to configure the whole set of existing elements available in the system, having three key goals in mind (Said 2009): 1) • apply and configure security mechanisms that guarantee that the existing security policies and rules are enforced to the maximum extent possible; 2) dissuade attempts to break the rules; and 3) maintain mechanisms that help identifying potential violations of the rules, including being able to pinpoint suspects (in order to support *punishments* and avoid additional attempts).

The challenge is, therefore, to determine the following: 1) what are the *security elements* (mechanisms, processes, configurations, procedures and behaviors), in the form of *security recommendations*, that have to be put in place to accomplish the identified goals? and 2) what is the relative impact of each element in terms of security? The problem is that, as usually happens with security aspects in complex scenarios, to date there is no known process or methodology to automatically deduce a complete list of these elements, and therefore field research and practice is the only option to accomplish the task.

4.3.2.1 Identification of Security Recommendations

A very important requirement for our benchmark is that it must be independent of specific components brand (to allow portability); for instance, independent of any particular DBMS engine or operating system. Therefore, the analyzed security elements should come from different sources and not be tied with the restrictions of specific software. At the same time, the list as to include a comprehensive and realistic set of practical security recommendations, based on existing and consensually accepted security practices and mechanisms that can be used in real situations, without the requirement of special conditions (e.g. considerable additional money or time/effort).

Unlike in the case of security threats, there is an enormous quantity of security recommendations for databases and infrastructures in the form of books, reports, papers, manuals, etc. available for free in the literature. However, due to the complexity and time needed to gather all this information, the collection of recommendations must be narrowed. In our case, we focused on two reliable independent sources: the Center for Internet Security (CIS) (CIS 2008) and the USA Department of Defense (Defense Information Systems Agency 2001). Like in the case of threat vectors, we consider that these sources provide a representative list of all the security recommendations that exist for the domain of transaction systems: CIS documents from a software perspective (drawing from all security mechanisms available in the most important DBMS engines used nowadays), and the DoD document from a higher level behavioral perspective. Nevertheless, we note that the list could be extended easily, although the time to execute the analysis would grow accordingly.

As mentioned before, CIS has created a series of security configuration (CIS Benchmarks 2012) documents for several commercial and open source DBMS, namely: MySQL, SQLServer 2000/2005, and Oracle 8i/9i/10g. These documents focus on the practical aspects of the configuration of these DBMS and state the concrete values each configuration option should have in order to enhance the overall security of real installations. Although CIS documents are indeed very useful, three key problems have to be noted:

- The goal is to show which values or procedures should be used when configuring the system and not to provide a way to assess the DBMS configuration in terms of security. Although CIS refers to these documents as benchmarks they are not explicitly designed for DBMS configuration assessment or comparison.

- Each document targets a specific DBMS version and the configurations and concepts cannot be easily generalized. Additionally, each document follows a different approach regarding the way settings are presented. For example, the level of detail is different from one document to another and the way recommendations are written also differs.
- Although there is a concise rationale in some cases, the general security problem that is being addressed by each choice is not clearly presented. This is a relevant problem as the DBA learns barely anything about what he is doing, which in the end prevents him from applying his own alternatives for the same goals. That may also stop the DBA from understanding the gains and dangers associated to each configuration option, keeping him from being able to assess configuration alternatives when new software is available.

The other document we used in our study is the Database Security Technical Implementation Guide, version 8, release 1 (Defense Information Systems Agency. 2001), developed by the Defense Information Systems Agency for use within the USA Department of Defense. This document contains a very complete series of mandatory and recommended requirements that the DoD employees must follow when installing a database in the department. Although it is a generic document applicable to any DBMS engine, it enforces a very strict set of requisites that clearly implement a policy defined by the US government, which therefore may make it incompatible with the requirements of database installations in general. Nevertheless, it is a very good and complete source of information on database security practices.

The first set of security recommendations, presented in Table 4.2 and interchangeably referred as *security best practices*, is based on the detailed study and subsequent generalization of the configuration settings stated in the set of CIS documents. For each recommended setting, we identified the security property being targeted and analyzed the value and procedure recommended. This allowed us, for the majority of the cases, to determine the more general security recommendation being addressed by each setting. Additionally, we counted the number of different configuration recommendations that could be classified as having the same practice as basis.

Table 4.2 DBMS configuration security best practices devised from the analysis of the CIS documents

#	SECURITY RECOMMENDATION (CIS)	# of Recommendations in CIS documents			
		M	O8	O10	S
ENVIRONMENT					

1	Use a dedicated machine for the database	1	1	1	28
2	Avoid machines which also run critical network services (naming, authentication, etc)	1	1	1	1
3	Use Firewalls: on the machine and on the network border	1	3	3	1
4	Prevent physical access to the DBMS machine by unauthorized people				1
5	Remove from the network stack all unauthorized protocols		1	1	1
6	Create a specific user to run the DBMS daemons	1	1	1	
7	Restrict DBMS user access to everything he doesn't need	1	4	4	3
8	Prevent direct login on the DBMS user account	2	1	3	3
INSTALLATION SETUP					
9	Create a partition for log information	2	1	1	1
10	Only the DBMS user should read/write in the log partition	1			
11	Create a partition for DB data	1	1	1	2
12	Only the DBMS user should read/write in the data partition	1			
13	Separate the DBMS software from the OS files	1	2	2	2
	Remove/Avoid default elements:				
14	»»»Remove example databases	1			1
15	»»»Change/remove user names/passwords	1	4	4	2
16	»»»Change remote identification names (SID, etc...)		3	1	
17	»»»Change TCP/UDP Ports		1	1	1
18	»»»Do not use default SSL certificates	1			
19	Separate production and development servers		1	1	
20	No developer should have access to the production server		5	5	
21	Use different network segments for production and development servers		1	1	1
	Verify all the installed DBMS application files:				
22	»»»Check and set the owner of the files	1	2	3	
23	»»»Set read/running permissions only to authorized users	4	18	22	14
OPERATIONAL PROCEDURES					
24	Keep the DBMS software updated	3		1	1
25	Make regular backups	1			4
26	Test the backups	1		1	
SYSTEM LEVEL CONFIGURATION					
27	Avoid random ports assignment for client connections (firewall configuration)		1	1	
28	Enforce remote communication encryption with strong algorithms	1	1	11	3
29	Use server side certificate if possible	1		1	
30	Use IPs instead of host names to configure access permissions (prevents DNS spoofing)		1	1	
31	Enforce strong user level authentication	2	6	8	4
32	Prevent idle connection hijacking		2	2	
33	Ensure no remote parameters are used in authentication	1	2	1	
34	Avoid host based authentication		1	1	
35	Enforce strong password policies	1	2		2
36	Apply excessive failed logins lock		1	1	
37	Apply password lifetime control		1	1	
38	Deny regular password reuse (force periodic change)		2	2	
39	Use strong encryption in password storage	3			
40	Enforce comprehensive logging	1	2	1	
41	Verify that the log data cannot be lost (replication is used)		2	2	1
42	Audit sensitive information		14	19	25
43	Verify that the audit data cannot be lost (replication is used)		1		1
	Ensure no "side-channel" information leak (don't create/restrict access):				
44	»»»From configuration files		2	1	
45	»»»From system variables	1			
46	»»»From core_dump/trace files		8	8	1
47	»»»From backups of data and configuration files		1	1	4
	Avoid the interaction between the DBMS users and the OS:				
48	»»»Deny any read/write on file system from DBMS used	2	3	2	
49	»»»Deny any network operation (sending email, opening sockets, etc...)		4	3	
50	»»»Deny access to not needed extended libraries and functionalities	1	11	11	54
51	»»»Deny access to any OS information and commands	2			
APPLICATION LEVEL CONFIGURATION AND USAGE					

52	Remove user rights over system tables	1	23	25	1
53	Remove user quotas over system areas		3	1	
54	Implement least privilege policy in rights assignments		9	10	6
55	Avoid ANY and ALL expressions in rights assignments	1	3	3	
56	Do not delegate rights assignments	1	3	3	3
57	No user should have rights to change system properties or configurations	3	4	4	2
58	Grant privileges to roles/groups instead of users		1	1	3
59	Do not maintain the DB schema creation SQL files in the DB server		1		
Total number of recommendations		**48**	**166**	**183**	**177**

The first column of Table 4.2 is a number that univocally identifies each security recommendation and the second is the recommendation description. The last four columns show the number of specific recommendations from each CIS document that was associated with each generic recommendation (or best practice). The column M is for the MySQL Benchmark document, O8 is for the Oracle 8i Benchmark document, O10 is for the Oracle 9i and 10g Benchmark document, and S is for the SQLServer 2000 Benchmark document.

There are three key aspects that deserve special attention regarding the procedure followed to identify the best practices presented in the table. The first is related to the cases where a given configuration setting can be associated with more than one general best practice. For example, in the CIS document for Oracle 8i, recommendation 1.32 states that the "tkprof" utility, used to access trace data, should either be removed from the system (which can be associated with the security best practice #50) or have its permissions reviewed in order to be available only to authorized people (related to security best practice #23). In these cases, field database administration experience and expert judgment were used to determine the prevalent best practice. For the previous example ("tkprof"), we have considered this recommendation to be related with best practice #50.

The second aspect is related to the configuration settings that are not clearly related to a generic security best practice (e.g. Oracle 10g recommendation 6.03 related to the Automated Storage Management, and SQL Server 2000 recommendation 5.4 related to the SQL Profiler application). We were able to observe that these recommendations are typically related to database management and not to security aspects, and therefore are not exactly suitable for our goal. Also, in many cases, they are applicable only to a particular DBMS and can hardly be generalized. That is the reason why the number of items in each column does not match the exact number of recommendations presented in the CIS documents.

The last noticeable aspect about the definition of the best practices is that some of them can be seen as special cases of more generic ones. The problem here is to decide when a specialization of a particular best practice is relevant enough to

spawn a new one. For example, best practices #48 and #49 may be seen as specializations of best practice #50. Practical experience on security trade-offs was then used to evaluate and decide when such separation was important. For the previous example, it is well known that network operations and access to the file system are extended functionalities that, although useful to some extent, represent potential sources of attacks and hence should be explicitly avoided. At the same time, a more generic practice related to other possible extensions and functionalities (as in practice #50) is also important. In fact, although in some cases it may not be possible to decide for sure if a given extension can or cannot be used as an attack path, the possibility frequently exists.

Table 4.2 is divided in 5 groups of practices that have common characteristics. This division is useful when it becomes necessary to focus in a given subset of practices related to a specific configuration step (i.e. installation, operation, application deployment, etc.). The groups considered are:

- **Environment**: recommendations related to elements surrounding the DBMS engine and the machine hosting it;
- **Installation setup**: recommendations to be considered right before and after the installation of the DBMS engine;
- **Operational procedures**: periodic operations related to the DBMS maintenance;
- **System level configuration**: the general working parameters recommended for the DBMS;
- **Application level configuration and usage**: recommendations that are application dependent.

In terms of the representativeness of the best practices, a brief analysis of Table 4.2 raises some immediate considerations. The first one is related to the fact that there are many recommendations that appear only in a subset of the CIS documents. This is mainly due to two reasons: on one hand, the documents are based on the empirical experience of different people, which results in different sensibilities of what are the most important security problems in each DBMS; on the other hand, the documents are focused on the configuration mechanisms and parameters *available* in each DBMS, meaning that whenever a particular feature is absent or not configurable in a given engine then it is not addressed in the corresponding document.

The absence of certain best practices in a given document should be considered a problem, even if they represent minor issues in the context of DBMS targeted by

the document. By being completely subjective and dependent on the environment, security assessment should always be an exhaustive task, despite of the DBMS considered. For example, the precaution related to not storing sensitive information in system variables is mentioned only in the MySQL document (e.g. best practice #45). However, this can be clearly a problem in any database environment, and should not be overlooked. This is one of the reasons why our complete list, which comes from the aggregation of all documents, represents a better approach than simply using a specific document to harden a specific engine.

Another case is when a specific feature is not available in a given DBMS. For instance, MySQL does not have auditing capabilities, so there are no recommendations related to auditing in the CIS document. However, it is easy to understand that auditing can be implemented, to a certain extent, using other DBMS features like *triggers* (Da-sheng 2010). The important issue to be focused is not to "have auditing turned on", but instead to have ways of tracking operations done on the system (e.g. trigger based auditing).

Another aspect that can be noticed in Table 4.2 is that some recommendations have a highly variable number of configuration settings across the four documents (e.g. best practice #1). That is a natural consequence of the fact that different people designed the documents. Thus, it can be seen as a side effect caused by the differences of how fine-grained the recommendations are.

The total number of recommendations in each document (last line of the table) also shows an interesting aspect. Even though the commercial DBMS engines considered (Oracle and SQLServer) have a quite similar number of recommendations, the open source one (MySQL) has significantly less. This is understandable as the number of configuration settings presented in the CIS documents is obviously related to the number of functionalities and configuration options available. MySQL is an open source DBMS that provides a reduced set of functionalities when compared to more complex DBMS like Oracle and Microsoft SQLServer (this is a result we obtain beyond any doubt in Chapter 6).

After identifying the set of security elements based on the analysis of the CIS documents, we turned to the second source: the DoD document. As we already had an initial table of security elements, our goal was then to screen the document looking for things that were not yet included in the list. After a very careful analysis, we were able to find only a small number of complementary recommendations that did not show in any of the CIS documents. All other advices in the DoD document can be generalized as at least one of the CIS related best practices shown in Table 4.2. The new best practices and corresponding groups are presented in Table 4.3.

Table 4.3 Complementary DoD best practices

#	Complementary Best Practices (DoD)	Group
1A	Monitor de DBMS application and configuration files for modifications	Operational Procedures
2A	Do not use self signed certificates	System Level Config.
3A	Protect/encrypt application code	Appl. L. Config./Usage
4A	Audit application code changes	Appl. L. Config./Usage
5A	Employ stored procedures and views instead of direct table access	Appl. L. Config./Usage

Following a Defense-in-Depth approach, all the 64 security recommendations presented in Table 4.2 and Table 4.3 were selected as the set of security recommendations for our base scenario (and will guide the rest of the benchmark definition). Obviously, we are aware that the process employed to create this set carries out some limitations namely:

1) *It may become outdated when technology advances*. This is true for almost all aspects related to security, and most of all for practical and useful security tools. As technology advances, attack techniques also change and a set of recommendations that is enough in one time may become deficient in the future.

2) *It may be incomplete*. We tried to the best of our knowledge to identify additional sources of security information that would provide more security recommendations for our base scenario, and we are aware that additional sources of information exist. For example, the documentation of most of the DBMS engines (usually several hundred pages of technical documentation) includes security information that is spread within the text in the form of configuration suggestions. Academic books about database administration and a very high number of research papers (some not focused on security) also contain security information that might complement our practices. However, the process of screening and evaluating such a high volume of disperse information is beyond the ability and the goal of a PhD work, being more suitable for a targeted research effort accomplished by several researchers simultaneously. This way, we decided to focus on a smaller, but more precise, set of documents, knowing that this may leave out some important aspects. Nevertheless, the incompleteness of the list presented in this thesis does not invalidate the methodology used to create it, nor it diminishes the process used to conceive and design the proposed trustworthiness benchmarking procedure.

4.3.2.2 Impact of Security Recommendations

Although the identification of the security recommendations is the most relevant part of the process, we need to take into account that some of them are more effective than others in terms of their contribution to the reduction of the attack surface of the system. We may see this effectiveness as "how critical" it is to have the recommendation implemented. Defining this value, however, is not an easy task, as the security perception regarding the impact of any mechanism not only varies from one person to another, but also may depend on the target environment (e.g. the lack of communication encryption with the DBMS is only a concern if the business application is executing in a remote client, which is not the case if it is executing within a web server on the same physical machine, in which case the web server itself would be in charge of encrypting the communication). Additionally, although security recommendations can be identified from sources like books, forums, checklists, etc., their impact and contribution to the reduction of the attack surface is typically not addressed or is unclear. The representativeness of our benchmark would be compromised if this aspect was not taken into consideration.

The process followed to incorporate the impact of each recommendation into the benchmark was based on the definition of weights, drawn for the consensual judgment of several experts. In this sense, the diversity of experiences becomes a relevant issue, and experts from different fields should be explicitly included (and not only security experts), including: database administrators, database application developers, operating systems experts, network specialists, etc. Ideally, this group should include a large number of both practitioners and academics. The expectation is that, in average, the most important practices are emphasized, even if there is no unanimity (this average should be representative of the reality taking into account a base scenario).

Interviewing experts to obtain the importance of security recommendations is a complex problem, and there are a few caveats we have to consider. As we want to capture the most of a person's experience and knowledge, the scale used for the classification needs to be well defined, easy to understand, and include a short (but adequate) number of values. For example, an excessively detailed scale with 20 different values forces the expert to make irrelevant considerations to decide between close values (e.g. deciding between an 15 and a 16 is very difficult and may be irrelevant), and makes the weighting process a lot harder without gaining much from it. On the other hand, a too vague scale (e.g. with 2 values) does not allow distinguishing and expressing the notion of importance of different recommendations. In this work, we use a scale with four values (from 1 to 4), with a very specific semantic for each one (the reason why use an even number of values

is to avoid falling into the "*select the middle-one*" syndrome, i.e. when in doubt select the middle value). The description we created for each value is intended to induce the interviewee to ask himself the following question: "*how preoccupied would I be if the system I manage did not have this feature/security element implemented?*" The semantics and scores we used are presented in Table 4.4.

Table 4.4 Best practice impact key

Score	Importance to the system
4	Critical to the system
3	Important
2	Advisable to implement
1	Not much relevant

Having decided on the scale, we designed a spreadsheet and handed it to the nine experts we invited to participate in our evaluation. We asked them to assign a score to each recommendation using the keys presented in Table 4.4. This group of experts included five people from academia and four engineers from industry. From the academics, three are professors in a university (two of them teach databases courses and the third one teaches a security course), and two are PhD students (one working on intrusion detection and security vulnerabilities emulation and the other working on security benchmarking for web servers). In the engineers group, we have three full time database administrators and one technical manager for the databases area in a medium size company.

The *individual weight* of each security recommendation is computed as the sum of all scores assigned by each expert, normalized to a logarithm scale (base 10). This normalization tries to stress the difference of the scores, highlighting and distinguishing recommendations found critical even by a small number of experts (the idea is to differentiate these from the ones that no expert found critical). The *final relative weight* (which is a percentage) of each security recommendation is defined as the *individual weight* of the recommendation divided by the sum of all *individual weights*.

The summarized relative weights are shown in Table 4.5. The recommendations presented in the second column of each row (see tables 4.3 and 4.4 for the correspondence between the numbers and the description of the practices) are ordered by the computed weights, and have a relative importance in the interval presented in the second column. For example, all the practices presented in the third row of Table 4.5 (Class 2) have a relative weight between 1% and 2.5%.

Table 4.5 Best practices ordered by relative weights

Class	Weight (W)	Ordered Recommendations (all 64 practices)

4	5,26 % > W ≥ 4%	4, 3, 19, 28, 57
3	4 % > W ≥ 2,5%	2, 24, 39, 35, 15, 1, 6, 52, 25
2	2,5% > W ≥ 1%	20, 23, 18, 31, 8, 29, 51, 32, 36, 54, 33, 37, 10, 12, 42, 41
1	1% > W ≥ 0,15%	22, 34, 5, 48, 21, 47, 38, 55, 46, 50, 7, 44, 45, 49, 26, 40, 43, 9, 4A, 11, 17, 13, 56, 30, 1A, 53, 58, 27, 2A, 14, 5A, 16, 59, 3A

From the analysis of the detailed results (which can be found in Annex A) it is clear that each recommendation typically falls into one of four distinct groups: 1) the ones that are unanimously critical, 2) the ones that are not critical but are important, 3) the ones that are advisable to implement, and 4) the ones that are unanimously not relevant. This is very interesting and can be seen as a guide of which best practices should be implemented in a system according to its criticality. For example, consider three database infrastructures: one for a business critical application like a bank, another for an important application like the human resources database in a small company, and the last one for a non-critical application like a web portal that disseminates information about cultural events. It is clear that a database in a bank needs to implement all best practices, including the less important ones; the human resources database should implement the best practices in the three first groups (the critical, important and advisable groups) and may relax the less important ones if their implementation brings unaffordable costs; and, finally, the less important database needs only to implement the best practices in the two first groups (the critical and important groups) and may relax the others.

A very important observation is that the 14 most important recommendations account for exactly 51.61% of the security impact of the whole set of recommendations, while the other 50 best practices account for less than half that same impact. This is a major aspect that shows that there is a subset of the best practices that is unanimously considered as important for any DBMS installation. These 14 practices are the ones presented in the first and second rows of Table 4.5.

4.3.3 Pessimistic Scenarios

Having established the set of security recommendations (including a consensual *relative impact weight*) and the set of threat vectors relevant for our base scenario, we need to establish a relation between both. This relation specifies the threat vectors that are affected, directly or indirectly, by the implementation of the recommendations, and will therefore allow evaluating the relative contribution of each recommendation in preventing each threat from turning into real attacks.

The key problem in establishing such relation is that, on one side, we have static configuration characteristics of the environment (i.e. the security recommendations) and, on the other side, we have the high-level *bad, malicious or*

otherwise undesirable effects and circumstances that are considered harmful whenever they occur in the infrastructure. In other words, the main difficulty of this analysis arises from the fact that real attacks or events (corresponding to particular threat vectors), may depend on other conditions that have little to do with the static characteristics of the environment. For example, how do we evaluate the security problems that may arise when an infrastructure does not implement recommendations, such as "*separate development and production platforms* or *use a dedicated platform for the DBMS engine"*, without taking into consideration a real environment configuration (and therefore no considering attackers or using business rules to differentiate a security breach from a normal usage of the assets involved)? At this stage of the benchmark design, we need a methodical reasoning process that allows identifying the connection between both sets (i.e. threat vectors and recommendations).

As mentioned before, security recommendations (like the ones proposed in Section 4.3.2.1) are typically provided by security experts and experienced practitioners in the form of procedures and state configurations that are consensually accepted as having the ability to make a system or environment more secure. However, this assumption also implies the opposite consequence: if the recommendations help making a system more secure, then, by definition, their absence can always be associated with a particular insecurity circumstance. Therefore, if our list of security recommendations is complete, then the list of insecurity circumstances yielded that can be drawn from the pessimistic scenario and the absence of each of the recommendations will be also complete.

4.3.3.1 Mapping process

Taking advantage of this reasoning, we propose the following methodology for establishing the relation among threats and security recommendations (to better understand the process, see example in Section 4.3.3.2):

1. For each security recommendation, identify a situation where not following the recommendation creates an obvious vulnerability (in practice, a situation where the recommendation is in fact the last layer of defense, and where the associated insecurity would not exist if and only if the recommendation was enforced). In this work, we refer to these situations as *pessimistic scenarios*. They are pessimistic, in the following sense: although neglecting a recommendation may not necessarily lead to attacks, in this pessimistic circumstance it would certainly do. In other words, neglecting the security recommendation degrades the security of the infrastructure in the perspective posed by the pessimistic scenario.

2. Starting from each pessimistic scenario, identify any concrete attacks that could exploit the related vulnerability. These attacks should have harmful effects that may be correlated with the threat vectors. The reasoning is that, whenever a vulnerability in a pessimistic scenario and a threat vector are related by an attack that exploits the related vulnerability and instantiates a threat, a correlation between the original security recommendation and the threat vector can be established.
3. While evaluating the attacks allowed by the pessimistic scenarios, it is important to recall that real individuals (and attackers) interact with the system by using one of the four interaction classes defined by our base scenario (i.e. application *userid*, OS *userid*, DBMS *userid*, or none). This is important because, to properly identify the plausible attacks, we need to know *how much access an attacker already has inside the system* and that is defined by the environment where the attacker is contained (i.e. the interaction class he is using). Different interaction classes may allow different attacks according to the different privileges associated to each class.
4. This process, when completed for all pessimistic scenarios, for all interaction classes, and for all threat vectors, generates a list of attacks that identifies the trustworthiness relationship between each security recommendation, each interaction class, and each threat.

It is important to emphasize that *pessimistic scenarios are not the simple negation of a security practice*. The absence of a security practice simply shows that the administrator is not fully aware of the potential system states in terms of security. A *pessimistic scenario*, however, is the definition of a system state where neglecting the associated practice derails into an obviously insecure circumstance that can easily be associated with actual security attacks (this notion is important as it supports the specific reasoning step that allows aligning the benchmark being proposed in this chapter with the definition of trustworthiness benchmarking presented in Chapter 3). In practice, by neglecting consensually accepted security recommendations, the administrator of a transactional system infrastructure is allowing for a certain set of circumstances to become possible, and therefore increasing the probability of a set of attacks to happen. The harmful effects that consequentially have their probability raised are defined by the consequences of each possible attack allowed by a specific pessimistic scenario. How much each probability is raised is given by the *relative weights* of the recommendation from which the pessimistic scenario was identified (which is related with the consensual impact identified for its implementation, as defined in Section 4.3.2.2).

We finally point out that, even if these probabilities give no guarantees that the system can be attacked, *they provide necessarily some evidence that the system cannot be trusted to be secure*, which is exactly what we are proposing to measure (see discussion on metrics in Section4.3.5). In order to better illustrate the process, next section presents a complete example detailing the reasoning behind each step.

4.3.3.2 Mapping example

Consider the security recommendation "*separate development and production platforms*". In practice, what this recommendation defines is that the database server used by the developers to develop and test applications should not be the one that hosts the production data. Elaborating on the opposite of this recommendation allows us to identify a pessimistic scenario where developers have the ability to execute untested and under development code on the production database. Note that if testing and production platforms were in fact independent, then any code could be tested thoroughly before reaching production data, which may not happen when both environments coincide (the goal of the recommendation is to prevent users with access to the development infrastructures from executing malicious or potentially destructive code in the production environment).

Given the pessimistic scenario "*developers can execute code in the production DBMS engine*", we need to analyze the threat vectors from the point of view of the 3 interaction classes (DBMS *userid*, OS *userid*, application *userid*) and also from the point of view of non-system users. First consider the *Legitimate excessive privilege achievement* vector (see Table 4.1). For each interaction class we should ask if the scenario enables "*an increase of the probability of a user legitimaly obtaining more privileges than he should have*". Clearly, a malicious code injection is not a legitimate way for obtaining more privileges, so there is no mapping between the best practice ("*separate development and production platforms*") and the vector ("*Legitimate excessive privilege achievement*").

Let's now look to the second vector ("*Illegitimate privilege elevation*") and assess if the scenario enables "*an increased probability of an user obtaining an arbitrary privilege that he should not have in any circumstances*". As a relation seems to exist, each of the four interaction classes should be analyzed individually:

1. From the point of view of *non-system users* the answer to the question is yes, as code injection may be used for bypassing authentication, allowing a non-system user to access private data.
2. From the point of view of an application *userid*, the answer is also yes, as the malicious code injected could bypass privilege checks, augmenting the current *userid* privileges.

3. From the point of view of an operating system *userid*, the answer is no, as it is not possible to elevate OS privileges by executing code in the production DBMS engine.
4. For a DBMS *userid* the answer is yes again, e.g. if code injection is performed over a stored procedure meant to control operations over tables. For a particular DBMS *userid*, the stored procedure could then behave in a malicious way and allow increasing privileges.

This way, the final mapping of the security recommendation "*separate development and production platforms*" into the "*Legitimate excessive privilege achievement*" threat vector is not possible, while into the "*Illegitimate privilege elevation*" is possible for three interaction classes: application *userid*, DBMS *userid* and non-system users. The same process should be repeated for all remaining threat vectors and security best practices, resulting in a three dimensions matrix relating security recommendations, threat vectors, and interaction classes, as discussed in the next section.

4.3.3.3 Complete mapping

The process of relating the 64 security practices identified in Section 4.3.2 with the eight threats presented in Section 4.3.1 is extremely complex to be executed correctly, and actually not suitable to be executed by a small number of researchers. With the help of several database administrators and security researchers, we performed the complete process for the fourteen most important practices identified Table 4.5 (which already account for more than 50% of the identified impact). The fourteen pessimistic scenarios devised are presented in Table 4.6.

Table 4.7 presents an excerpt of the complete correlation of pessimistic scenarios with threats (as the complete matrix is too extensive, it is presented in (PhD Thesis Complementary Info 2012)). The attacks presented in the table are preceded by one or more of four acronyms, stating that the attack presumes a given interaction class: A – Application *userid*; D – DBMS *userid*; O – Operating system *userid*; and N – Non-system user. Recall that these interaction classes do not map directly to real individuals, and it is expected that some roles need more than one interaction class. In particular, any real individual (including the ones that have *userids*) can accomplish attacks that require no relation with the system (identified by the N acronym). For instance, the attacks under the pessimistic scenario #1 can be accomplished by anyone able to achieve a physical proximity with the machine, and that has the knowledge needed to carry out the actions indicated (e.g. rebooting the system with a live CD for an illegitimate access to the file system).

Table 4.6 Pessimistic scenarios associated with not following security recommendations.

#	Pessimistic scenarios
1	The platform is physically stationed in a place where people that have nothing to do with the DBMS have regular unsupervised access
2	a) The platform does not have an operating system firewall, leaving all locally open ports accessible to the local area network
	b) The network does not have a firewall separating the internal network (LAN) from the servers that provide services to the Internet
	c) The network does not have a border firewall, leaving all network fully accessible to internet traffic
3	a) The development DBMS is installed in the same platform as the production DBMS, but use different DBMS instances with separate data and configurations
	b) The development DBMS and the production DBMS are the same, and are only set apart by privileges within the database
4	Remote communications with the DBMS can be very easily captured and understood (no encryption)
5	a) DBMS *userids* can alter or influence the DBMS environment and behavior
	b) OS *userids* can alter or influence the operating system environment and behavior
6	The DBMS platform also hosts a email, naming or similar critical network service which is completely open for access from the Internet
7	The DBMS has known critical vulnerabilities which are of public domain knowledge
8	Stored password information in the database is clear text
9	DBMS/applications/OS users may choose any password they like, even the most easy to guess ones
10	Information of one username/password pair that can be used to login in the database is public domain
11	a) The operating system of the DBMS loads several unknown default services on the boot process, which may open listening ports on the server and may contain security vulnerabilities
	b) The operating system of the DBMS have several applications and tools installed on the file system, which may be used by an operating system user as leverage to an attack (like a compiler, for instance)
12	a) The OS *userid* used to run the DBMS daemons has administrator's privileges
	b) The OS *userid* used to run the DBMS daemons is used for other daemons and tasks as well
13	DBMS *userids* have privileges to access internal control information, and may alter the DBMS engine behavior
14	a) There is no regularly updated copy of the production data in a separate storage
	b) There is no regularly updated copy of platform file system and important configurations in a separate storage

Table 4.7 Set of attacks correlating the pessimistic scenarios and the threats

#	Legitimate Excessive Privilege Achievement	Illegitimate privilege achievement	Denial of service	Communication Weakness
1	D: User can bypass application/network level restrictions, by logging directly to the database, as long he can login to the OS	N: Boot by a CD/USB pendrive, copy all file system	N: Disconnect cables, turn off the server or simply destroy it physically. Each of those actions can be intentional or not.	N: Install a sniffer physically in the network adapter
2a	D: User can bypass application level restrictions and connect directly into the DB through a DB client	N: A LAN user connects to a vulnerable local listening service, causes a buffer overflow allowing arbitrary code execution	N: A LAN user connects to a local listening service and causes it to consume all CPU resources	
2b		N: An attack on a server with internet applications may be used to launch another attack on a private network host, achieving access to all computers on the network, including the DBMS platform		
2c		N: Internet port scans are free to find servers with vulnerabilities and which can be used as leverage to other attacks	N: Internet users may request all kinds of connections to any ports in any network server, flooding the network and hogging resources	N: Local network may be flooded with invalid requests consuming all Internet bandwidth
3a	A, D: Development and testing may cause effects on the behavior of the production applications			O: Developers may be able to eavesdrop production connections
3b	A: Untested applications can mess with production server resources and data	D: Execute malicious stored procedures may read or write over production data	A, D: Activated malicious code may erase information O: A system command may consume all CPU resources	O: Developers may be able to eavesdrop production connections
4		N: LAN users may have access to the data transit O: OS users may sniff all traffic from the network interface		N: LAN users may have access to the data transit O: OS users may capture all traffic from the network interface
5a			D: DBMS users may modify the size of working areas as to not allow correct operation	
5b		O: OS users may alter environment variables that affect the DBMS startup or behavior	O: OS user may modify memory configurations affecting availability	
6		N: Buffer overflow in the offered service, taking control of the machine and the DBMS	N: Overuse of the offered service, causing CPU or disk exhaustion	N: Buffer overflow in the offered service, taking control of the machine installing a packet sniffer

Table 4.8 presents the complete mapping for the fourteen most important security recommendations. It is important to emphasize that we are aware that this mapping is most probably incomplete, as it is very hard to envision all the ways security can be affected in such a complex environment, even when considering a focused approach with several assumptions (like the ones defined in the beginning of Section 4.1). The benchmark needs to be completed and perfected in an incremental way, by integrating knowledge of more and more experts and by incorporating the new attack information that becomes available. Nevertheless, we believe that even such an incomplete mapping can be used to implement a fairly representative benchmark, thus allowing demonstrating the effectiveness of our trustworthiness benchmarking approach.

Table 4.8 Mapping for the fourteen most important security recommendations

#	Legitimate Exc. privilege achievement	Illegitimate privilege achievement	Denial of service	Communic. Weakness	Authentication Weakness	Side-channel Data Exposure	Audit Trail Weakness	SQL Injection Enhancement
1	D	N	N	N	N	N	N	
2	D	N	N	N	N			
3	A, D	D	A, D, O	O	N, A			
4		N, O		N, O	N	N, O	N	N
5		O	D, O					O, D
6		N	N	N	N	N	N	N
7	A, D	N, D, O						
8	O, D				N, D, A, O		N, D, A, O	D, O
9		N			N, D, A, O		N, D, A, O	
10	N, O	N, O			N		N	N
11		N, O	N, O	N, O	N, O	N, O	N, O	N, O
12	O	N, O				O	O	N, O
13	D	A	D					D, A
14	A, O		O, D, A					

4.3.4 Benchmark Procedure

In the previous sections we discussed the reasoning and justifications behind the internal assumptions and the design of the proposed trustworthiness benchmark. A key aspect that has to be addressed when proposing a benchmarking procedure is the practical use of the benchmark (i.e. the steps required for executing it). Any benchmark specification has to include a set of deterministic operations or procedures that, when carried out by benchmark user (which we assume to be the administrator of a transactional system infrastructure), allow the computation of the metrics (Grey 1993).

A key aspect is that a benchmark is expected to be repeatable, at least in a statistical basis, and should depend the least possible on external variables (Grey 1993). Typically, a benchmarking process based on the simple execution of a deterministic software application (or a set of applications) ensures the "ideal" means for obtaining correct results. Unfortunately, in our case the person that executes the benchmark is, by definition, an external variable, as the input of the benchmark is the user perception about the status of the implementation of the security recommendations (which cannot be obtained in an automated manner). In fact, the high complexity and variability of the systems and environments targeted by the

proposed benchmark, allied with the complexity of the semantics of the security recommendations, makes it unfeasible to create a program able to perform this assessment automatically for all cases, even if for a small number of recommendations this could be accomplished. For example, identifying the privileges of the existing *userids* is something that is trivial to automate, while at the same time, identifying if the physical hardware that hosts the database is adequately protected is not so easy.

In practice, as we cannot avoid having information gathered by a person, at least should prevent, to a certain extent, the security knowledge and biases of this person from affect the benchmark results. This can be done by focused only on *technical details and procedures* and in *the configuration state of the system*, as these elements are usually so evident that, assuming a competent and knowledgeable benchmark user, their identification would be unambiguous and independent of any particular previous knowledge.

Taking into account these restrictions, the proposed benchmark is based on a non-automated process that tries to minimize the human factor. In practice, the benchmarking tool consists of a list of deterministic tests, in the form of *yes* or *no* questions that depend exclusively on palpable characteristics of the environment and on the procedures applied to the systems. Examples of the tests are presented in Table 4.9, while the complete list can be found in the Annex A.

Table 4.9 Benchmark security tests (sample)

#	Test	Fail
1	If the machine is turned off, does any service other than the database become unavailable? Is there any process running on the machine which is not demanded by the DBMS, the OS or the machine maintenance/security?	Yes
2	If the machine is turned off, does any critical network service, like naming, directory or authentication services, becomes unavailable?	Yes
3	Is there a firewall on the network border? Is there a firewall running on the DBMS machine? Are both firewalls properly configured by experienced staff with solid network knowledge? (Wool 2004, Kaufman 2002)	No
4	Is it possible to an unauthorized person to physically access the machine without supervision at any given time?	Yes
5	List the protocols available in the network stack in the OS of the DBMS machine. For each protocol, is there a clear justification for its availability?	No
19	Is there any kind of development or testing being done in the production server?	Yes
25	Is a carefully thought out, documented backup procedure regularly executed? If the person in charge suddenly quit, is it easy for anyone else to resume its task?	No
32	Establish a connection with the DBMS and let it stay idle. Is the connection severed in a reasonable amount of time?	No

The benchmark tests should be answered by an experienced DBA with deep knowledge about the operating system in use, and some knowledge about computer networks. For some of the tests, however, there are variable parameters that cannot

be easily predicted, and may require the security knowledge of the user to be correctly determined. Such parameters are identified using the figures *security expert* and *experienced staff*, the later also assuming deep knowledge about the usage of the underlying infrastructure. In practice, input coming from professionals that do understand security is required to pass some of the tests (i.e. to have a *yes* answer). In these cases, to simplify the work, we provide references to bibliography where such security knowledge can be obtained (e.g. in test #3 we provide solid references to information regarding the correct configuration of firewalls).

Two other figures that appear in the tests, *reasonable* and *regularly*, also depend on bounds that cannot be defined without taking into account the business applications that are using the database (e.g. in test #32 we have to define a reasonable time for a timeout, which clearly depend on the application in question). In these cases, we expect the DBA to either estimate those bounds or to discuss them with the system analysts and other experts. We could have provided average values for these cases, but obtaining this information would require detailed field studies that were not in the scope of this work.

As can be observed in the second column of Table 4.9, the tests typically include two steps. The initial step, which is not defined for every case, is a procedure to obtain the particular information necessary to answer the test (e.g. in test #32 we indicate a procedure that will provide the timeout configuration to the benchmark user even if he does not know what a timeout is or where this information is configured). This step is also of optional execution, in the sense that the DBA might obtain the same information in alternative ways (e.g. technical manuals or previous experience). The second step is a series of yes/no questions that should be answered systematically. <u>If, for any of the listed questions, the answer is the one stated in the rightmost column of Table 4.9, then the test is considered as failed</u>. Also, in some cases, the benchmark user might not know how to answer a particular question, which is an "unknown" answer that should be treated as a failed test (we follow a pessimistic approach, as one cannot trust in if there are some unknown aspects). In this case, the user is expected to further investigate in order to better understand the current state of the system.

Although for some recommendations designing the tests is a straightforward task, for others this brings two key difficulties. The first difficulty is related to knowing if the test really covers all aspects of the recommendation implementation. This is tricky due to the specificities of each scenario and is widely dependent on the generality of the best practice statement. For example, test #25 is designed for a recommendation that states that regular backups should be made. However, checking whether the DBA is in fact accomplishing this practice correctly is

something that cannot be done by means of only two complementary questions. In this case, we heavily assumed that a backup procedure that is not documented and that cannot be quickly understood by anyone other than the DBA would not be a reliable backup procedure, and therefore the test should fail..

The second difficulty is about how easily it will be for the benchmark user to perform the tests. This problem does not have an obvious solution, and it is possible that, depending on the case, the administrator might not have enough knowledge to execute some of the tests. As mentioned above, we suggest these cases to be treated as failed tests, meaning that if the administrator does not know whether a given security practice is implemented or not, then he should assume it is not (i.e. should follow a pessimistic approach). As a matter of fact, this is very much expected to happen with non-security experts: either they do not understand what they should do to improve security or they were never called attention to a particular aspect. In fact, having an administrator that does not know if a certain configuration is in place or not can already be considered a security risk, even in the cases where the system is correctly configured. As a consequence, by applying the tool, he will have the benchmark user attention redirected to the configuration aspects that experts believe are more important to improve security in such an environment (which will therefore increase the trust the user can put in the configuration).

4.3.5 Benchmark Metrics

The main goal of a benchmark is to allow *comparison*, and that requires the existence of *metrics*. In the previous sections we discussed and analyzed the steps required to build a body of knowledge, whose goal is to allow the calculation of a set of metrics that can be used for comparing the trustworthiness of transactional systems infrastructures. This section presents a deeper discussion regarding the benchmark metrics, including the algorithm needed to compute them.

As mentioned before, the metrics are represented as a percentage that should be interpreted as the relative proneness of the *bad or harmful effects* of the threat vectors to manifest. These percentages arise from the analysis of several characteristics of the system that may allow, given certain events, the emergence of circumstances equivalent to the pessimistic scenarios identified. The characteristics we are concerned with are the lack of rigorous enforcement of the set of security recommendations identified for our base scenario. The main assumption is that if these recommendations are not enforced, then *the system cannot be trusted as being protected against the bad effects of the threat vectors effects.*

The process that leads from the analysis of the state of the system to the metrics that express *justified trustworthiness* is entirely based on our definition of trustworthiness benchmarking, and as such is based on the *collection of evidence to place justified trust* instead of on the identification of actual vulnerabilities that can be exploited. As discussed in Chapter 3, within our framework actual vulnerabilities are considered during the *security qualification* step. In summary, the benchmarking process is based on the following assumptions:

a) The lack of active security precautions may let the environment derail into a configuration state equal or equivalent to pessimistic scenarios.

b) Assuming the pessimistic scenarios as representative, the only elements that prevent the occurrence of attacks are *intention* (which we assumed that exists) and the achievement of some other indeterminate requirements (e.g. physical proximity to the server or the opportunity to connect a computer to the same network segment of the DBMS server).

c) As the two requirements mentioned in item b) depend on the environment, we assume that there is a non-zero probability of them to happen.

d) Given an attacker with *intention* and given the right circumstances, the absence of active security measures in place allows actual attacks to happen with some undetermined, but non-negligible, probability.

e) Whenever two different security recommendations are related with the same threat vector and/or the same interaction class, and are both not enforced, we assume that they can be "accumulated". The reasoning is that they are *two independent alternatives* for accomplishing the same threat. In other words, if an attack related with security recommendation 1 has X probability of happening and another attack related with security recommendation 2 has Y probability of occurring, and if both attacks can take place independently, then we can safely say that the threat may be accomplished with a probability $Z > X$ and $Z > Y$, despite the real values for X, Y and Z. For practical reasons, in our benchmark we assume that $Z = X + Y$.

The proposed security benchmark for transactional systems infrastructures is able to compute 13 distinct trustworthiness metrics, namely:

a) one general metric summarizing the trustworthiness of the whole infrastructure.

b) eight metrics portraying the trustworthiness of the infrastructure in regard to the eight threat vectors;

c) four additional metrics characterizing the trustworthiness related with each interaction class.

The full algorithm for the computation of these 13 metrics is as follows:

1. The DBA executes the benchmarking procedure (as discussed in Section 4.3.4). The result of the application of the evaluation tool is an answer of *Passed* or *Failed* for each of the 64 security recommendations included in the benchmark.
2. Be W_{rt} the relative weight of the recommendation r that maps (i.e. has at least one identified attack) to the threat vector t. For each threat t compute:

$$\frac{W_{rt}(Passed)}{W_{rt}(Passed) + W_{rt}(Failed)}$$

where $W_{rt}(Failed)$ is the sum of the weights of all recommendations that map to the threat t and that had a Failed as an answer, and $W_{rt}(Passed)$ is the sum of the weights of all the recommendations that map to the threat t and had a Passed as an answer.
3. Be W_{ri} the weight of the recommendation r that maps to the interaction class i for any of the threat vectors. For each of the four interaction classes compute:

$$\frac{W_{ri}(Passed)}{W_{ri}(Passed) + W_{ri}(Failed)}$$

where $W_{ri}(Failed)$ is the sum of the weights of all recommendations that map to some threat with interaction class i and that had a Failed as an answer, and $W_{rt}(Passed)$ is the sum of the weights of all the recommendations that map to some threat with interaction class i and that had a *Passed* as an answer.
4. Compute the overall trustworthiness value by dividing the sum of the weights of all recommendations that had a *Failed* as an answer by the sum of the weight of all the recommendations.

A key aspect is that it is possible to increase the level of detail of the benchmark characterization by crossing each interaction class with each threat vector (computing 32 additional metrics). For example, we could specifically compute the trustworthiness related with the OS system users causing a denial of service in the infrastructure. This might be of interest in the cases where the administrator wants to assess the pros and cons of the trust he actually puts in the people that possess *userids* of each class against the costs of implementing new security precautions

(e.g., what is more cost-effective? To disable the operating system *userids* that were given to individuals that may not necessarily need them, or to implement the security best practices that raise the trustworthiness in this case).

Another variation that is semantically interesting is to consider a subset of the threat vectors in the computation of the interaction classes' values instead of all threat vectors. In this case, the result is the level of trustworthiness that one can put into the fact that some individual of that class may cause some kind of harmful effect. For instance, we could compute the metrics for the case of operating systems users causing either denial of service, obtaining privileged information through a side channel or taking advantage of an authentication weakness.

The algorithm presented above is based on the notion of (positive) *trustworthiness*, which expresses how much of the evidence gathered by the benchmark user supports positively the security of the installation. At the same time, we can easily do the inverse reasoning. The inverse of trustworthiness is called *untrustworthiness*, which computed as 1 – *trustworthiness.* The untrustworthiness metrics are exactly the values we would get if, in the algorithm above, we computed all the metrics relatively to the *failed* tests instead of the passed ones. Both trustworthiness and untrustworthiness are *trust-based* metrics in the sense that they express relative levels of justified trust (in one way or another).

It is important to notice that, even though trustworthiness is the numeric complement of untrustworthiness, the way security aspects should be reasoned about make the distinction of both these concepts quite important, especially when non-security experts are using these values to support decisions about their infrastructure. In fact, we have to pay attention to the fact that computing trustworthiness is based on a summary of the amount of evidences that justify how much one should trust the infrastructure. Conversely, if we are computing untrustworthiness, we are summarizing the amount of evidences that may lead us to not trust the infrastructure. When interpreting these values, however, we again face the fundamental assumption over which our benchmark is based on: *security has much to do with what we don't know about the system*. It is wiser, therefore, to interpret the metrics from a pessimistic perspective (as already mentioned several times), as the benchmark user has to be aware of the impossibility to always consider all aspects that are involved in the security of the system. For this reason, we decided that the main metric of our benchmark is the *Minimum Untrustworthiness*, which represents *the amount of evidence we have about how much we should distrust the system, at least.*

This definition of *Minimum Untrustworthiness* helps the benchmark user to understand the error that the metrics may have, particularly due to lack of information, which will more easily *lower the justified trust* than increase it. This approach looks arbitrary, as a typical standard error is usually considered symmetrical (Zwillinger 1995), but this effect comes as a consequence of the assumptions we made for our framework.

Let's examine in detail one example in which we try to demonstrate why a pessimistic view of security is always more correct than an optimistic view. To simplify the example, instead of a whole infrastructure, let's assume that we are benchmarking a small piece of software. A trust-based metric gives a certain value that represents how much we can trust that the software will not present security problems in the future. We also know that, in the context of our benchmarking framework, this metric takes into consideration only the characteristics that can be found in the software, excluding outside variables.

Now, let's assume that there is a very important and influential external variable: *community support and active development.* Assuming that for this particular software we do not know if there is an active community supporting development, this manifests in the metric as an error (i.e. the value reported will be incorrect because this information did not affect the metric). In fact, it is more or less obvious that *if there is an active community* then the trust we can put in the software is higher, and if *there is no active community*, then the trust we can put in the software is lower. But consider the following issue: is the error symmetrical? In other words, the existence of an active community should increase the metric as much as the lack of the community should decrease it?

The answer is no, and it is quite easy to understand why. The lack of an active community assures the following: new software bugs will not be quickly corrected; if a user of the software finds out a bug then the rest of the users have no way to be quickly warned; and for solving problems raised by a security incident the user will not have the help of any other experienced user or developer. The asymmetry of the metric comes from the fact that the mere existence of an active community does not guarantee an opposite result. The existence of an active community *does not guarantee* that software bugs will be quickly corrected, *does not guarantee* that security information found by users of this community will be quickly disseminated, neither guarantees that the users would get any kind of help in solving security incidents. In summary, improving security is a lot harder than decreasing it, and a trust-based metric should be interpreted considering this behavior.

We can also use the same reasoning in terms of the trustworthiness benchmark we are proposing to justify why *Minimum Untrustworthiness* provides the best semantic meaning. Assuming that the benchmark definition is correct, then an error in the metrics computation can essentially be due to two mistakes: a test that should have failed is reported as passed, or a test that should have passed is reported as failed. Let's then examine what happens in each case:

1. If the test was wrongly considered as passed, then the minimum untrustworthiness is correct because the real untrustworthiness should have been higher;
2. If certain test was wrongly reported as unknown or failed but the real configuration actually should have passed, this indicated that the benchmark user does not know or does not understand correctly the exact state of the system. Basically, he erroneously perceived one configuration as another configuration, and therefore he does not know the answer to the test. As unknowns are treated as failed tests, this error *does not change the value of the* metric. In fact, from a trustworthiness perspective, a test reported as unknown is always correct, as not knowing the state of a system is a lack of control that justifies less trust (even if the security of the system is in fact higher).

4.4 Case Study

The main goal of a benchmark is to provide information that allows making comparisons across different systems or different configurations of the same system. When comparing database infrastructures, however, we quickly notice that the idea of "selecting" one of a set of infrastructures does not appear to be much useful if we take the point of view of the DBA that is in charge of it. Instead, in a benchmarking context, his goal would be to evaluate the overall security state of his installation, in order to be able to improve it, even if that improvement would further imply being able to select alternative components for the system (e.g. the DBMS engine or the operating system). For this reason, we focused only on trustworthiness benchmarking in our experiments.

Selecting a secure software component of an infrastructure is an important problem that is discussed and addressed in Chapter 6. We point out, however, that this reality is changing, and with the appearance of database cloud services (Zhao 2012), effectively selecting a secure transactional system infrastructure among several alternatives is becoming a relevant problem that can be addressed with our methodology. For such cases, the base scenario would have to be adjusted, but the overall methodology would hold.

A security benchmark is also expected to provide information that helps administrators in further improving the evaluated system. This way, to evaluate our methodology, we have applied it to four real database installations, and thoroughly analyzed and discussed the results in terms of what information would the benchmark user really obtain from the benchmark.

The main input data that required by the benchmark for the computation of the metrics are the results of the tests that check whether the installation is in fact following consensual security recommendations (or not). Even though the main output of the benchmark is the set of trustworthiness metrics, the process of applying it already provides extremely useful information. Besides showing the validity of our proposal, we also intend to demonstrate this fact in the case study.

This section is divided in three parts. First, we show the main details of the four infrastructures we analyzed. Second, we take the results of raw tests to show that the benchmarking process, by itself, allows drawing some conclusions regarding the security of the installations (even before computing the benchmarking metrics). Finally, we compute the benchmark metrics and directly compare the infrastructures from the perspective of the security problems that they might have.

4.4.1 Systems Under Testing

The proposed trustworthiness benchmark has been applied to four real DBMS installations using four distinct engines. Table 4.10 presents the relevant details about each installation, including the DBMS engine used, the operating system running on the machine, the number of distinct applications using each database at the time of the evaluation, the number of distinct database administrators and the number of developers that are not administrators, along with the amount of time needed to execute the tests.

The tests were applied by one DBA of each installation, with the exception of Case 2 where two DBAs participated in identifying the answers to the tests. Two cases were evaluated under the direct supervision of the authors (Case 1 and Case 3) and the other two cases were done independently (Case 2 and Case 4). In these two cases, the users that performed the evaluation had only as basis a document that contained the list of tests (available in Annex A).

Table 4.10 Infrastructures details

	Case 1	Case 2	Case 3	Case 4
DBMS	Oracle 10g	SQLServer 2005	MySQL 5.0	PostgreSQL 8.1
OS	Windows 2003	Windows 2003	Windows XP	Windows 2000
Applications	3	54	3	2
DBAs	2	5	2	2

Developers	8	39	0	0
Test Duration	3 hours	1,5 hours	1 hour	1 hour

As we can see in Table 4.10, the scenarios have very different characteristics, which help in evaluating the benchmark portability. The differences start with the DBMS engines (which is different in all scenarios) and operating systems used (three different versions of the same brand). Also, two scenarios are based on free engines (cases 3 and 4) and two on commercial engines (cases 1 and 2). Most importantly, two scenarios have a fair number of developers, while in the other two the DBAs are also the developers. This is an important factor when deciding what threats are most relevant in each case as, for instance, we are not concerned with problems involving developers in the situations where there are none. All databases are used within an academic context in two different universities, being mostly utilized to support administrative processes that have university staff, teachers and students as end-users.

Let's start our discussion by analyzing the time needed to answer all the tests defined by the benchmark (i.e. the 64 tests). In the worst case (Case 1) the tests took about 3 hours of work, but the average time spent is slightly more than 1 hour for all cases. This suggests that the test set is not particularly burdensome and does not require too much work for an experienced DBA. Another interesting aspect is related to the comparison between the commercial DBMS and the open source DBMS. The DBAs evaluating the open source DBMS took much less time to answer the tests than the ones evaluating the commercial ones, and this becomes even more evident if we remember that in Case 2, which took 1.5 hours, two people cooperated in the process. We investigated the reasons for this and found out that the smaller set of security mechanisms provided by the open source DBMS allowed more easily identifying certain tests as *failed* (basically because the DBAs knew they did not have support for the operations stated in the test). The support offered by the security mechanisms available in the DBMS software is an important issue that is discussed in more detail in Chapter 6.

4.4.2 Analysis of the Results of the Tests

The first analysis we can do is related to the number of *passed* tests (that identify the number of security recommendations that are implemented in the infrastructure), the number of *failed* tests and the number of tests for which the DBA does not know the answer, the *unknown* tests. In this analysis we aggregated the results using the recommendations classification proposed in Section 4.3.2.1, and computed for each group an Impact Index, which corresponds to the relative weight (see Section 4.3.2.2) of all the passed tests of a group over the relative

weight of all the tests that are part of each group. This impact index shows how much of the security surface of each group is correctly protected considering the different impacts of each recommendation. The aggregated results are shown in tables 4.11, 4.12, 4.13 and 4.14, one for each infrastructure under testing. The analytical results for each test and each infrastructure can be found in Annex A.

Table 4.11 Case 1, Oracle 10g installation

	Tests Passed	Tests Failed	Unknown	II
Environment	6	2	0	83,89%
Installation setup	4	11	0	27,30%
Operational Proc.	1	3	0	34,76%
System level config.	16	8	2	55,53%
App. level conf./usage	7	4	0	92,07%
Total	**34**	**28**	**2**	**58,44%**

Table 4.12 Case 2, SQLServer 2005 installation

	Tests Passed	Tests Failed	Unknown	II
Environment	4	4	0	59,73%
Installation setup	5	9	1	30,43%
Operational Proc.s	2	2	0	85,56%
System level config.	12	13	1	39,20%
App. level conf./usage	3	8	0	50,84%
Total	**26**	**36**	**2**	**46,63%**

Table 4.13 Case 3, MySQL 5.0 installation

	Tests Passed	Tests Failed	Unknown	II
Environment	3	5	0	44,30%
Installation setup	7	8	0	35,66%
Operational Proc.	1	3	0	50,80%
System level config.	12	13	1	38,78%
App. level conf./usage	4	7	0	65,74%
Total	**27**	**36**	**1**	**43,07%**

Table 4.14 Case 4, PostgreSQL 8.1 installation

	Tests Passed	Tests Failed	Unknown	II
Environment	3	5	0	46,53%
Installation setup	4	11	0	26,02%
Operational Proc.	1	3	0	34,76%
System level config.	9	15	2	29,29%
App. level conf./usage	6	5	0	68,52%
Total	**23**	**39**	**2**	**37,21%**

The most important aspect we can observe in the results is that the number of unknown answers is very low (always below 2 for the 64 questions in any of the 4 cases). Test number #27, related to the range of ports that are used to connect to the DBMS, was answered as *unknown* in cases 2, 3 and 4, meaning that maybe it should be revised or better explained. However, the rest of the unknown cases are spread randomly through the test set, which suggests that they are probably due to either lack of experience of the corresponding benchmarking user or some difficulty imposed by the software on obtaining the information. Nevertheless, from a high

level perspective, the DBAs did not report any difficulties in applying the test set. Obviously, it is hard to generally assess the usability of a complex benchmark such as this one using only four assessments, but the conclusion we reach within this limited set of results is that the tool appears to have a high *usability*.

Concerning the analyses of the types of tests that were pass or not, an interesting result is the low number of passed tests in the *Installation Setup* group in all cases (always less than 50%). Three factors seem to contribute to these results: the default installation settings are kept and used (this may be exploitable as default settings are universally known), the inexistence of file system partition planning for logs and data (which can lead to Denial of Service by exhaustion of disk space), and the use of an operating system that does not provide easy ways to keep track of files permissions (that usually force users to use administrative roles for several tasks).

In terms of the 14 most important database security recommendations presented in Section 4.3.2.2, we list in Table 4.15.the critical practices missing for each case. Given the impact assigned to these recommendations, implementing them would have two immediate consequences: the total impact on the security surface of the infrastructure related to the overall set of recommendations implemented would raise to more than 50% in all cases; for the same reason, they would boost to the overall trustworthiness of the infrastructure.

Table 4.15 Most important best practices yet to be implemented

Case	Missing critical recommendations	#
1	19, 28, 24, 15, 6	5
2	3, 19, 28, 35, 6, 2	6
3	4, 19, 28, 35, 1, 6, 25	7
4	3, 19, 28, 24, 35, 1, 6, 2	8

As final analysis let's use the raw results of the benchmark from the point-of-view of the tests with unanimous results in all cases, as shown in Table 4.16. The analysis of the description of the security recommendations from which we devised these tests spots some patterns. For example, it is clear that tests #6, #7, #8, #10, #12, #13, #23 and #45 are heavily OS dependent. Thus the same outcome to all of them can be explained by the fact that all the infrastructures benchmarked use some version of the same operating system; thus it is plausible that by simply changing the operating system one could solve most of the issues. Furthermore, the following is true for the four infrastructures:

- Testing is executed directly over critical production data;
- No auditing is performed (even when provided by the DBMS);

- There is no policy about backup testing;
- There is at least a small list of privileges attributed directly to *userids* instead of groups/roles;
- No host based authentication is used;
- None of the DBMS engines has file system access functionalities enabled.

Table 4.16 Tests with unanimous results in all four cases

	# of tests with unanimous results
All cases passed	2, 8, 30, 33, 34, 39, 45, 47, 48, 57
All cases failed	6, 7, 10, 12, 13, 19, 23, 26, 28, 29, 32, 37, 38, 42, 43, 58

4.4.3 Trustworthiness Assessment

Trust-based metrics only make sense after security qualification, where the obvious attack paths are identified and vulnerabilities that can be easily discovered are mitigated. Also, the lack of any fundamental security mechanism is already accounted for. Security qualification of transactional systems infrastructures is a complex problem, as already discussed in Section 4.2, and further revisited in Chapter 6.

In this section we are concerned with understand the relative likelihood of the manifestation of harmful effects (the ones defined by the threat vectors) that may lead to attacks and vulnerabilities, basically by evaluating how prone certain security problems are. At this point, we assume that the security of the installations is at least at an acceptable level (i.e. higher than zero), and the goal is to distinguish in terms of their ability to prevent future security incidents or having hidden security vulnerabilities.

We start the analysis by inspecting the general values of the minimum untrustworthiness metric for each scenario. As presented in Figure 4.1, Case 1 is the least untrustworthy, while case 4 is the untrustworthiest. This, in general, means that more configuration problems (and more critical ones) are present in Case 4 than in all other cases. However, to obtain more information about the problems we must analyze the results from the point-of-view of the relevant threat vectors.

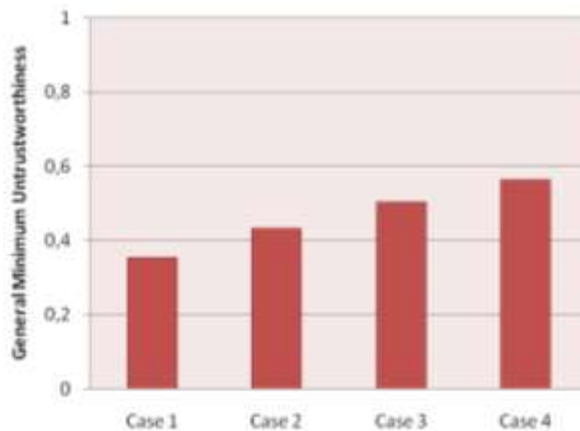


Figure 4.1 General untrustworthiness for each scenario.

Figure 4.2 presents the results of the minimum untrustworthiness metrics for each threat, grouped by case study. From an analytical point-of-view, there are several important trends in each scenario. Case 1 appears to be generally the least untrustworthy of all, and in particular, *Denial of Service* (DoS) is a threat that is very unlikely to actually be accomplished.

Legitimate excessive privilege achievement (LegExPrA), on the other hand, is the threat against which Case 1 is untrustworthier. Considering the fact that there are 8 developers in this scenario, they may end up achieving excessive privileges. In Case 2, the configuration is very untrustworthy against *Communication weaknesses* (CommW). This may be a serious problem as Case 2 has a very high number of developers and applications (that represent a high number of application users), and communication weaknesses can be used to eavesdrop data and authentication information. Case 3 strikes the eye as being very untrustworthy against *Side Channel Data Exposure* (SCDtEx). This may or may not be a problem, depending on the exact characteristics of the environment. In particular, by having no developers, this might not be a big concern for the DBA, which can also exclude *Communications weaknesses* (CommW) from his priorities. *Audit trails weaknesses* (AudTW), however, can be a problem, and *Denial of Service* (DoS) surely is. These observations can also be generally visualized in alternative presentations, as shown in Figure 4.3.

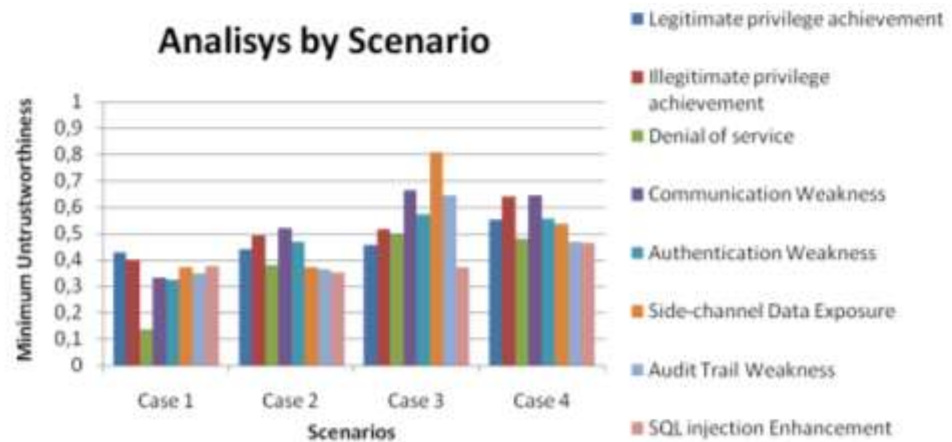


Figure 4.2 Untrustworthiness for each threat, grouped by case

Figure 4.3presents the same data of Figure 4.2, but in a way that allows easily comparing each case against the others when it comes to individual threats. On the left graph, the very small untrustworthiness against *Denial of Service* (DoS) in Case 1, and the extreme untrustworthiness against *Side Channel Data Exposure* (SCDtEx) in Case 3, are the two aspects that are highlighted. The radar graph presented on the right side of Figure 4.3 allows evaluating again the general prevalence of untrustworthiness on the different cases. It becomes clear that Case 1 has, in general, the least untrustworthy configuration, and that cases 3 and 4 have the more untrustworthy ones (although it is not obvious that Case 4 is generally more untrustworthy than Case 3, as is presented in Figure 4.1).

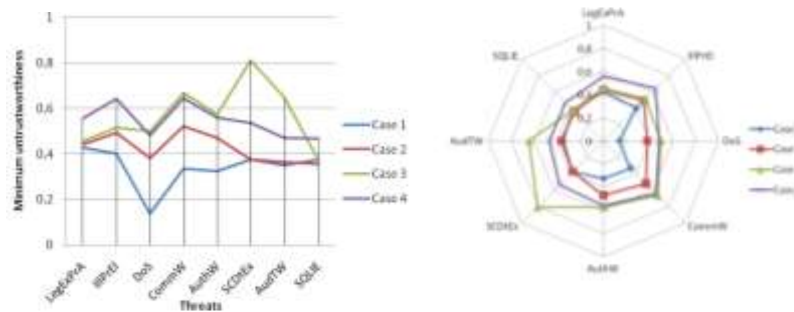


Figure 4.3 Alternative presentations for untrustworthiness comparison between cases

From an administrator perspective, comparing individual threats against each other provide the most useful piece of information of the benchmark, in the sense that it

allows focusing in the threats that are the most relevant for a particular environment and that have the higher untrustworthiness. One way to approach this analysis is to evaluate the list of threats ordered from the least untrustworthy to the untrustworthiest, which allows comparing threats two by two. This analysis is summarized, for each case, in Figure 4.4. Besides the untrustworthiness associated with each threat, the graphs also present visually the standard error associated with the results (Zwillinger, 1995).

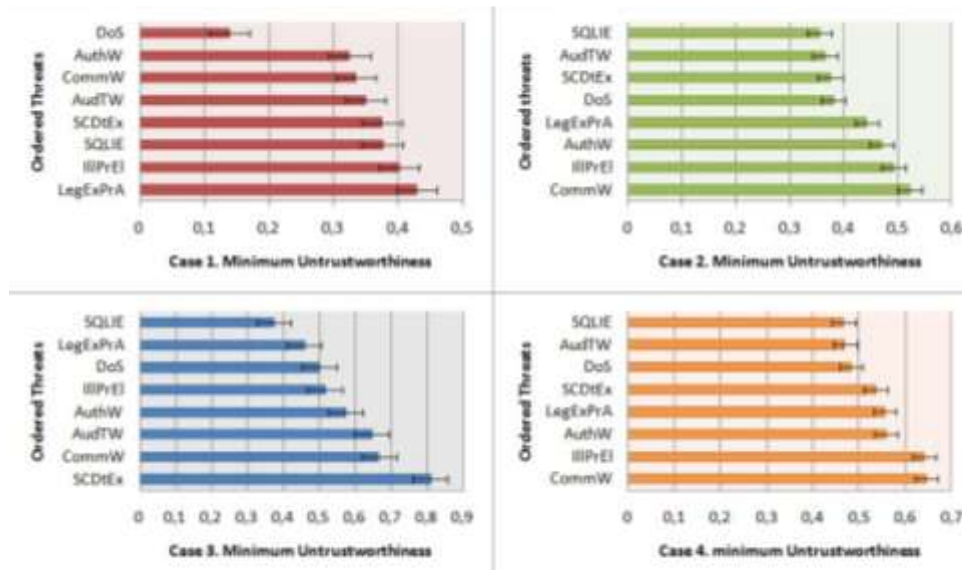


Figure 4.4 Fine grain analysis of untrustworthiness, for each case

The untrustworthiness values for Case 1 suggest that the most untrustworthy area of configuration are related to the *Legitimate excessive privilege achievement* (LegExPrA) threat. We can see, however, that given a margin of error, *Illegitimate Privilege Elevation (IllPrEl)* should also be a concern in this scenario. We can actually see a pattern dividing the threats in three or four distinct groups, with these two threats forming the most untrustworthy group, and *Denial of Service* (DoS) being in the least untrustworthy group. Obviously, these observations depend on the administrator's perceptions of what would be the most dangerous threats to his system. Along these lines, in Case 2 we can spot 3 different groups with *Communications Weaknesses* (CommW) having the highest untrustworthiness, while the four least untrustworthy (SQLIE, AudTW, SCDtEx and DoS) have more or less the same values. Case 3 presents at least five clearly distinct groups, with *Side Channel Data Exposure* (SCDtEx) being clearly a very poorly covered threat. Case 4, on the other hand, presents three groups of threats, with *Communication*

weaknesses (CommW) and *Illegitimate Privilege Elevation* (IllPrEl) on the top priority.

The benchmark can also be used to analyze the untrustworthiness from the perspective of the interaction classes. Figure 4.5 presents the minimum untrustworthiness computation for each interaction class, in each infrastructure. This analysis also shows some interesting trends. First, in Case 1, the DBMS users are the least untrustworthy and the operating system users are the most. If in this installation there are only a few operating system users, this may not be a big concern. However, if for example all developers also have an operating system account, this may be a wake-up call that some improvement should be done. In Cases 2 and 4, application users are the untrustworthiest. Case 4, in particular, is highly very untrustworthiness against applications users. In Case 2, on the other hand, due to the large number of developers, we might consider DBMS users a most relevant threat than application users.

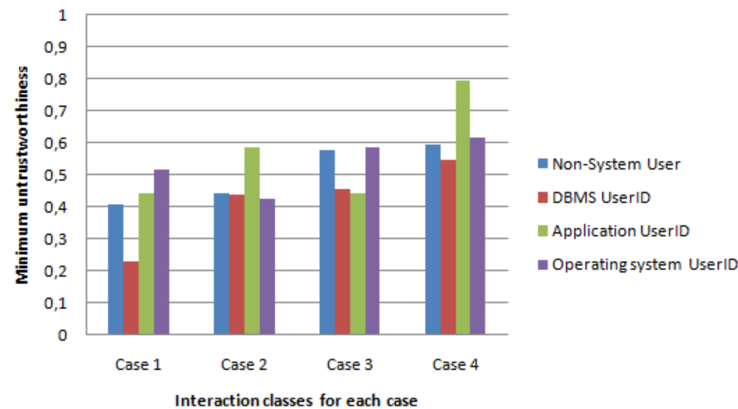


Figure 4.5 Untrustworthiness computation for the interaction classes

4.5 Conclusion

This chapter presented the instantiation of the framework proposed in Chapter 3 to the case of transactional systems infrastructures. In the first half of the chapter we developed a set of strategies and techniques aimed at designing the various components of the benchmark. In the second half we actually applied the benchmark to four real transactional systems infrastructures, identifying the characteristics of the installations and demonstrating the potential of our approach.

The most important conclusion of this chapter is that our approach is viable and can be applied in practice. Nevertheless, we cannot ignore the fact that the process

is long and demanding, even considering that the outcome is worth it. A key aspect to notice is that the design of the benchmark was, in a very basic level, a process that took as input an amount of consolidated security knowledge about a domain and converted it into a tool able to provide indications and metrics that can be readily interpreted by administrators and higher level business managers that are not security experts. No part of the ben benchmarking use adds security information to the benchmark, as all the security knowledge that is part of the benchmark, from the threat vectors, to the pessimistic scenarios and the security recommendations, is external information provided by reliable sources and experts. Another aspect is that the framework conducts the benchmark designer to correctly process and reason about the security information obtained externally, therefore leading to a tool that effectively represents and takes advantage of all the knowledge that is put into it.

Finally, an aspect that was not considered yet is the validation of the tool. Intuitively, a validation process for this kind of tool would be as follows: first, we would compute the trustworthiness metrics for a set of infrastructures. Then, for a certain time, we would analyze the existing security incidents within those infrastructures. The validation would consist of crosschecking the benchmark metrics with the types of problems observed. However, there is a fundamental problem with this approach: the security incidents observed would depend on the two main factors that determine the successfulness of a security breach: capabilities and intention (also indirectly related to *value*). Our benchmark, by design, provides metrics related only with one of these aspects, which is the *capabilities*. As discussed in Chapter 3, we should not include in the benchmark definition external factors such as intention being (see Section 3.2 for a thorough discussion about the effects and distortions that external factors may cause in the metrics). As both capabilities and intention are independent, the fact that a certain well protected area (as indicated by the benchmark) of the system suffers more security incidents than another less protected one, does not allow to conclude that the measures are wrong. The reality is that effectively validating our benchmark proposal is a complex problem that does not have an easy answer, and for that reason we leave it as future work.

5

Trustworthiness Benchmarking of Web Applications

This chapter explores the concept of trustworthiness benchmarking in the context of a controlled evaluation target, i.e. business applications, which are the part of the system that usually implements the business rules and that provide the interface to the end-users. The simplest definition we can give to *business applications* in the context of transactional systems is that they are the software designed to handle two main aspects (Yang 2011):

1. To provide the interface via which the end-users interact with the transactional system (e.g. by inputting information, retrieving information, and issuing commands);
2. To implement and enforce the rules of the business domain.

For a particular domain, a business application should evaluate what information requires authentication (or not) to be accessed, and provide the means to perform such authentication. Also, the application should define the available commands (and to whom they are available), which processes can be executed, and what data is required for each process (e.g. the mandatory fields in a data input form).

Even considering that most transactional systems follow a client-server model (Ram 1999), the exact place where the code is executed largely varies from one architecture to another. On one side of the business applications spectrum, we have thin client architectures, where most of the code executes within the server infrastructure (much like the old mainframe architectures) and the clients serve mostly for data input and information display. On the other side, we have

architectures where all the code of the application runs on the client platform, and the communication with the server infrastructure is basically to store and retrieve data by issuing database SQL calls (or any other equivalent data driven communication protocol). Obviously, a variety of intermediary approaches can also be used, including solutions where executable code is present on the client, on a server, and also in the database engine (e.g. in the form of *stored procedures* (Eisenberg 1996)).

One important variation of this distributed approach consists of using *application servers*, which are responsible for hosting the executable code that implements the business rules (in this context the clients have no direct connection to the backend database). When processing a request, the application server connects to the database and submits the required data access operations, in a way that is isolated from the clients. One advantage of this architecture is that it allows the database infrastructure to be shared by several business applications, remaining at the same time as an independent server, which allows it to be isolated from (potentially) untrusted computers and networks (the clients communicate only with the application server). This approach to transactional systems architecture design is frequently referred as *three-tier architecture* (Cardellini 2002).

Transactional systems based on *web applications* can be seen as a specific type of three-tier architecture that is becoming more and more popular (Hoffman 2008). A web-based application transactional system takes advantage of several standards (e.g. HTML for form design, CSS for interface styling, Javascript for interface functionalities (Hevery 2009), HTTP and HTTPS for network communications (Kaufman 2002)) that ultimately allow the developers to focus on programming the business rules, while most of the communication, network and infrastructural aspects are automatically handled by a diversity of available solutions. A web application is typically built based on the following set of standardized elements:

a) **Web server**: being the kernel of the application server, the web server is in charge of receiving client requests and sending the responses back. In practice, when a request is received, the web server redirects it to the local process responsible for processing it, and sends the output of that process back to the client. Several implementations, free and proprietary, are available out-of-the-box (e.g. Apache HTTP Server, Tomcat, Nginx).

b) **Web browsers**: applications that run in the client's computer and that communicate with the web servers using the HTTP and HTTPS protocols and primarily display content encoded in HTML. Almost all standard browsers support CSS (which is essentially a formatting standard) and

JavaScript (that allows including some local processing capabilities) languages (Hevery 2009). Standardization makes web applications inherently cross-platform, providing usability in a diversity of devices.

The development of web applications is highly tied with the infrastructure restrictions (the database engine and the server application), but it is almost independent from the client devices. Nevertheless, a huge variety of implementation languages, from compiled languages to interpreted scripting languages, can be used for the implementation of web applications (e.g. CGIs, java, PHP, .net, aspx, etc.) (Morrison 2002).

A key aspect is that the application server is a critical element in a three-tier architecture, and its security should also be considered in the context of a security benchmark. However, being a part of the transactional system infrastructure, we do not address it here. In practice, application servers should be benchmarked together with the transactional system infrastructure, following an extension of the methodology proposed in Chapter 4. Although we did not include the application server in the base scenario defined in Chapter 4, we addressed the specific problem of web servers' trustworthiness benchmarking in (Mendes 2008), a joint work that followed the framework proposed in Chapter 3.

Unlike a transactional system infrastructure (typically composed by a variable set of diverse devices, network infrastructure and software), what defines the runtime behavior of a web application is contained, in one way or another, in its source code (possibly along with some small set of configuration files), which makes available to a benchmark all the relevant information about the inherent security characteristics.

In the context of our framework (see Chapter 3), a security benchmark for web applications includes the processes and the analysis required for *security qualification* and *trustworthiness benchmarking*. The first should be defined by stating the set of tests needed to determine if the web application under evaluation fulfills the minimum set of security requirements needed to be acceptable in the application domain (a detailed discussion about those requirements is presented in Chapter 3). Such requirements are, by definition, primarily domain dependent, and therefore we refrain from providing any definitive list, as that is considered out of the scope of this chapter. Nevertheless, for illustration purposes, the following paragraphs briefly discuss the qualification step.

The qualification elements that can be expected in a wide range of web applications include: a variety of authentication methods, fine grain permissions settings, role

based privileges, general encryption capabilities (communication and storage), backup support, auditing mechanisms, logging, support from an active community or reliable organization, etc. All of these are actual security elements that are greatly described in typical security literature (Stallings 2010).

Another part of qualification would be the actual search for vulnerabilities, which can be defined as programming or configuration characteristics that allow the application to be attacked. These vulnerabilities can be identified by a variety of methods, ranging from automated static code analysis and penetration testing, to manual analysis by experts (McGraw 2006). As defined in our security benchmarking framework, the result of the qualification step is a set of systems that are acceptable for use, and are thus considered *reasonably secure* (i.e. the result of this step should not be used to compare the qualified systems). The benchmark user will use the tests and evaluations of the qualification specification in order to sort out the candidates that will therefore have their trustworthiness evaluated.

This chapter discusses and proposes approaches to obtain relative trustworthiness metrics for web applications. Section 5.1 presents an analysis of web applications from a security perspective Section 5.2 proposes a very simple method that allows computing a trustworthiness metric by using only a set of reliable static code analysis tools, and this approach is evaluated using several experiments. Knowing the limitations of the approach proposed in Section 5.2, Section 5.3 develops a theoretical approach to trustworthiness benchmarking of web applications, which includes the definition of what would be an "ideal" trustworthiness benchmarking metric. Finally, Section 5.4 concludes the chapter.

5.1 Web Applications from a Security Perspective

Web applications have several characteristics that make them particularly prone to security attacks, being their widespread exposure the most important one (Fonseca 2008a). This exposure obviously increases the probability of being attacked, including the risk of being used to leverage attacks against other applications, which forces us to assume the possibility of composite attacks (which makes the problem even more complex) (Balzarotti 2007).

Another key characteristic of web applications is that the base protocol over which they are built (HTTP) is essentially stateless, meaning that two distinct interactions between the web server and a client are more or less independent (*session tokens* are actually a work-around for this characteristic) (Chen 2009). After a first communication between a user's web browser and an application hosted by a web server (potentially including an authentication step), a session token (which is a simple global unique identifier) is generated and sent to the end user's web browser.

This token allows the server to keep the track of the actions performed by the user that owns it (ownership in this case if defined in terms of knowledge, and an attacker would successfully "steal" a session from a legitimate user if he manages to discover the value of the session token). From this point on, communication consists of stateless requests that usually include the following steps (Balzarotti 2007):

- **Step 1.** The user sends to the server a set of parameters (e.g. key-value pairs, which might include the session token) and indicates a target resource (e.g. a web page).
- **Step 2.** The server processes the code of the target resource using the parameters provided by the user. This processing can be extremely complex, including, for example, file system calls, database calls, and the execution of other services and processes. If a session token is provided, then the values stored in the server (and that are associated with that token) may also be used as input for this execution (e.g. as the case of session variables).
- **Step 3.** After finishing processing, the server replies with an output. This output is usually a stream of data that can have several formats depending on the application context (e.g. html text, file contents, forms).

Based on this simplified processing model, a typical web application attack consists of crafting one or more of the input parameters in a way that at least one of the following effects is achieved:

1. The output on step 3 presents either out-of-format data (e.g. an executable script code instead of text information) or confidential data (e.g. confidential database records/fields, private/critical files content, internal server state information).
2. Step 2 causes the state of the application to be modified in an unintended way (e.g. database or files modification, unexpected services call, resources usage).

In other words, a threat can be defined as a particular set of parameter crafting techniques that aims at causing one or more of the previously mentioned effects (Jovanovic 2006). For instance, SQL injection (Amirtahmasebi 2009) consists of manipulating input parameters in order to cause a semantic change in a specific SQL command that is sent to a database. A cross-site-scripting (XSS) attack (CGI Security 2010), on the other hand, includes a set of crafting techniques that cause a state change, forcing the server to output out-of-format data in a set of subsequent requests (e.g. a executable script code that is sent to another user or reflected back to the same user).

An important characteristic is that an attack that implements such a threat usually aims at a specific line of code (or a few strongly coupled lines of code with a single semantic goal). For example, an SQL Injection attack typically aims at a single database SQL call, and a XSS attack targets the statements in charge of returning the output to a user (e.g. a "printf" or "echo" statement). This way, for each threat type it is possible to identify a set of code statements that can be the target of such a threat. We call these statements **hotspots**. In practice, even though several hotspots may exist for a specific threat, a particular attack is usually aimed at one specific hotspot (Integrigy 2007). Therefore, the goal of the attacker is to manipulate input variables that influence a particular hotspot in order to cause a malicious effect.

From the developer's perspective, each hotspot is designed with a particular "business activity" in mind, and helps implementing a given functionality (or set of functionalities). Usually, a developer defines a set of input values that are processed (directly or indirectly) by a particular hotspot, and design that hotspot to generate the corresponding output values or actions. The set of input values represents the *input business domain* of the hotspot. Attacks are accomplished by using values outside that domain and for which the hotspot may not be correctly designed.

Input business domains are relative to each hotspot. This is important, and means that these domains may vary from one hotspot to another, and may also differ from the input domain of the whole web application (i.e. the domain of the parameters actually provided by the end user). This is a frequently overlooked characteristic that makes the task of securing the entire web application considerably more difficult.

The web applications characteristics previously presented suggest two distinct lines of defense against threats. The first consists of reducing the input domain of the application as a whole, acting directly on the values provided by the end users. The idea is to force the input parameters to be within the valid business domains (for the whole web application) or to interrupt the execution when a value outside the domain is provided (this is frequently called *input validation* and can be achieved by a set of filtering (Liu 2006)). This line of defense, however, is frequently not enough, as the input business domain of a hotspot may not coincide with the domain of the application. The problem is that the business domain of the application corresponds to the composition of the input business domains of all hotspots, which makes this reduction extremely complex (or even impossible in some cases). Consider, for instance, the classical problem of a string value that contains a single quote, which is the character used as a string delimiter in most SQL statements (Integrigy 2007). It may not be possible to escape this single quote universally

because the string value may be used in other places besides SQL statements (e.g. it may be outputted to the user). In this case the developer must either create an escaped copy of the value (which may not be practical if the value is further processed, creating potential inconsistencies) or delegate the responsibility of dealing with this issue to each hotspot. Thus, the actual relation between the input parameters and each hotspot may be hidden under the application's complexity.

The second line of defense, necessary to complement the limitations of a general input validation strategy, is to guarantee that the values actually used in each hotspot lie within the input business domain of that hotspot. Several aspects must be considered in this case, namely: *technical characteristics* (e.g. the SQL language details for a SQL execution, the file system structure for a file access); *context characteristics* (e.g. the output generated by the hotspot, how the data should be interpreted and in what context); and the *application's business rules*. These aspects strongly define the characteristics that the values used in the hotspot must respect in order for the hotspot to always behave in the expected way. In practice, this set of characteristics defines what we call the *Business Data Type* of each hotspot. Strong Business Data Typing (in the same sense of traditional strong data typing (Tomatis 2004)) is different from a typical data typing because it takes into consideration all the aspects related to the use of the value, and not just a programming language and codification perspective of data typing (e.g. a numeric variable may not contain a string). A key difficulty is that the Business Data Type of a hotspot may not be easy to identify, as it is the result of a mixture of business rules, and context and technical characteristics. Guaranteeing its correctness is, however, the most important part of the defense, as this is where attacks will take place in a web application (Monga 2009).

In summary, coding best practices for secure web applications can be divided into two big groups:

- **General Input validation**: each input parameter of a web application should be validated against a valid business domain. Values outside the specified domain should either be replaced by values within the domain, or the application must halt indicating an input problem.
- **Business data typing for hotspots***:* any value used within a hotspot must conform to a set of technical, context and business constraints.

In a defense-in-depth approach, the developer is expected to always consider these two types of best practices, even when one of the types seems to be enough to protect against a specific threat.

5.2 Benchmarking the Trustworthiness of Web Applications using Static Code Analysis

Static code analysis is a well-known white-box technique based on the assessment of the source code (or the *bytecode* in more advanced analyzers) of an application, frequently used by developers to discover bugs and security vulnerabilities in web applications and components (Chess 2007). The goal of this technique is to identify specific code patterns that represent security vulnerabilities. Most analyzers are based on expert knowledge (Livshits 2005) that is built directly in the tool and several tools implementing such technique are currently available (including free and commercial tools) (FindBugs 2011; Yasca 2011; IntelliJ IDEA 2011).

From a high-level perspective, a Static Code Analyzer (SCA) commits to a certain set of patterns that define the types of bugs that it can identify. These patterns are necessarily limited within the available expert knowledge, which means that even excellent analyzers may miss particular types of bugs (Chess 2007). In practice, pattern sets for static code analysis can be classified as *loose* or *tight*. A tight pattern matches precisely a wide range of code bugs, but allows bugs represented by other unpredicted patterns to slip through. On the other hand, a loose pattern is better for finding bugs in unpredicted formats, but more easily points portions of code that (even though they appear to be) are not bugs, which are known as *false positives* (Chess 2007).

False positives are usually considered bad as they cost time to analyze without bringing useful information to the evaluator (i.e. they point nothing to correct) (Nadeem 2012). However, one possibility is that they may carry other kind of information. In fact, false positives are usually code patterns that "look as" bugs but are not. In other words, they are code patterns that somehow are "close to bugs and usually a single detail (either in that portion of the code or in another related part) separates them from becoming actual vulnerabilities". In other words, one **hypothesis** is that this kind of code (i.e. false positives) may also be dangerous, meaning that a source code filled with code patterns leading to too many false positives may be more untrustworthy and prone to vulnerabilities than one with less.

In this section we present a series of experiments that investigate the possibility of combining the output of different SCAs to define metrics for trustworthiness benchmarking of web applications. In practice, we try to answer the following question:

Is the combination of state-of-the-art static code analysis tools a potential approach towards obtaining metrics for comparing the trustworthiness of web applications and components?

In this study we define four metrics and investigate their behavior, trying to verify their relation with security attributes. In other words, we attempt to provide evidence that there is a correlation between these metrics and the security properties of the benchmarked code. The metrics are based on the raw number of vulnerability warnings reported by a set of static analysis tools. In this scenario, instead of formally defining the threat vectors (as in Chapter 4 for transactional systems infrastructures), we simply assume that the threat vectors are defined by the insecurity characteristics that the tools are designed to detect (in the end, the goal is to assess if such threat vectors are representative and correlate with real security issues or not).

To understand the effectiveness of the proposed metrics, we conducted a set of controlled experiments. In these experiments, the benchmarking approach was applied to the detection of SQL Injection vulnerabilities (which are among the most frequent and dangerous vulnerabilities in the web environment (OWASP 2010)) in different implementations of the TPC-C, TPC-W, and TPC-App standard applications (TPC 2012). Results show that the raw number of vulnerabilities detected by static code analyzers allows establishing a rough tanking of applications, but the unstable nature of false positives is a problem when performing fine grain comparison. To account for this, we then calibrate the metric based on false positive rate estimations, which indeed allow improving precision. To demonstrate the effectiveness of the calibrated metrics, we present the results of the approach applied to a set of real web applications, namely seven distinct web forums developed in Java. Results are validated based on an expert analysis, further showing the usefulness of the proposed approach.

5.2.1 Trustworthiness Metrics

The aggregated total number of security warnings reported by a set of static code analyzers is the key for building the proposed trustworthiness metrics, but its raw value cannot be used directly for comparison, and we have to make clear why this is true before proceeding. In practice, to design a proper metric suitable for comparison, two problems have to be accounted for, as discussed next.

The first problem is related with the applications being compared, which may have different *sizes*. The problem is easy to understand through an example. Suppose that we are comparing two applications that use exactly the same coding style, but one is twice the size of the other. If they are similar, they have the same type of coding patterns, and thus trigger false positives approximately with the same rate. In this case, the application with bigger size will be considered *untrustworthier*, which may not be true.

In fact, suppose we benchmark two applications, A and B. Application A presents 2 warnings and has size X. Application B also presents 2 warnings, but its size is 10X. The idea is to use the raw metric, i.e. the number of warnings, as an estimator of the number of lines of code that can be considered bad programming practices. We also hypothesize that more examples of bad programming practices will tend to lead to a proportionally higher number of real hidden vulnerabilities. Thus, as both applications have the same number of warnings, the number of hidden vulnerabilities they are assumed to have should be similar. The issue is that finding one vulnerability among tens of thousands of source lines of code (SLOC) is harder, on average, than finding the same vulnerability among a few thousands of SLOC (from an attacker perspective), which means that, for two applications with the same number of security warnings, the one with smaller size is more likely to have one of its vulnerabilities exposed. This rationale is the extrapolation of the concept of "defect density" (Sherriff 2006), which is used as a metric of software quality, where it is assumed that software with a higher defect density most frequently manifests its defects, as code with defects is executed with a higher frequency (therefore, the software with the higher defects density is the one that is classified worse, and not the one with the higher absolute number of defects). This way, to allow fair comparison, the number of security warnings has to be normalized by the size of each application, so that the size does not distort the results. In other words, instead of the absolute number of security warnings, what we need is the *security warnings density of the application*.

The second problem that has to be taken into account when using the results of several static code analyzers to build a trustworthiness metric is related to the effectiveness of such tools, and has to do with the *frequency* with which each one yields false positives. As static code analyzers are mostly based on search patterns, the number of times that these search patterns are triggered is directly related to the intrinsic characteristics of each implementation, and varies drastically from one analyzer to another. While we want these different patterns to count, we do not want one analyzer to be awarded more importance in the results than the others. In other words, if one analyzer tends to trigger proportionally much more warnings than another one, this would lead the results of this analyzer to have more important in the calculation of the final metric. This way, it is necessary to guarantee that all analyzers contribute in the same way for the final result.

The exact number of false positives depends not only on the analyzers, but also on the combination of their search patterns and the code being analyzed (Littlewood 2010). However, as we are designing a procedure that should serve to compare different applications, we may not have access to the source code beforehand,

therefore the best that we can do is to compute an average estimation of the false positives rate for each analyzer. Assuming that a tool implements either a tight search pattern (that tries to hit a precise set of known vulnerability types) or a loose search pattern (having a more broad, but also more unreliable, search pattern), then it will tend to report, respectively, less or more false positives in a consistent manner. Obtaining these factors - an average of the false positive rate for each tool - allows us to calibrate the number of vulnerabilities reported in a way that all tools end up having approximately the same contribution to the final metric. As computing these estimates is a difficult problem and should rely on an extensive and targeted evaluation, in our experiments we adopted the estimates provided in (Antunes and Vieira 2010), where the authors computed such factors in the context of web services for the same tools that we use in our experiments (see Section 5.2.2). However, for other tools, these values have to be estimated, possibly using a methodology similar to the one proposed in (Antunes and Vieira 2010).

Considering the previous discussion, the metrics we propose and analyze are:

- **Raw Number of Vulnerabilities Reported (*Raw-NVR*)**. Represents the sum of the number of vulnerabilities reported by each of the SCAs considered. Obviously, we are expecting that different tools detect different vulnerabilities (as is demonstrated in (Littlewood 2010)) and that the union of the search patterns of all tools achieves higher coverage than any single tool. As explained before, this metric is expected to be biased by the tools characteristics, and we do not expect this to be the best metric, even though it should also correlate with security aspects. However, it is very easy to obtain.
- **Calibrated Number of Vulnerabilities Reported (*Cal-NVR*)**. To reduce the impact of different false positives rates we evaluate the application of a calibration factor, as previously explained. This metric is computed by applying a constant factor to the Raw-NVR metric using the estimates provided in (Antunes 2010).
- **Normalized Raw Number of Vulnerabilities Reported (*Norm-Raw-NVR*)**. To take into account the size of the application, we also compute normalized metrics. In our experiments we define Norm-Raw-NVR as Raw-NVR per 100 lines of code. This could be done using any other normalization factor relative to size, like the number of classes or the number of features; what is important is to allow expressing the warning density of the application (Gencel 2008).
- **Normalized Calibrated Number of Vulnerabilities Reported (*Norm-Cal-NVR*)**. This is the normalized version of the Cal-NVR metric,

considering again 10k SLOC as the normalization factor.

The next sections present a detailed analysis of the semantics of these metrics from a benchmarking point-of-view, trying to reason about what is exactly the meaning of the numbers being reported.

5.2.1.1 SCAs Reports as a Trustworthiness Metric

An important assumption of this work is that the aggregated reports of a set of static source code analyzers can be considered a fair measure of trustworthiness (i.e. they provide enough evidence of security practices to allow comparison from a security point-of-view). This assumption has a crucial consequence: as *true vulnerabilities* are not distinguished from *false positives*, we are effectively giving them the same importance. This is extremely important and deserves some justification.

It is clear that any real vulnerability in a web application is an immediate security hazard. If a static code analyzer can find it, then it is likely that some attacker will also be able to find it, thus it would be extremely dangerous to use the application as is. However, within our framework, the task of distinguishing acceptable applications from the unacceptable ones is performed during the security qualification step, and not via trustworthiness benchmarking. Because of this, we assume that if true vulnerabilities (that can be found by static code analyzers) are present during trustworthiness benchmarking, then these vulnerabilities are, from an objective perspective, as harmless as false positives. In other words, if those vulnerabilities are not harmless, then the application under benchmarking would not qualify in the first place. Therefore, our trustworthiness benchmarking approach starts from the principle that any security problem in the source code is related to hidden and hard to detect vulnerabilities that can only be estimated and not actually found. In this sense, the original hypothesis translates into the idea that the aggregated results of false positives of several SCAs may help on estimating the quality of web application code (from a security point-of-view), which is directly affected by the number of hidden vulnerabilities.

Another assumption we make is that the characterization of the trustworthiness of an application must go beyond what is allowed by a simple vulnerability identification process. Typical web applications are constantly being upgraded, fixed and improved, and these maintenance tasks are often a source of new vulnerabilities (Shahzad 2012). Also, it is well known that new features are usually developed more or less in the same coding style of the rest of the application. The reality is that the probability of new bugs to be added during a source code maintenance task has a direct relation with the probability of having vulnerabilities introduced due to the coding style being used (Shahzad 2012). This is fairly simple

to understand if we consider that most vulnerabilities are simple forgotten details (e.g. one parameter among several that is not validated properly). On the other hand, if the coding style makes it inherently difficult to disregard such details, then the code should be considered *trustworthier*.

Considering these aspects (and assuming a *benchmarking perspective* where the goal is to fairly compare applications), metrics based on the number of reports seem to be quite reasonable, as long as they do relate to secure or insecure coding styles. The reasoning is that if they do correlate with security aspects (and the most they correlate, the better) then the proposed metrics are useful.

5.2.1.2 Combining the Output of Several Tools

The Number of Vulnerabilities Reported (NVR), which is the simple count of the security warnings reported by a tool, can be expressed by three factors: the number of *True Vulnerabilities* in the code analyzed, the number of *Missed Vulnerabilities* (MV), and the number of *False Positives* (FP). In short, NVR can be defined by the following equation:

$$\mathbf{NVR = TV - MV + FP}$$

As mentioned before, different analyzers end up presenting different results because they scan for different vulnerability pattern sets. One way to find more vulnerabilities and insecure coding patterns is to have a looser pattern set (potentially increasing the number of false positives). Another way is to use several different tools that implement different and complementary patterns. The combination of several SCAs is an easy way to amplify the search pattern, without raising the false positives rate significantly. In this case, the aggregated result can be expressed as:

$$\mathbf{Raw\text{-}NVR = TV - MA + FP_1 + FP_2 + \ldots + FP_n}$$

where FP_1 to FP_n represent the false positives reported by each tool and MA is the number of vulnerabilities missed by ALL scanners. MA will be significantly smaller than any individual MV if the search patterns complement each other.

As we assume that obvious vulnerabilities (detected by SCAs) were previously (i.e. before the trustworthiness benchmarking step) fixed by developers or are as harmless as false positives, we can consider that only the vulnerabilities missed by ALL analyzers remain in the code, and no true vulnerabilities are reported. So, Raw-NVR can actually be defined as:

$$\mathbf{Raw\text{-}NVR = FP_1 + FP_2 + \ldots + FP_n}$$

We expect this metric to give an insight on the trustworthiness of the benchmarked code, based on the number of false positives. In other words, the biggest the Raw-

NVR, the more untrustworthy is the code and the higher is the number of vulnerabilities hidden. If our metric (i.e. the number of false positives) correlates to the security of the application, then in some sense false positives must be proportional to the number missed/hidden vulnerabilities. In practice, if this proportion is equal for all SCAs in all benchmarked applications, then the Raw-NVR should be the best metric in our set. However, as false positives depend much on the patterns of each tool and on the code being benchmarked, it is possible that Raw-NVR unrealistically award more importance to the results of the SCA that tends to report more false positives, which would not be in the best interest of the benchmark. To better understand this case, we should consider calibrated metrics.

5.2.1.3 Calibrated Number of Reported Vulnerabilities

In order to reduce the influence of false positives rate of specific SCAs, we propose to calibrate the results from the individual tools by applying a factor to the number of reported vulnerabilities. Assuming that that rate depends on the pattern of the SCA and is proportional (on average) to the number of missed vulnerabilities (MV), we conclude that NVR is determined by the following equation, where FPF represents the False Positives Factor for a specific tool:

$$\mathbf{NVR = MV * FPF}$$

By dividing the number of vulnerabilities reported by the False Positives Factor, we obtain the number of missed vulnerabilities. Thus, if we aggregate several calibrated SCAs, we get the following calibrated NVR metric:

$$\mathbf{Cal\text{-}NVR = NVR_1 / FPF_1 + \ldots + NVR_n / FPF_n}$$

$$\mathbf{Cal\text{-}NVR = n * (MV_1 + \ldots + MV_n)}$$

Assuming that the vulnerabilities missed are the same for all analyzers (i.e. the detected ones were corrected before starting the trustworthiness benchmarking step) then Cal-NVR is proportional to the number of hidden vulnerabilities. A key aspect is that the False Positives Rate required for each tool corresponds to an estimation of the average rate of false positives reported by that tool in a wide range of possible source codes. The problem then becomes gathering realistic estimates for FPF, which is not a simple task. In our work, we use the estimates presented in (Antunes 2010). This work provides an evaluation of the average false positive rates for several SCAs in the context of Web Services, which are usually based on similar constructions and programming languages as Web Applications in general (Almonaies 2011).

5.2.1.4 Normalized Metrics

The proposed normalized metrics are quite easy to compute. Basically, the idea is to apply to the previous two metrics a factor that represents the size of the application being benchmarked. The metrics present then the following form:

Norm-Raw-NVR = Raw-NVR / Size_Factor

Norm-Cal-NVR = Cal-NVR / Size_Factor

Any factor that represents what the benchmark user understands by "application size" can be equally fair. For instance, the number of classes or the number of features can be both used (Gencel 2008). However, as vulnerabilities tend to manifest in specific lines of code (see discussion in Section 5.1), source lines of code (LoC) appear to be the most interesting and adequate size metric. In our experimental evaluation, we consider 100 LoC as the size factor for convenience and readability, as it has absolutely no effect in the relative values (i.e. they do not affect the comparison of tools).

5.2.2 Empirical Analysis of the Metrics

To understand the effectiveness and validity of the proposed metrics, we conducted a series of experiments under controlled conditions. For these experiments, we designed three distinct versions, each one with distinct security qualities, of four of the web services specified by the TPC-App standard (tpc 2011), which is widely accepted as being representative of web services. Using these implementations, we analyzed the behavior the NVR-Raw metric by comparing it to the number of true vulnerabilities in each version. This analysis was done for all the applications and also at a component level. In a subsequent experiment, we created sixteen versions of three completely distinct web services, one from the TPC-App, one from the TPC-C (TPC 2005) and another from the TPC-W (TPC 2002) standards. These sixteen versions where created by injecting real vulnerabilities in each of the versions, creating a progressively worse set of applications. We then computed and analyzed the NVR-Raw metric and the calibrated metrics of each of these versions.

5.2.2.1 Static Code Analyzers and Web Applications Studied

The experimental setup is based on three well-known SCAs: FindBugs (FindBugs 2011), Yasca (Yasca 2011), and IntelliJ Idea Analyzer (IntelliJ IDEA 2011). These tools are widely used by practitioners and were also applied in several previous research works (e.g. (Ayewah 2007, Antunes 2009, Antunes 2010)). The experiments focus only on SQL Injection, as this vulnerability is one of the most frequent and dangerous in web applications (OWASP 2010), and also because (according to the vendors' web sites) the three tools are able to detect them. Note,

however, that any other type of vulnerabilities for which good tools exist could have been considered.

To implement the services, we started by inviting a 3rd year undergrad student. During a subsequent security inspection conducted by us, 9 SQL Injection vulnerabilities were identified in this first version (referred to as implementation V1). Afterwards, we took this implementation and, by performing the minimum changes possible, corrected the 9 vulnerabilities, creating an implementation similar to V1, but with no SQL Injection vulnerabilities (called V2). Finally, we invited an experienced programmer (with more than 3 years of programming experience and extensive knowledge of security of web applications) to develop a secure version of the same application (named V3), which presented zero vulnerabilities during code inspection. In summary, the experiment included implementation V1, with 9 vulnerabilities, implementation V2, with 0 vulnerabilities, but having a coding style very similar to V1, and implementation V3, with 0 vulnerabilities and having a coding style completely different from V1 and V2. All applications have approximately the same size (a few hundreds of lines of code), and therefore normalization of the metrics is not necessary. We study metrics normalization when comparing real applications in Section 5.2.3.

5.2.2.2 General and Component Level Analysis of Raw-NVR

We started the experiments by computing the Raw-NVR metric for the three versions. Figure 5.1 presents the results, including the true vulnerabilities (as detected in our manual analysis).

As shown, the metric clearly highlights some differences in the security of the applications. The actual Raw-NVR value is very different from the true number of vulnerabilities, but the relative values resemble very accurately the security of each version. In fact, both V2 and V3, which have no vulnerabilities, scored the same value (10), while the implementation with 9 vulnerabilities scored more than the double of the others. Even though we expected similar values for V2 and V3, it was a surprise that they both scored so equally. To better understand this, Figure 5.2 Component level evaluation of Raw-NVRbreaks down the metric for the four services in each version. As we can see, even though V2 and V3 scored equally in total, the distribution of the false positives is quite different in both implementations. In V2 they are centered in the NewCustomer service, while in V3 they are more evenly spread. The higher than average score found in the service in V3 calls the attention as this means that this service was built using a programming pattern different from the rest. At the same time, we notice that the programming style used by the experienced programmer was more consistent, and no module stands out from the others. Nevertheless, we cannot forget that this is the Raw

metric, and these results are biased by the false positives rates of the tools.

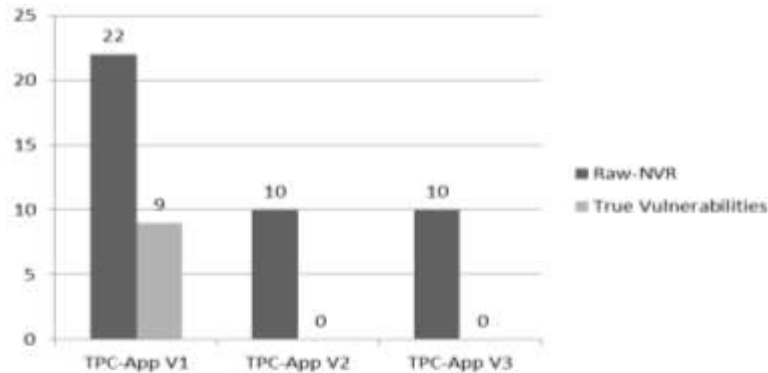


Figure 5.1 Benchmark results of our controlled TPC-App versions

To more extensively evaluate the problems of the NVR-Raw metric we did another experiment using implementations of three different TPC services implemented by three distinct developers: NewCustomer service from TPC-App, CreateNewCustomer service from TPC-W, and Delivery service from TPC-C, having zero known vulnerabilities each (these specific classes were chosen by the simple fact that at the time of the experiments they were already implemented for other research works, but were exactly what we needed for our experiment, therefore we would not have to wait again for new implementations. It is important to understand that other classes could also have been chosen).

Based on these three initial implementations, we created 15 more versions for each service by injecting randomly chosen SQL Injection vulnerabilities in the code (the vulnerabilities injected are from real samples drawn from vulnerable versions of the same applications). The idea was to create different versions of the same applications that were progressively worse in terms of security, which would allow analyzing the metrics behavior by comparing the values computed for each version. The 15 versions of each service were generated as follows: first we created four versions with one different vulnerability each; then, we took these four versions and by mixing each of the four vulnerabilities we created the remaining combinations (6 versions with all combinations of 2 vulnerabilities, 4 versions with 3 vulnerabilities, and one version with the 4 vulnerabilities).

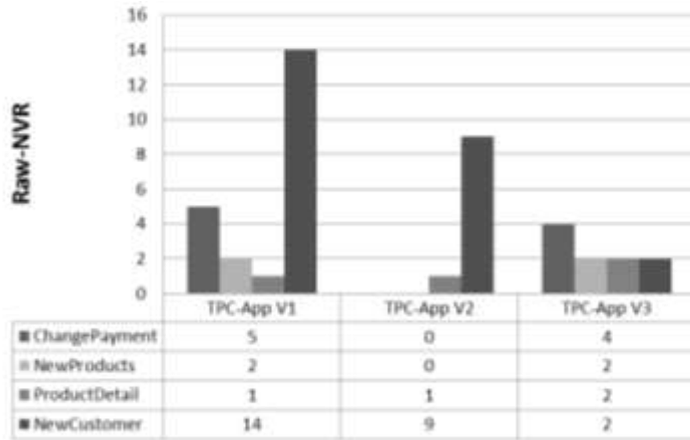


Figure 5.2 Component level evaluation of Raw-NVR

Figure 5.3 shows the Raw-NVR metric for the 16 versions of each service, ordered by version (the version with no vulnerabilities is number 1, the ones with 1 vulnerability are numbered 2 to 5, and so on). The dotted line in the graph (the bottom one) is a baseline representing the true number of vulnerabilities in each corresponding service version (i.e. the vulnerabilities injected).

The data presented in Figure 5.3 clearly shows the imprecise nature of the Raw-NVR metric when used to compare components that have a very similar (or equal) number of true vulnerabilities. We can see that in some cases the metric is more influenced by the false positives than in others, yielding a varied number of erroneous characterizations. For instance, the profile of the *Delivery* service metric is very similar to the base line (which portrays the true number of vulnerabilities of each version), allowing a fair relative comparison. In fact, the metric on this service shows an error only in 3 cases: when we compare versions 4 and 5 with versions 6 to 8, and when we compare versions 11 and 12. On the other hand, for the service *CreateNewCustomer*, the metric leads to several erroneous comparisons, stating, for instance, that version 5 is worse than versions 6 to 13, which is not true because we know that version 5 has less vulnerabilities than the others.

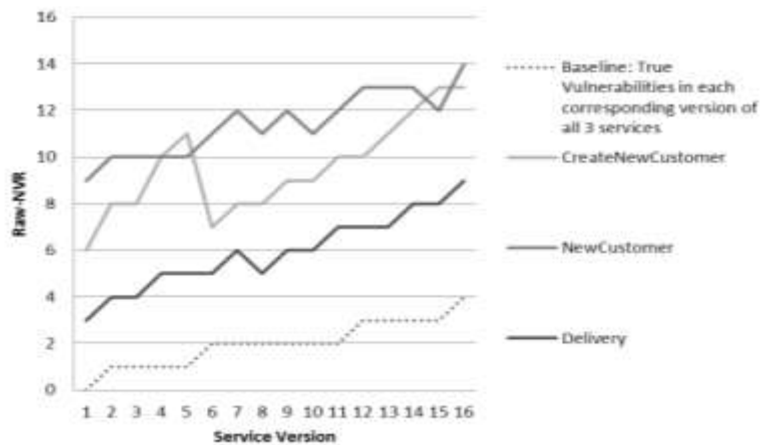


Figure 5.3 Raw-NVR evolution in 16 versions of 3 different services, ranging from 0 to 4 vulnerabilities

An important aspect that can be observed in Figure 5.3 is that the experiment confirms our first hypothesis: if there is a significant difference in the number of vulnerabilities, the metric actually portrays it. In fact, in all implementations, the versions with 0 or 1 vulnerabilities are better scored than versions with 4 vulnerabilities, despite how erratic the false positives rate. This suggests that using the results of several representative SCAs may be a representative way to compare the trustworthiness of web applications that have a very distinct security quality, but may not be a so good approach to distinguish applications that are too similar (in security terms).

5.2.2.3 Analysis of Cal-NVR

As mentioned before, to calculate the Cal-NVR metric we adopted the calibration factors proposed in (Antunes 2010). The False Positive Factors used are 7% for Findbugs, 36% for Yasca, and 67% for IntelliJ Idea. To understand the accuracy of this metric we computed it for the 16 versions of the *NewCustomer*, *CreateNewCustomer*, and *Delivery* web services mentioned above. The results are presented in Figure 5.4.

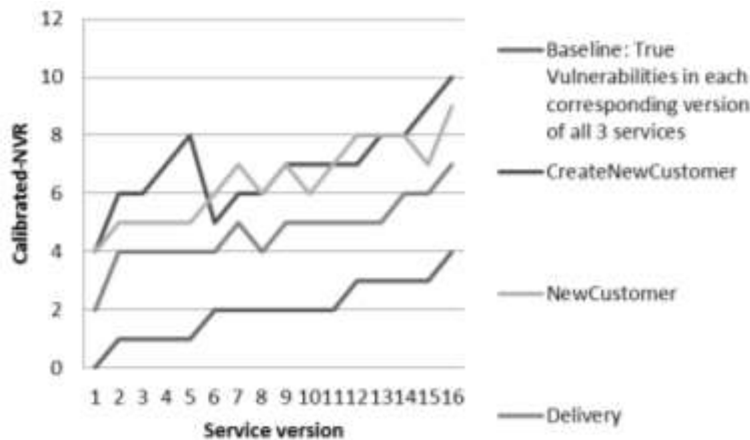


Figure 5.4 Calibrated metric analysis for the 16 versions of each service

Figure 5.4 shows that the detailed pattern of the curves did not change much (when comparing to the Raw-NVR metric shown in Figure 5.3). This was more or less expected, as the calibration factor is constant. The other thing that can be observed is a really important improvement. While for the Raw-NVR metric the three curves almost never intersect (as shown in Figure 5.3), the same does not happen for Cal-NVR. This suggests that, even though the comparison between versions of the same service is roughly accurate when using the Raw-NVR metric, comparisons between different services are completely off. The calibrated metric, on the other hand, is better than the raw metric when comparing diverse software. This claim, however, requires more evidence, as presented in a more broad evaluation in the next section.

5.2.3 Experimental Evaluation

In this section we present an experiment conducted to understand the validity of the metrics in a scenario more close to a real use case of trustworthiness benchmarking. To accomplish this, we used the proposed benchmark to rank seven distinct web forums implemented using Java, and having a variety of sizes and features. In order to have a baseline for comparison, we invited six experts to rank these same seven web forums. Of these six experts, four are PhD students working in the area of web applications security, all of them with at least two years of experience in the field. The other two are software engineers with more than five years of experience in the development of web applications with security requirements.

The problem we proposed to these volunteers was quite simple and representative, and can be summarized as follows:

> *"Suppose your company wants to install a web forum for its employees to communicate internally, but the forum will also be accessible through the web. Concerning features, usability and performance, it was determined that any of these seven web forums can be used. Your job is to provide a ranking among these seven web forums concerning security: the ones most secure (in the broadest sense of the word) come first. No ties are allowed"*

To conduct this task, we asked the volunteers to consider in the ranking process all the aspects they believe to be important from a security point-of-view and also to report the overall process and judgments that lead them to their decisions. This allowed us to have a rough idea of the most important aspects considered by the experts when analyzing the web applications, which we took into consideration in our final analysis (see Section 5.2.2.3).

The web forums benchmarked are the following: Yazd 3, JavaBB v0.99, JForum v2.1.9 and v3beta, JGossip v1.1.0, mvnForum 1.2.2 and JSForum 0.0.1 beta (Forums Benchmarked, 2011), all available for free download. Most of these are extremely popular (e.g. Yazd and mvnForum), others not so much (e.g. JSForum). To make the experiment the most representative possible, we used a set of representative criteria to select the forums, namely: they have the most common features expected in a web forum, they are developed in Java, and the source code is publically available. The last two criteria were necessary as the static analyzers used target only Java code and require the source code of the application to be available (even though FindBugs only requires access to the *bytecode*). Clearly, these constraints may be changed if another set of analyzers is chosen. At the same time, it is expected that results provided by different sets of analyzers should not be compared (in absolute terms).

In this experiment we decided to evaluate only the Raw-NVR and Norm-Cal-NVR metrics, omitting the intermediary Cal-NVR and Norm-Raw-NVR. We chose to not analyze these metrics for two reasons: first, we already established, in the controlled experiments, that the calibrated metric is better suited for comparing diverse software, which is what we are doing in this experiment; second, we need to apply normalization because the forums being compared have very different sizes, and as discussed previously, we need to focus on the problem of *density*.

We also invert the Norm-Cal-NVR metric in order for it to grow with the trustworthiness of the application. This is only a cosmetic decision, and the behavior of the metric does not change. However, in order to be consistent, we call this inverted metric as Trustworthiness Metric (TM). TM is computed as the inverse

of Norm-Cal-NVR, so that it grows with less vulnerability warnings. The exact formula for TM is as follows:

$$TM = \frac{\text{No. Lines of Code}/100}{F*0.93 + Y*0.64 + I*0.33}$$

where F is the number of security warnings reported by Findbugs, Y is the number of warnings reported by Yasca, and I is the number of warnings reported by IntelliJ Idea, while the constants are the false positive factors of each tool, as explained before. The trustworthiness value is normalized in terms of the size of the target application considering blocks of a hundred lines of source code.

5.2.3.1 Analysis of the Overall Results

Table 5.1 presents the overall results of the benchmark, where the first column presents the rank of each application. We also include the number of Lines of Code and the average Cyclomatic Complexity of each application (Lyu, 1996). Cyclomatic complexity is a metric that tries to express how complex a certain code is by counting the number of linearly independent paths through a program's source code. It is speculated that a source code with high cyclomatic complexity could induce software bugs due to the difficulties involved in manipulating and testing such complex code correctly (Lyu, 1996). If this is the case, then it is possible that cyclomatic complexity may also be a good estimator for the trustworthiness of a web application, so this comparison is relevant.

Table 5.1 Web forums ranked by Trustworthiness (TM).

#	Web Forum	Lines of Code	Avg. CC	Raw-NVR	Trustworthiness Metric (TM)
1	JGossip 1.1.0	34633	1,89	4	138,5
2	JForum 3	47650	1,43	8	93,4
3	JForum 2.1.9	61262	2,05	16	64,5
4	Yazd 3	56255	2,41	58	17,7
5	JavaBB 0.99	23807	1,49	41	10,2
6	mvnForum 1.2	76774	2,73	108	10,2
7	JSForum 0.0.2	1693	2,76	58	0,4

Up to now, we have not yet established the reliability of the proposed metrics, so we cannot assure that the order is correct; this will be addressed later in Section 5.2.3.3. However, we can start analyzing the relationship between the total number of security warnings (Raw-NVR), the Trustworthiness Metric (TM), and the average Cyclomatic Complexity (CC) of the benchmarked applications. At first

glance, the average CC does not appear to correlate well with any of the metrics. When it comes to CC and Raw-NVR, JGossip and JForum 3 have inverted positions, and JavaBB, which has a small CC, actually has a fair high Raw-NVR. The last two positions are also inverted regarding these two metrics.

When comparing CC with TM, even though the last two positions are the same, bigger differences in the metrics are observed. For example, JavaBB and mvnForum, while having the same TM values, also have dramatically opposite CC (one on the top and other at the bottom). Given these differences, the only conclusion possible is that if CC is a good estimator for the trustworthiness of code, then our metrics are not, and vice versa. In Section 5.2.3.3 we show that our metric has merit to compare applications, suggesting that CC is not a good trustworthiness estimator for security aspects.

Another important analysis is the comparison between Raw-NVR and TM. Although they present more or less similar rankings, like, for instance, in the three first positions, there are some crucial differences. Take for example the scores for JSForum and Yazd3. Even though they have exactly the same Raw-NVR values, they present very different trustworthiness values. This is mainly due to their relative sizes: Yazd3 is much larger than JSForum. Because they present the same number of warnings, JSForum has a higher warning density, which in principle may manifest as a high propensity to hidden vulnerabilities. The same rationale applies, in a smaller scale, to the differences between JavaBB and Yazd3. An interesting aspect is that, even though they have a quite different number of warnings, JavaBB and mvnForum ended up having the same trustworthiness. This means that, while they have different sizes and warnings, they present approximately the same defect density, so they have similar propensity to vulnerabilities.

As TM is essentially designed for comparison, the actual values of the metric are not meaningful, so absolute scores of 10 or 100 do not translate semantically into anything: what is meaningful are the relative values. If we compare the scores of each application with the others, we observe that the applications can be actually divided in three big groups: the first group is composed by the top 3 applications (JGossip, JForum 3 and 2.1.9), which have very high scores. The second group is comprised of the following 3 applications (JavaBB, mvnForum and Yazd3), which are separated from the first group by a factor of approximately four (calculated by dividing the TM of JForum 2.1.9, which is 64.5, by the TM of Java BB, which is 17.7). The last group includes only one application, JSForum, with a score of less than 1/20 of the worst score of the second group. Even though it is difficult to argue that an application within a given group is explicitly better (or worse) than the

others on the same group, the *difference* between each group is significant. The question now is whether this difference does map into real evidences; if it does not then the metric cannot be considered representative. To actually evaluate this aspect, we have compared this ranking with the assessment provided by the six security experts.

5.2.3.2 Benchmark Results *vs* Experts' Analysis

The final output of the assessment performed by each of the six experts was a table with their proposed ranking, which consists of a simple ordering accompanied by a qualitative description of the process they used to determine it. A key aspect is that no single pair of experts proposed the same ranking, which shows that individual human analysis may not be a good source for benchmarking, as the ultimate result is based on opinion and knowledge that varies from person to person, and that is, most likely, not repeatable (unless a detailed process is followed, as the one proposed in Chapter 4 for security benchmarking of transactional infrastructures).

In order to compare the experts' evaluations with the results of the trustworthiness metric, we need to have an agreement between the experts. Although several options could have been followed to achieve that agreement, we decided to consider a simple average between the rankings provided by them (similar to a voting scheme). Table 5.2 presents the ranking proposed by each expert, along with the average for all experts, and the values for the trustworthiness metric.

Table 5.2 Experts' rankings

Forum	Exp1	Exp2	Exp3	Exp4	Exp5	Exp6	Avg. Rank	TM
JGossip 1.1.0	3	2	3	2	6	7	3,83	138,5
JForum 3	1	1	1	1	1	1	1,00	93,4
JForum 2.1.9	4	3	4	3	2	2	3,00	64,5
Yazd 3	6	5	2	4	4	5	4,33	17,7
JavaBB 0.99	2	6	6	5	5	4	4,67	10,2
mvnForum 1.2	5	4	5	6	3	3	4,33	10,2
JSForum 0.0.2	7	7	7	7	7	6	6,83	0,4

There are several relevant aspects in this analysis. The most obvious is the unanimity regarding the first place, JForum 3, which was actually ranked in second by the benchmark. This does not invalidate our benchmark, as the scores of the three first positions are proportionally very close. What differs most is the fact that

JGossip, the first in the benchmark ranking, came as third in the average of the experts, which requires a more thoughtful analysis.

A close look to the scores provided by the experts shows that the first four put JGossip in the top 3 forums (which, in average, would actually put it in the second position), while experts Exp5 and Exp6 decided that it should be positioned in the bottom of the ranking, along with JSForum. By analyzing the experts' justifications, we can observe that both Exp5 and Exp6 did not take into account the source code specificities, which is actually the only aspect that is portrayed by our trustworthiness ranking. While experts 1, 2, 3 and 4 mention clearly the fact that JGossip is correctly designed, something that our metric expressed quite well, the justifications for the Exp5 and Exp6 rankings were threefold: lack of paid support, not being actively updated, and inexistence of a community of users capable of helping mitigating future security incidents.

The goal of our trustworthiness benchmark is to provide a metric able to help selecting the application that *is least likely to have security incidents in the future*, and not the one with better outside support when incidents occur. Obviously, this information cannot be extracted from the source code of the application, and it is unlikely that we will ever be able to include it in a security benchmark. While this is a complementary aspect that should, of course, be taken into consideration when selecting among applications, it does not invalidate the value of our automated benchmark.

Another key observation is that, although ignoring source code aspects, experts 5 and 6 still considered JForum 3 the best option. Their confidence is justified by the existence of an active development community and the offer of paid support. Obviously, source code quality cannot be directly related to this, suggesting that the experts ranking may be actually a coincidence. In fact, this coincidence is confirmed by the scores given to mvnForum, which was ranked in the second half of the raking by the first four experts, for reasons like: being "*less organized and maintainable*" and employing "*incorrectly prepared statements, using concatenations of values instead of parameters*". These characteristics clearly show that mvnForum is based on an insecure coding style, where a simple coding error may cause the introduction of vulnerabilities. However, experts 5 and 6 ranked it quite high based on the argument that an active community supports its development.

If the scores given by the benchmark for JGossip are too high, and should actually have been lower because JGossip lacks of an active community and support (which is the opinion of experts 5 and 6), than we could also argue that the scores that were

given to mvnForum by these same experts are also incorrect as they did not take into account the insecure coding style, something which was fairly expressed by our automated benchmark. The problem we are considering here is that even though we cannot automatize the identification of the fact that certain software does not have active community, we can automatize the identification of insecure coding patterns in the software. If the information provided by our benchmark was available to experts 5 and 6, then they would surely not consider giving mvnForum a ranking as good as they did and, at the same time, they would possibly consider the fact that JGossip is in fact securely designed. This discussion demonstrates how important is the kind of results that our benchmark provides when it comes to complement other types of analysis. Should experts 5 and 6 have an automated method to accomplish this technical evaluation, they would never fail in this regard.

Another important aspect that can be observed in Table 5.2 is the unanimous ranking given to JSGossip. As pointed by some of the experts (particularly experts 1 and 2), this application is crawled with vulnerabilities, and should never be considered for use because it has "*the worst design possible when it comes to security precautions*". Being a project abandoned since 2003, experts 5 and 6 also assigned low rankings to it. However, if they ever had to choose between JGossip and JSForum (both of which do not have active communities), only an automated tool like ours could point out how dramatically better-designed JGossip is. In fact, we do believe that positioning JGossip after JSGossip, as done by Exp6, is an indefensible mistake that should be prevented.

A key aspect that can also be noticed when analyzing the average rankings of the experts is that the three groups of applications suggested by our trustworthiness benchmark are exactly the same as the ones we could create based on the experts' rankings (even if we also include the biased evaluations given by experts 5 and 6). The top three applications (for both the experts and the benchmark) are JForum 3, JForum 2.1.9, and JGossip. The intermediary group is formed by mvnForum, JavaBB and Yazd 3. Finally, JSForum is isolated in the last position.

Looking closely to the rankings of the middle group (Yazd 3, mvnForum, and JavaBB), we can see that the experts that did consider source code evidences could not reach any kind of consensus regarding their relative ranking. Our benchmark could also not differentiate them very much: while all of them are not terribly designed, they are not good examples of secure design. In fact, all three present coding patterns with a "*propensity to the introduction of vulnerabilities*", as stated by one of the Exp1.

In summary, our benchmark ranking matched fairly well the joint opinions of the six experts. While one drawback of our method is the inability for evaluating the kind of support the users can get from the community, which is indeed an important aspect when evaluating some new software, it correctly considered and portrayed all source code aspects that our experts took into consideration. Actually, in the cases where the experts did not take source code information as basis for the ranking, some poor decisions were made. This shows that our proposal can help in benchmarking the trustworthiness of applications, by considering technical aspects regarding the source code, which may be far from the reach of administrators and users with reduced security knowledge. Although characteristics like *the existence of an active community* can be easily assessed by an administrator, technical details like the correctness and security of the design of an application begs for the use of an automated tool, role that our proposal seems to fulfill in an adequate manner.

5.2.3.3 Cross Validating based on Source Code Characteristics

To further understand and cross-validate not only the decisions of the experts, but also the behavior of the benchmark metric, we analyzed in detail the source code of the applications. The summary of our findings, together with our own qualitative ranking is as follows:

1) *JForum 3*. This application has the most secure design. This is mainly due the use of the Hibernate persistence framework (Hibernate 2011), which is well known for providing high protection against SQL Injection (OWASP 2010). The use of this framework appears to be correct; thus, it is very unlikely that there is a way to break the application.

2) *JForum 2 and JGossip*. Both of these applications perform database accesses through *prepared statements*, which are recognized by programmers as an effective method for protecting against SQL Injection (Amirtahmasebi 2009). The security is guaranteed by carefully using only constant SQL queries and by correctly passing values via parameters to previously prepared commands. No traces of vulnerabilities or bad design could be found during our analysis.

3) *Yazd 3*. This application also uses prepared statements, but, in various locations, external variables are directly concatenated to SQL query strings (i.e. system properties are directly appended to the query, without using parameters). The main input values, however, are passed through parameters. This construction is clearly more error prone than the others, and the risk of this design is in accidentally concatenate to a query a variable that

the programmer believes is a constant, but that is not, or whose value can be influenced by an attacker indirectly. A typical Yazd 3 query is as follows:

```
private static final String LOAD_USER_BY_USERNAME = "SELECT *
FROM " + SystemProperty.getProperty("User.Table")+" WHERE " +
SystemProperty.getProperty("User.Column.Username")+"=?";
```

4) *mvnForum* and *JavaBB*. Both applications concatenate input values directly to create SQL statements. Even though mvnForum uses prepared statements, the feature is useless due to this construction (i.e. no use of the query parameters). The application input parameters appear to be all validated before this concatenation, but all it takes to create a vulnerability is failing a single input validation, as no extra defenses are in place. Examples of such code constructions found in these applications are:

   mvnForum:

```
Collection globalPermissions= execSqlQuery("SELECT
Permission"+" FROM "+MemberPermissionDAO.TABLE_NAME+
" WHERE MemberID="+Integer.toString(memberID));
```

   JavaBB:

```
ResultSet rs = stmt.executeQuery("select downloads from
jbb_posts_files where file_id=" + fileId);
```

5) *JSForum*. This application has a large number of vulnerabilities, as input values are extracted from the HttpServletRequest object and concatenated directly, in String format, to the queries being built. No validation is done on the inputs. Most database access occurs like the following:

```
String RegUser = request.getParameter("user");
ResultSet rs=db.selectQuery("SELECT * FROM forum_users "+
"WHERE user_name=\""+ RegUser + "\"");
```

As can be seen, our evaluation also resembles the ranking provided by the proposed benchmark. The reality is that all evidence we gathered regarding our original hypothesis of using false positives as a coding quality estimator suggests that our original assumption is valid. In fact, the dangerous coding practices that we put forth as evidence for security or insecurity of the applications, are exactly what shaped the results of the analyzers and, therefore, of our benchmark.

5.2.4 Lessons Learned

Several lessons can be deduced from the experiments conducted, some related to the strengths of the approach and a lot related to the weaknesses. From the strengths of our benchmark, we immediately learned that using static code analysis tools to

perform trustworthiness benchmarking automatically guarantees several of the properties expected in any benchmark: repeatability, simplicity of use, portability, scalability, non-intrusiveness, and representativeness (which were discussed in Chapter 2, Section 2.4).

Repeatability, which is the ability of re-executing a benchmarking campaign and obtaining the same results (at least, in statistical terms), is guaranteed by the fact that SCAs are deterministic. If ran multiple times with same input, they report the same results.

Simplicity of use is another property expected in a benchmark. Static analyzers are applications that take source code as input and automatically provide as output a list of potential bugs/vulnerabilities. Because of this simple process, most static analyzers are naturally very simple to use. The automated analysis of the reports is also simple, as all tools provide them in XML format. This is also required in order to provide **scalability** to the benchmark, or the evaluation of the results would be unfeasible for applications with too big pieces of source code.

Fulfilling two additional properties of benchmarks, SCAs are naturally **non-intrusive**, as they perform a passive analysis of the provided source code, and **portable**, as they work over most source codes of a specific programming language (i.e. in our case, the approach will work to compare all applications that were designed in java, but will not be usable for other programming languages).

The most important property of all, however, is related to the **representativeness** of the results. Our analysis put forward evidence that a carefully chosen set of SCAs provide enough representativeness to be used for benchmarking the trustworthiness of real complex web applications. Although the evidence we present demonstrates this point specifically to Java, the construction should apply for all programming languages that have a good set of tools.

We also have to evaluate the weaknesses of the approach. The first thing is that we must account for the discrepancies in the validation experiment. By design, our benchmark can only take into account the characteristics of the software, and aspects like community support cannot be part of the benchmark (at least in an automated manner). The conclusion we can take from this is that trustworthiness benchmarking is actually an excellent tool to help in the decision of what software to choose, and in fact provides information that cannot be easily obtained. However, it is unlikely that it is possible to conceive an automated benchmarking procedure capable of guaranteeing the selection of the best alternative in all situations, without taking additional information into consideration.

Another important problem that we noticed is that the benchmark can only be used to evaluate source code developed without considering the benchmark specification, which is a huge problem to benchmark approaches. In section 5.3 we discuss why this is true, and why we need an approach that is not dependent on tools like static code analyzers.

Nevertheless, the lasting conclusion of our experiments is that, on average, coding styles can be correlated with security attributes by searching for evidences of secure coding best practices. Cross-validation and manual analysis suggest that such correlation is indeed useful to support the selection of secure web applications, even if source code metrics are not enough to account for all important aspects (e.g. outside support and active development).

5.3 Towards a General Approach for Trustworthiness Benchmarking of Web Applications

In the previous section we explored the use of expert analysis tools to build a practical and usable trustworthiness benchmark. The assumption is that the *false positives* of a good set of static code analyzers is a good predictor of the quality of the source code of a web application, and that too many false positives may be related, to a certain extent, to bad programming practices. In our experiments we provided evidence that this assumption is sound and that a benchmark built upon it could be sufficiently accurate, thus allowing the comparison of the trustworthiness of web applications.

There is, however, a contradictory aspect to that proposal: we exploit the failures (in the form of false positives) of otherwise good static analysis tools to obtain information that the tools were not designed to provide in the first place. Assuming that such a benchmarking approach becomes a common standard, two effects should be considered in the future:

1) As static analysis tools become more precise at their task, which is finding actual vulnerabilities while avoiding false positives, they will progressively contribute less and less to the benchmark. For instance, an ideal tool able to find 100% vulnerabilities and report 0 false positives would not contribute to a better benchmark, as detected vulnerabilities would not contribute for the calculation of the trustworthiness metrics (real vulnerabilities are used only for qualification purposes, as explained in Chapter 3).
2) In order to improve software rankings, the techniques that software developers employ would shift to the ones that more efficiently avoid false

positives. However, the tendency could be for such coding practices to only be better at avoiding false positives, nothing more, and therefore would not lead to more secure coding practices. False positives tend to be correlated with bad coding practices only if the software is not developed with such a benchmarking context in mind, which is a problem in the long run. In practice, the benchmark can be *gamed*, in the sense that developers can improve the metrics for a given application without improving its quality in the way that the metrics are intended to portray.

In this section we build upon these aspects and propose a process to design a benchmarking tool able to accomplish the specific goal that static code analysis tools accomplish only as a side effect: *how to evaluate if a web application coding style is prone to security vulnerabilities or not*. Even though our goal is not build a complete ready to use benchmark, we will present the main requirements needed for building one.

5.3.1 Web Applications Code Threat Vectors

As explained in Section 5.1, a web application threat is a set of parameter crafting techniques aimed at leading the application to behave in a malicious way. These techniques focus particular types of lines of code, designed for specific purposes, which we call **hotspots**. When subjected to the crafted input data, an insecure hotspot behaves in way that does not conform to the application business rules. Therefore, threat vectors can be defined as sets of programming practices that either facilitate those crafting techniques or that block them. In this initial proposal we focus on two of the most important web applications threats, namely:

#SQL Injection threat

a) *description*: crafting techniques aimed at modifying semantically a target SQL command that is sent to a backend database.

b) *hotspots*: any line of code which submits a SQL command to a database.

#Cross-Site Scripting

a) *description*: crafting techniques that lead the application to send executable code to a client that expects only textual information. This executable code may comprise *scripting* code or embedded applications (e.g. activeX, flash, etc.). The malicious executable code may be stored for later retrieval or be immediately reflected back to the client.

b) *hotspots*: any line of code that sends an output to the client application.

5.3.2 Security Precautions in Web Applications

A representative trustworthiness benchmark depends of properly identifying the source code characteristics that distinguish a secure software from an insecure one, and therefore we must understand those characteristics in more detail. The relevant security precautions that can be applied in the context of web applications are divided in two groups (Liu 2006): *general input validation* and *strong business data typing* for hotspots. The problem is how to find enough evidence indicating that both types of precautions are being applied in a source code (and to what extent they are being applied). Our approach consists of looking for code patterns typically used to implement these security precautions in order to prevent the considered threats. The next sections provide an overview of the patterns being considered in our proposal.

5.3.2.1 General input validation

General input validation can be done using three major algorithmic approaches: *accept known good*, *reject known bad*, and *transform invalid into valid*.

The *accept known good* approaches (sometimes called *whitelist filtering*) include any strategy that implements a "*if not exists in, then remove/reject*" semantic. The "remove" part of this approach might be implemented as the complete replacement of the value by a known good value, therefore completely ignoring the actual value used as input. Implementing this kind of validation usually requires only information about the input domain of the application (which the developer is expected to know). This is an important strategy that is considered the safest type of validation, as it is the one that offers the developer more control over the inputs. Several code patterns are associated with this strategy, including:

- Enforcing strong variable data types
- Match against a regular expression
- Algorithms implementing a "*if not exists in, then remove/reject*" filter
- Out of range check/set to known good
- Out of length check/set to known good
- Empty/null check/set to known good

The *reject known bad* approach (or *blacklist filtering*) is comprised of strategies that try to enumerate exhaustively bad values that input parameters can have and try to remove/reject these values. Basically, it includes any kind of algorithm implementing an "*if exists in, then remove/reject*" semantic. The main problem is that the set of values used to validate the inputs may be extremely difficult to define

and maintain. The reason is that inputs considered as acceptable in a certain moment might become bad in the future due to technology evolution and context modification. Also, bad inputs frequently depend on specific threats and attack techniques, meaning that more information besides the business domain of the application may be required. However, in some cases this approach may be easier to implement than an *accept known good*, as it may be impossible to specify all the "known good" values for a certain input parameter.

Transforming invalid into valid (also known as *massaging the data*) is used when a combination of the two strategies above is applied. This consists of situations where only parts of the data are bad (but not all the data are) and it is not possible to simply replace a "contaminated" input with a known good value without losing information. This approach is comprised of any "*replace x by y within z*" algorithm, and is based on the ability to separate the bad parts of the input from the good parts. This is the technique most difficult to implement due two key aspects:

1) *It may not be easy to identify the bad parts of the input*. This issue presents the same problems of the "reject known bad approach", but with an additional difficulty: the bad data is mixed with the good data, and therefore a simple comparison may be insufficient.
2) *The replacement algorithm may be difficult to implement in a secure way*. This happens because whenever some bad piece of data (let's say X) is replaced by some good piece of data (let's say Y), then this good piece may lead to the creation of another piece of bad data (i.e. Y might not be universally good, and may become bad when included in the context previously occupied by a certain X).

The correct implementation of each of these three input validation approaches demands different degrees of control and knowledge from the developer. Accept known good algorithms are relatively simple to implement correctly, as most of them depend only what the application is expected to do (i.e. the business rules of the application). Reject known bad approaches are more difficult to develop and maintain correctly, as they depend on knowing what are the bad values, and these are related not only with information about threats, but also with the techniques used to accomplish such threats (i.e. the real attacks). Transformation techniques are the hardest of all to implement securely, as not only they depend also on threat information, but also on the context where the bad input values might appear.

These difficulties require the definition of a hierarchy of what would be the preferable ways of implementing the validation of a particular input parameter. Therefore, we argue that *accept known good* approaches are usually better then

reject known bad approaches, which are better than *transformation* approaches. In practice, this hierarchy is based on the previously presented characteristics of each approach and the control and knowledge required for implementing them.

However, the possibility of using the best available approach depends on the application being developed, as some applications may have input parameters that do not have a clearly identifiable “good” form. For instance, arbitrary files and free text input frequently have an open form that may be extremely hard to match against a “known good” format. Validating these inputs may require the use of the other approaches, and therefore knowledge about threats and attack techniques.

5.3.2.2 Strong Business Data Typing for Hotspots

As defined before, hotspots are the lines of code in a web application that are the target of an attack (Integrigy 2007), and protecting hotspots should be done by enforcing a strong business data typing for all variables used as input to the hotspot. The main idea is that whenever the values used in a hotspot conform to its corresponding business data typing, then the hotspot will behave as expected.

Reliable protection of hotspots requires the developer to know the business data type for each hotspot. If the hotspot is a function call, this requires knowing exactly the domains of parameters that are expected by the function is expecting and guaranteeing that no value outside those domains is processed. Also, if there are business restrictions for such parameters, then should also be considered as part of the business data typing for the hotspot. For instance, for most DBMS engines a string containing an *unescaped* single quote is not a valid string in the context of a SQL execution call, as it may change the semantics of the command. The same is true for a numeric value containing text characters.

Enforcing strong business data typing can be done in several ways. The input validation algorithms presented in the previous section can also be used to validate variables of hotspots according to its business data type. However, depending on the business data type of the hotspot and on the algorithm used to validate its input, threat information may be required to properly design a correct validation algorithm.

Depending on the case, automated methods for enforcement of business rules may be available when it comes to the technical aspects of a hotspot. For instance, *parameterized queries* can be used in a SQL command call to guarantee that, independently from the input passed to the database, the semantic of the SQL command does not change. In this sense, the call itself will force variable data typing (accept known good) in such a way that no semantic change of the SQL is

possible. Automated enforcement methods should be preferred against the manual development of validation algorithms, as the developer has more control over them and the probability of error is lower.

5.3.3 Accounting for Secure Coding Practices

For each threat vector defined in the benchmark (SQL Injection and Cross Site Scripting as described in Section Web Applications Code Threat Vectors), we need a set of coding best practices consensually accepted as being able to reduce or eliminate the probability of malicious effects of threats. However, in contrast to our approach for trustworthiness benchmarking of transactional systems infrastructures presented in Chapter 4, we should now look for practices directly related with each threat, so the correlation is quite obvious. However, field research is always necessary, as explained next.

In practical terms, the process is based on the analysis of the hotspots and their relation with the input variables. For each security recommendation, we provide specifications of the preprocessing and post-processing activities that should be implemented to each particular value used in the context of the hotspot. We call these specifications *variable accountability statements*, which are aimed at the variables that "carry" the values from the input to the hotspots. Based on the discussion in Section 5.3.2, three general types of accountability are defined: *business data typing*, *automatic enforcement*, *general input validation*.

Accountability statements can be either positive statements (that, when applied, tend to improve the trustworthiness of the code) or negative statements (that, when not applied, lower the trustworthiness of the code). Additionally, hierarchies of recommendations may generate interrelated accountability statements, which may represent positive and negative statements simultaneously. In fact, although using lower quality alternatives (i.e. not preferred solutions) is positive (better than not using anything), it is also negative due to the existence of better solutions that could have been applied (e.g. removing known control characters from a string is a good practice, but a better choice would be to allow only known good characters instead of removing only the bad ones, therefore this would be a good and bad practice simultaneously).

A bibliography study (including, but not limited to (Cenzi 2009, CGI Security 2010, Fonseca 2007, Howard 2006, Integrigy 2007, Jovanovic 2006, OIWASP 2010, Seacord 2006) regarding typical countermeasures against the threats considered in this benchmark yielded several recommended security best practices (again, we remember that researching for security best practices is an error prone

task, and therefore the list should be periodically evaluated and updated for further use). In the next paragraphs we present those general recommendations and their translation to accountability statements. The accountability statements have the weights indicated before their description. Most statements have weight 1 (+1 or -1), but some negative statements have weight -2 and -3. The reason is that the application of these practices only occurs when some other preferred method is ignored. For instance, the accountability statement C for Cross-Site Scripting ("*Variable does not output any of the characters ><()&# as is*") has weight -3. Indeed, if the program outputs those characters, then the variable is not being addressed by any of the following filters: accept known good (only known good values are accepted), reject known bad (known malicious values are rejected), and transformation (invalid values are transformed into valid values). The three missing alternatives results in a -3 weight.

#SQL Injection prevention recommendations

- Use strongly typed parameterized query APIs, either by applying the mechanisms provided by the programming language or using stored procedures (provided by the database backend).
- Validate input parameters and enforce correct data types.
- Properly escape values used in dynamic queries (i.e. query construction through concatenation).

#SQL Injection variable accountability statements

Strong Business Data Typing

A) (-1) Business data typing is enforced; strings are escaped according to the DBMS characteristics

Automated Enforcement

B) (-1) Variable is not concatenated to the SQL statement

C) (+1) Variable is assigned through a proper parameterized assignment function

General input validation

D) (+1) Variable has its length/range checked; it is rejected or set to a known good value if the length/range checking fails

E) (+1) Variable is subjected to a "*if not exists in SET, remove/reject*" filter algorithm or regular expression

F) (+1) Variable is checked for empty/null values; it is rejected or set to a known good value if the empty/null checking fails

G) (-1) Variable is subjected to at least one *accept known good* validation algorithm (i.e., statements D, E or F)

H) (+1)(-1) Variable is filtered using *reject known bad* algorithm

I) (+1)(-2) Variable is filtered using *transformation* algorithm

#Cross-Site Scripting prevention recommendations

- Enforce proper character output encoding (e.g., UTF-8).
- Validate input parameters, enforcing correct data types.
- Escape output according to the output context (e.g. HTML section, CSS section, script section, etc.).
- Avoid the output of any of the following characters `><()&#` if not as HTML entities.

#Cross-Site Scripting variable accountability

Strong Business Data Typing

A) (-1) Business data type is enforced

B) (-1) Variable is outputted with an enforced fixed character encoding

C) (-3) Variable does not outputs the characters **`><()&#`** as is

General input validation

D) (+1) Variable has its length/range checked; it is rejected or set to a known good value if the length/range checking fails

E) (+1) Variable is subjected to a "*if not exists in SET, remove/reject*" filter algorithm or regular expression

F) (+1) Variable is checked for empty/null values; it is rejected or set to a known good value if the empty/null checking fails

G) (-1) Variable is subjected to at least one *accept known good* validation algorithm (i.e., statements D, E or F)

H) (+1)(-1) Variable is filtered using *reject known bad* algorithm.

I) (+1)(-2) Variable is filtered using *transformation* algorithm.

Variable accountability statements are as simple as possible, turning the verification of their implementation into an easy task. This is aimed towards making the benchmark application as easy as possible (and certainly much simpler than a deep vulnerability analysis). This simplification however, may have some drawbacks. For instance, take as an example the accountability statement C for Cross-Site Scripting (i.e. "*Variable does not outputs any of the characters ><()&# as is*"). Actually, depending on the section of HTML code where they are inserted, some of these characters may be harmless. But assuming that a certain character is harmless in a certain section also *assumes that the developer has absolute control* over where he is outputting them. Verifying if a character is outputted is relatively easy, but verifying if it is outputted in a harmless section is significantly more difficult. Our benchmark proposal follows a pessimistic approach in these cases, assuming that developers may make mistakes. Therefore, they should always avoid outputting these characters. If it is impossible to avoid it, then the application is penalized. Nevertheless, for the sake of comparison, if it is impossible to not output them, then all other equivalent applications will also find it impossible, therefore being penalized too.

5.3.4 Trustworthiness Metrics

The benchmark defines five complementary metrics that characterize different trustworthiness and untrustworthiness aspects of the benchmarked code:

- **Average Code Prudence (ACP)**: the sum of the average positive accountability statements applied to the hotspots. This metric expresses how much precaution the developer employed in the hotspots benchmarked.
- **Average Code Carelessness (ACC)**: the sum of the average negative accountability statements not applied to the hotspots. This metric expresses, on average, how careless the developer was on the hotspots benchmarked.

- **Average Code Quality (ACQ)**: the sum of the positive aspects of the code and the negative aspects of the code, yielding an overall comparison metric for the code general quality concerning the threats of the benchmark. This metric can also be computed for each hotspot, providing a way to compare the hotpots of the same application in a relative way, highlighting the ones that have lower quality (thus may deserve more focus in terms of improvement efforts).
- **Hotspot Prudence Discrepancy (HPD)**: this is the standard deviation of the ACPs of all accounted hotspots. This metric portrays the consistency of the developer (or developers) in his prudence or tendency to harden some parts of the code, but not others.
- **Hotspot Carelessness Discrepancy (HCD)**: this is the standard deviation of the ACCs of the hotspots, portraying how much inconsistent the developer (or developers) is when considering negative accountability statements.

The algorithm that should be used to compute each of these metrics includes five main steps:

- **Step 1**. For each threat, scan the applications to identify the lines of code that comply with the description of the hotspots.
- **Step 2**. For each hotspot, list the variables used. Select all the variables whose value depends directly or indirectly on an external source of data. External sources are: a) direct input from a user or call, b) values read from a database, c) values read from local files. If there is no variable whose value depends on any these sources, then discard the hotspot.
- **Step 3**. Compute the **partial metrics** for each variable in the non-discarded hotspots by evaluating the path followed by the value from the external source to the hotspot. Considering this path, evaluate all the variables affected against all the accountability statements of the benchmark. Applied positive accountability statements count +1 multiplied by its weight; not applied negative accountability statements count -1 multiplied by its weight. The metrics for the hotspot are proportional to the number of variables involved in the hotspot, which is as follows (for each hotspot and each threat):

Hotspot ACP	=	(ΣPositive statements)/Number of variables
Hotspot ACC	=	(ΣNegative statements)/Number of variables

Hotspot ACQ = Hotspot ACP + Hotspot ACC

- **Step 4**. Compute the overall code metrics as follows:

Code ACP = (Σ Hotspots ACP)/Number of hotspots

Code ACC = (Σ Hotspots ACC)/Number of hotspots

Code ACQ = Code ACP + Code ACC

– **Step 5.** The discrepancy metrics are computed as follows:

$$HPD = \sqrt{\frac{\Sigma\,(\text{Each Hotspot ACP} - \text{Average ACP})^2}{\text{Number of hotspots accounted}}}$$

$$HCD = \sqrt{\frac{\Sigma\,(\text{Each Hotspot ACD} - \text{Average ACD})^2}{\text{Number of hotspots accounted}}}$$

For each pair threat/hotspot the metrics should be interpreted as follows:

- **Hotspot ACP**: higher values mean that more security precautions against the threat are present in a given hotspot.
- **Hotspot ACC**: low (negative) values mean that the hotspot has characteristics that typically yield vulnerabilities.
- **Hotspot ACQ**: higher values denote that more security precautions are evident in the hotspot.

For the overall code, the metrics and their relative interpretations are as follows:

- **Code ACP**: higher values show that more security precautions against the considered threats are present in the overall benchmarked code.
- **Code ACC**: low values suggest insecure coding practices.
- **Code ACQ**: higher values show that more evidence of security best practices is present in the code.
- **Code HPD**: higher values denote a developer that is more inconsistent when protecting hotspots.
- **Code HCD**: higher values show that the developer is more inconsistent when avoiding dangerous coding characteristics.

Like the very idea of measuring trust and trustworthiness, these metrics are not absolutely precise, meaning that the confidence on the results increase with the difference on the scores (e.g. the most distant are the scores of two evaluated pieces of software, more confidence we may have that the one with higher score is better designed than the one with lower score). It is important to emphasize that the overall code metrics are defined in a way that the number of hotspots and the number of variables will not influence them. Also, we propose a set of discrepancy metrics, in the form of standard deviations, to complement the analysis of the main metrics, as simple averages sometimes may hide important irregularities in the distribution of the values.

5.3.5 Preliminary Experimental Evaluation

To demonstrate the ideas behind the proposed benchmarking approach, we compared two distinct implementations of an application in terms of the SQL injection threat. The implementations we decided to compare were the ones that were developed for the experimental evaluations presented in Section 5.2.2 one developed by a graduate student and the other one by an experienced developer, both implementations of the TPC-App web services benchmark (TPC 2011). In the context of these experiments, the application developed by the experienced developer was called V_0 and the other is referred to as V_1.

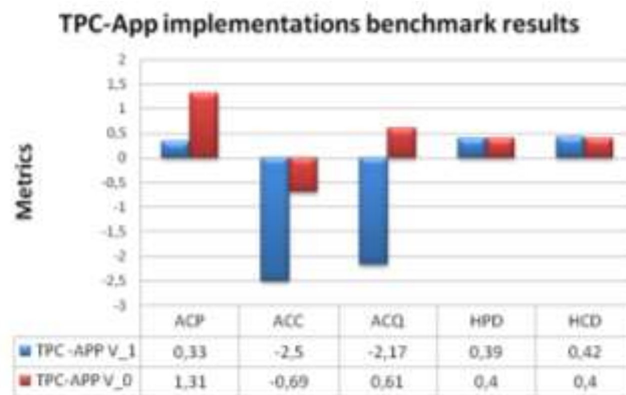
As the proposed approach is yet in a preliminary stage (the goal is yet to assess its applicability), we did not implement any tool to compute the metrics automatically, so we conducted a manual code inspection to execute the benchmark. Even though this is not ideal, it is enough to illustrate the concepts.

The first step of the analysis consisted of finding the hotspots. The two implementations use a JDBC connector (Bales 2001) to access the database, therefore the analysis started by finding all lines of code that invoked the methods *executeQuery* and *executeUpdate* (Bales 2001). In version V_1 these methods receive a string as parameter, which is traced back to a SQL command with several concatenations. In version V_0 no concatenation is found, and the variables are passed through parameterized assignment functions. In both applications, there are 6 hotspots (the number is the same in the two cases, as both versions implement the same standard specification), and in both hotspots 2 and 5 could not be traced back to any input source (they were constant SQL commands), so these were discarded from the analysis.

We proceeded to examine all variables directly or indirectly related with the remaining hotspots concerning the 9 SQL Injection accountability statements

presented in Section 5.3.3. For this experiment, the Business Data Typing was either numeric or free text for all table fields. Figure 5.5 presents the overall benchmarking results. By analyzing the final values, we can see that, for V_1, a huge penalization is given to the code, as the ACC score is higher (in absolute value) than the ACP score, while we see the inverse for V_0, certainly due to the use of parameterized queries. Nevertheless, it is clear that improvements could have been done to V_0, as several penalizations are still present (e.g. input values are not filtered in any way).

Overall, version V_0 is better in all metrics: better Average Code Prudence, lower Average Code Carelessness, and higher Average Code Quality, meaning that this version is trustworthier than V_1. The discrepancy metrics are similar, meaning that each developer took more or less the same considerations across all hotspots.



	ACP	ACC	ACQ	HPD	HCD
TPC -APP V_1	0,33	-2,5	-2,17	0,39	0,42
TPC-APP V_0	1,31	-0,69	0,61	0,4	0,4

Figure 5.5 Overall benchmark results

Although further research and validation is needed, this small experiment suggests that the proposed benchmarking approach is useful and may be applicable in practice. A tool implementing this algorithm would be a reliable replacement for the benchmark based on the outputs of static analysis tools presented in Section 5.2.

Although the goal of the benchmark is to compare applications that provide similar functionalities, its use is not limited to applications that implement the same specification. In fact, the metrics simply state how careful, on average, the developers were on protecting the hotspots of each application, thus any two applications are comparable (obviously, considering the same threats). Given an automated tool to compute the values, we could easily see it being used to choose between several brands of wikis or forums, for instance.

As a final remark, we would like to emphasize that the automation of the tool is a technical problem that, although requiring a lot of work, will not pose any theoretical difficulties. In practice, parsing the code, identifying the hotspots, and tracing the execution path between the hotspots and the entry points (in a similar way to what is done by static analyzers (Jovanovic 2003)), would provide the required support to build a graph representing the transformations suffered by the values of the variables (e.g. filtering, escaping, concatenation, etc.). This graph could then be used to find transformation patterns for the accountability statements, thus getting the information required to calculate the benchmark metrics. Nevertheless, this automation is out of the scope of our thesis, and is proposed as future work.

5.4 Conclusion

This chapter studied the problem of trustworthiness benchmarking of web applications, as a representative use case of transactional system business applications. We first investigated the idea of using tools that already exist, in this case *static code analyzers*, to perform automated trustworthiness benchmarking. We started from small controlled experiments, and finished with an evaluation of the proposal in a representative use case, which was the selection of web forums applications. We validated the results by cross-checking them with the manual analysis of real security experts, finding out that our automated evaluation resulted in an assessment equivalent to that of the experts. A set of limitations that were identified on the approach conducted us to the proposal of a general approach for the trustworthiness benchmarking, which was tested in small-scale preliminary evaluation, nevertheless showing promising results.

The most important result of this chapter is related to the effective correlation between source code characteristics and the security quality of software. Basically, our experiments clearly suggested (particularly by the validation done by the experts) that the way software is designed allows gathering a trustworthiness measure that is related with the presence or the absence of pro-active measures to avoid programming vulnerabilities.

The limitations of a benchmark based on static code analyzers, especially considering the effects of the evolution of the tools, should not be taken lightly. An inevitable conclusion is that even if this approach works for now, it wont keep working forever, particularly if developers notice that their software is being evaluated using such metrics. This is why a more generic approach is relevant. We believe that the approach proposed for automation of the benchmark would be a huge step towards the creation of a sustained solution to the task of trustworthiness

benchmarking of web applications. But even if we have sketched the most important steps in the design of such tool, we understand that the difficulties in doing so are clearly considerable, and therefore the problem is not closed.

6

Selecting Software for Transactional Systems Infrastructures

Chapters 5 and 6 were dedicated to the study of methodologies, approaches and actual implementations for trustworthiness benchmarking, in the context of two fairly representative scenarios: complex environments, namely, transactional system infrastructures, and web-based business applications. In both cases, even though we stressed the importance of separating security benchmarking in two parts (as a way of coping with the tangible and the intangible aspects of security), we did not propose detailed approaches for security qualification, leaving this step open for further investigation. The reason for this was already presented in the respective chapters, and has mainly to do with the fact that most of the research done nowadays on security mechanisms (and also on vulnerability scanning and prevention technologies) can be used as part of a security qualification step, which lead us to focus on the most promising step: trustworthiness benchmarking.

There is, however, one aspect of security qualification that calls for further attention under the structure of our framework: to implement a transactional system infrastructure (i.e. a complex structure with many separate parts and that can several distinct configurations) we need to select a *DBMS engine*, which is in charge of providing all the *transactional system business capabilities*. However, this selection step may not be easy due to the complexity of such software.

Today, several representative DBMS engines exist, for instance, Oracle, SQL Server, PostgreSQL, MySQL, etc., thus the selection of the better one in terms of security is a key aspect that should be considered if one aims to have the best transactional system infrastructure possible. Theoretically, under the context of the framework proposed in Chapter 3, the selection of the specific DBMS engine to be used in a infrastructure would call for a security benchmark in the lines of the benchmarking approach proposed in Chapter 5, even though the set of threat vectors for this case would still have to be studied, as they are clearly not the same of that of web applications. This is actually quite obvious, as we certainly do not want the engine to present vulnerabilities detectable by automated scanners and, if possible, we want it to be developed in a way that has a low probability of introducing hidden vulnerabilities.

At the same time, we should consider the other requirement of security qualification, which is to answer the following question: *what security mechanisms should the engine provide in order to be acceptable as an alternative?* We have already established in Chapter 3, Section 3.2.1 that the selection of a set of security mechanisms for security qualification in any benchmark is primarily domain dependent, and therefore changes with each specific business domain. For example, in the case of web applications, even though we can define a list of possible security mechanisms that can be required from typical web applications, for each security mechanisms there is always a situation where it is not necessary.

The goal of the security mechanisms of a DBMS engine is very clear: *to help improving the security of the transactional system infrastructure*, which is exactly what our trustworthiness benchmark measures. Therefore, if we use as reference the trustworthiness benchmark for transactional systems infrastructures proposed in Chapter 4, then we may extrapolate a list of security mechanisms that would help improving the security of a real live installation. Pursuing this path, though, requires taking into consideration a few restraining factors, namely:

1. Alternate layers of security may compensate for any security mechanism not provided by a DBMS engine. In the worst case, a software wrapper could be placed around the DBMS engine providing the missing mechanisms. Therefore, the absence of a mechanism does not imply that implementing the corresponding security precautions is impossible. Unless otherwise required for a specific domain, a single missing security mechanism does not necessarily make a DBMS to fail qualification.
2. We can, on the other hand, assume that if a mechanism is present, then the fact that we do not have to compensate for its absence leads, at the very

least, to a *decrease* in the configuration complexity (which leads to a lower probability of introducing interaction vulnerabilities and also vulnerabilities on the “compensating” mechanisms). Therefore, having mechanisms available directly in the software is better than not having them.

3. The existence of a security mechanism in the software has no relation in the final security of the infrastructure as a whole. For the mechanism to have any effect after deployment, it has to be used correctly, otherwise it is useless and may even decrease the overall security (one classical example of this effect is when a software is set to block authentication attempts after a certain number of authentication failures - a mechanism that can be used for Denial of Service attacks - and the number of allowed attempts is very high). In other words, the existence (or not) of a security mechanism in a given software product has no effect in the trustworthiness benchmarking assessment (thus, it should be considered during the qualification step).

We also have to take into attention another characteristic of today’s DBMS engines: their security is highly tied to the characteristics of the underlying operating system. This becomes clear when we look at the security recommendations identified in Chapter 4, where several of them are specific to the operating system, even though it is a “transactional system security recommendation”. Therefore, instead of selecting a DBMS engine, we deal with the selection of an entire *software package,* which in our case is the composition of a DBMS engine and an operating system. As it will be made clear in our experimental analysis, the security mechanisms available in a specific DBMS engine vary with the underlying operating system even for the same engine brand.

In the following sections we present the methodology used to devise and calibrate a list of the security mechanisms that should be implemented by DBMS engines for supporting the security practices identified in Section 4.3.2. The methodology includes the following general steps:

- Each security recommendation for a transactional system infrastructure is mapped into a desirable system state (*System State Goal*) that represents the state of the system when the recommendation is being correctly applied.
- That state goal is analyzed in order to identify the series of steps that must be used to obtain such goal (the *Mechanisms Goals*).
- Each of the steps is analyzed to evaluate which of them can be automated and therefore be supported by security mechanisms provided by the software.

The application of the benchmark results in a metric that represents an estimation of the aggregated importance of the mechanisms present and available in the package under benchmarking. Additionally, the procedure allows computing a gap analysis matrix that can be used to compare the actual security features of a set of software packages with the features that would have to be provided to fulfill all the security recommendations.

To demonstrate the approach, we benchmark seven distinct software packages that could be considered representative candidates for use in transactional systems installations. These packages are based on four different DBMS engines (Oracle 10g, SQL Server 2005, PostgreSQL 8, and MySQL Community Edition 5) and two different operating systems (Windows XP and Red Hat Enterprise Linux 5). We evaluate their main characteristics using gap analysis, and draw some general conclusions regarding their advantages and deficiencies.

It is important to emphasize that the results obtained are not supposed to be used alone to decide what is the best software package for a database installation, especially outside the context of our security benchmarking framework. Particularly, **what we provide here is a benchmarking tool that can be used for security qualification support, and not a trustworthiness benchmarking tool**. As part of the qualification step, several other factors should also be considered (e.g. cost, performance, availability, and familiarity), but those are out of the scope of this work. The reason is that, although there are tools to help evaluating several of these factors, evaluating the security capabilities og a software package is still an open problem.

This chapter is divided as follows. In Section 6.1 we discuss our methodology that we used to identify a list of security mechanisms that could have been implemented by the evaluated software packages. Section 6.2 presents a discussion of how to establish the potential impact that the identified mechanisms could provide to the security of the infrastructure. In Section 6.3 we present the benchmark metrics and execution process. In Section 6.4 we present the results of the evaluation of seven software packages done using our benchmark, discussing the most important conclusions that our tool is capable of. Section 6.5 concludes the chapter.

6.1 Identifying Security Mechanisms

The list of security recommendations used as the base for the trustworthiness benchmark for transactional systems infrastructures presented in Chapter 4, was also used to extrapolate the security mechanisms needed to fulfill those same

recommendations. However, this process was not trivial, requiring several steps of careful analysis, as detailed in the following paragraphs.

We started by analyzing the 64 security recommendations (see Table 4.2 and Table 4.3), where each recommendation was classified in terms of the type of support needed for its implementation, namely:

- *Hardware support*: recommendations that require either specific hardware components or a specific physical setup for the underlying hardware;
- *Network support*: recommendations that require the network to have some specific setup or characteristic;
- *Plain policies*: general guidelines that do not require any mechanism in particular, and are just behaviors that should be enforced;
- *OS support*: recommendations that require some features of the operating system;
- *DBMS support*: recommendations that require some specific DBMS features;
- *Third party support*: recommendations that require complementary software not usually found in a basic database software package (DBMS and OS).

Table 6.1 presents the number of best practices that were classified in each class. Note that some practices have been classified in more than one class, which explains why the second column of the table adds to more than 64 practices. This first classification allowed us to focus on the practices that required at least some support from the software components (a total of 51 out of 64 security practices), which is the focus of our approach.

The next step consisted of *rewriting the recommendations* in a way that allowed more clearly identifying the security mechanisms needed to support them. The original recommendations were stated as actions that should be conducted on the system to enhance security. However, these actions may contain several factors that may be implicit in their statements such as: what are administrators' responsibilities, what actions require software support, and what the environment dependent elements are. This way, instead of trying to identify security mechanisms directly from the recommendations, we decided to use two intermediary steps to help exposing these implicit factors (obviously, these steps could have been bypassed, but the process of explicitly performing them clearly allowed us to achieve more effective results).

Table 6.1 Classification of databases security best practices in regard to their requirements

Requirements	N. of Practices
Network Requisites	2
Hardware Requisites	4
Plain Policies (no software requirement)	10
OS Support	28
DBMS Support	38
Third-Party Support	2

In the first step we restated each of the best practices as a *System State Goal* representing the state of the system in a point in time when the practice is being correctly applied. For instance, one of the best practices related to the operating system configuration is stated as follows: "*Remove from the network stack all unused/unauthorized protocols*". A system state goal for this best practice is: "*The OS network stack has no unused/unauthorized protocol active*". Notice that, although obvious in some cases, this rewriting step moves the focus from the action to the consequences of the action. This is extremely important to disclose the fundamental effects that are expected when applying a best practice. Additionally, as several practices can actually be applied in several software components at the same time (e.g. password related practices must be applied at both OS and DBMS levels), this rephrasing forced the distinction to be made clear, allowing us to identify the practices for which more than one System State Goal should be defined (i.e. one for each of the components of the software package).

When analyzing the System State Goals it became easier to start distinguishing the effects of the practices that are exclusively administrators' tasks (e.g. defining what are the unauthorized protocols) from the ones that can be fully automated, and therefore can be supported by security mechanisms. From a high level perspective, any security practice is a policy that requires an action from the administrator (in the sense that he can always choose to not implement it), and can typically be automated to a certain point. For instance, the administrator may manually check if the users' passwords are strong enough, but a piece of software may also perform this check automatically (and also prevent users from choosing weak passwords in the first place). Obviously, maintaining the System State Goal in the first case (manual verification) is much more difficult than in the second case (when automation is present). In fact, it is widely accepted that the least work the administrator has to do to enforce security policies, the better is his productivity and the higher are the chances that these policies are correctly implemented. Thus, to identify the mechanisms needed to support a security practice, first we need to

know what are the steps required for achieving the System State Goal, which is done on the next step.

In the second step we rewrote again the System State Goals, but this time in terms of what we called *Mechanisms Goals*. In this additional step we break the System State Goals in the list of actions that would lead to the System State Goal. Mechanisms Goals can be seen as the functions that make the steps towards the accomplishment of the System State Goal as simple as possible (i.e. the complexity of the steps becomes hidden behind automation). Continuing the previous example, the Mechanisms Goals for the "*the OS network stack has no unused/unauthorized protocol active*" System State Goal can be described as two simple steps: "*Identify active protocols*" and "*disable unauthorized/unused protocols*". Note that, defining what the unauthorized/unused protocols are is environment dependent and can only be done by the system administrator. However, identifying the active ones and allowing them to be easily removed from the stack can be done by software mechanisms that may help accomplishing the System State Goal.

The identification of the security mechanisms based on the Mechanisms Goals was then quite straightforward, as can be seen in the example above. An important issue is that, in some cases, more than one mechanism may be required for the state goal to be accomplished. In other cases, different mechanisms may be used to accomplish the same goal, possibly with different amounts of automation. Alternative ways for performing the same tasks are useful to suit different administrators, environments and requirements. Table 6.2 presents a few examples of the mapping of security best practices into System State Goal and Mechanisms Goals. The complete list can be found in (PhD Thesis Complementary Info 2012).

Table 6.2 Examples of the mapping between security best practices, system state goals and mechanisms goals.

Security Recommendation.	Component	System State Goals	Mechanisms Goals
Remove from the network stack all unauthorized protocols	OS	The OS network stack has no unused/unauthorized protocol active.	Identify active protocols and disable unauthorized/unused ones.
Change default passwords	OS	No OS *userid* password is the default.	Prevent the installation of default passwords in the OS or allow

			identification and removal of default passwords.
	DBMS	No DBMS *userid* password is the default.	Prevent the installation of default passwords in the DBMS or allow identification and removal of default passwords .
Do not delegate privileges assignments	DBMS	Privileges a user have should not be delegated.	Prevent users from delegating their privileges or identify the use of privilege delegation operations.
Keep the software updated	OS	No patches provided by the OS vendor are unapplied.	Not allow an available OS patch to remain unapplied.
	DBMS	No patches provided by the DBMS vendor are unapplied.	Not allow an available DBMS patch to remain unapplied.
Restrict database OS *userid* access to everything it does not need	OS	The database OS *userid* has access only to DBMS software.	Set privileges to the dedicated DBMS *userid* to access only DBMS software.
		The database OS *userid* has access only to designated peripherals.	Set privileges to the dedicated DBMS *userid* to access only the defined peripherals.
Prevent idle connection hijacking	DBMS	Remote connections drop when unused for some period of time.	Set connections to timeout after a period of inactivity.
Change/remove default *userids*	OS	The OS has no default *userid* operational.	Prevent the existence of default *userids* in the OS (during or after the installation).
	DBMS	The DBMS has no default *userid* operational.	Prevent the existence of default *userids* in the DBMS (during or after the installation).
Make regular backups of the data	DBMS	There is an up-to-date copy of the DBMS data in a safe storage.	Make updated copies of all DBMS data.
Avoid ANY and ALL expressions in privileges assignments	DBMS	No user has privileges assigned from ANY and ALL expressions.	Prevent or warn the use of ANY and ALL expressions on privileges assignments.
Ensure no “side-channel” information leak through configuration files	OS	Configuration files do not contain sensitive information.	Avoid the inclusion of sensitive information in configuration files.

The whole process can be summarized as follows:

1. Rewrite the security recommendations in the form of *System State Goals* that describe the system when the recommendation is correctly being applied. In this step it is necessary to clarify to which component of the software package (e.g. DBMS or OS) the goal refers to.
2. Determine the associated *Mechanisms Goals*, which represent the steps required to achieve the System State Goal in terms of functions provided by the software.
3. List exhaustively the mechanisms that can be used to implement (partially or fully) the Mechanisms Goals.

By following this process we have identified the 112 security mechanisms, which are presented in Tables 6.5, 6.6 and 6.7. The first column of each table describes

the mechanisms that a target software component (second column) is expected to facilitate. The mechanisms should be read as "*The software provides automated support for…*", and are not tied to any specific product, being described in a broad way to allow a posterior assessment of their existence in the software packages under benchmarking.

6.2 Establishing the Impact of Security Mechanisms

After devising the list of expected security mechanisms for a database software package, an obvious problem arises: some mechanisms are more relevant than others in terms of security. This is the same problem that we had to address when developing our trustworthiness benchmark, as explained in Chapter 4.

One certainty is that the impact of a mechanism is directly related to the security recommendations that it allows to implement. This way, our proposal is to inherit the impact of the mechanisms from the relative weights computed for the corresponding recommendations. The problem, however, is not exactly the same, as the role of a mechanism within the context of a security recommendation varies, and while a recommendation may be important, a mechanism used to implement it may provide only partial support.

For each mechanism, we identified in which class its security recommendation could be found in our relative weight computation, and we assigned values ranging from 1 to 4 to each of the classes (first column in Table 4.5, in Chapter 4). It is important to emphasize that, although we computed specific weights for all recommendations, we used them only as a reference to find the high-level class of the mechanisms (ranging from 1 to 4). The reasoning is that the fine-grain comparison would not hold for a large number of environments as aspects like the usability and reliability of each mechanism in each package could not be measured. Furthermore, small differences (e.g. of 0.01%) could hardly mean anything in terms of impact and should be discarded. Nevertheless, the high-level class can be used as a reference to compare the mechanisms for most of the environments, always realizing that mechanisms within the same class are considered to have the same relative impact (e.g, the function of "Automated installation of OS pending patches" and the ability of "store credential information using a reliable encryption scheme" are both considered of the same relative impact because they ended with the same impact weight, even though they are completely different and unrelated security mechanisms).

As mentioned above, in some cases, security mechanisms may provide only partial support for the security recommendation, and may need to be complemented. This should be reflected in the weighting process, and can be solved using two

alternative approaches: either we value mechanisms that provide partial support only when their complementary counterparts are also present or we count them always as providing half of the support (having half the weight of the original importance). We opted for the second of the two alternatives due to the simple fact that, even though a complementary mechanism might not exist in the package, the existence of a partial mechanism may already help the administrator, in the sense that usually it can be used for supporting part of the recommendation implementation. Notice, however, that counting partial mechanisms as "half" is another issue open for discussion. The problem is that determining how much a mechanism actually fulfills of the recommendation (e.g. 80% of the practice or 30% of the practice) is generally impossible as this depends also on other resources that may or may not be available to the administrator (which may vary from case to case). We decided that, for the purpose of the benchmark, partial mechanisms provide on average half the support, even if under the certain conditions of real environments that might not be the case.

Another problem is that some security mechanisms can be used to support multiple best practices. In this case the choice is between emphasizing the importance of these mechanisms or not. In other words, we had to decide if the importance of a given mechanism should be somewhat accumulated for different practices. For instance, *should a mechanism required to implement three not very important practices be considered more important than another mechanism that can be used to support one single very important practice?* In this case, we decided that the best approach would be that yes, it should. We strongly support the idea that security should be exhaustive, meaning that, from a general perspective, in a trade-off decision, the higher is the number of security precautions in place the better. We are aware that this can be disputed, particularly when considering special situations where an excess of security mechanisms may cause more problems than that they solve, but the assumption seems to be overall reasonable. Anyway, this decision has a small impact on the overall benchmark, as there is a very small set of mechanisms that are related to more than one security practice (3 to be exact).

The impact weight of each mechanism was computed by multiplying the best practice importance class (from 1 to 4) by the weight of the support of the mechanism (1 or 0.5). The individual weights (i.e. the weights per best practice) for the mechanisms that may contribute for the implementation of more than one practice were then added, resulting in weights ranging from 0.5 to 5. Table 6.3 presents the mechanisms with the highest impact. The complete list of mechanisms can be found in (PhD Thesis Complementary Info 2012).

Table 6.3 Most important security mechanisms identified

Security mechanisms (automated support for…)	Target	W

Disabling access to extended functions.	DBMS	5
Configuring the system to always encrypt a remote connection to the DBMS.	DBMS	4
Encrypting the connection of native developer applications.	DBMS	4
Removing systems privileges of DBMS *userids*	DBMS	4
Restricting read/write privileges of a partition to a specific *userid*.	OS	4
Automated installation of DBMS pending patches.	DBMS	3
Automated installation of OS pending patches.	OS	3
Configuring the DBMS to store credential information using a reliable encryption scheme.	DBMS	3
Configuring the OS to store credential information using a reliable encryption scheme.	OS	3
Defining all DBMS passwords during the installation phase.	DBMS	3
Defining all DBMS *userids* in the installation phase.	DBMS	3
Defining all OS passwords during the installation phase.	OS	3
Defining all OS *userids* during the installation phase.	OS	3
Relying the DBMS on an outside specialized authentication mechanism.	DBMS	3
Relying the OS on an outside specialized authentication mechanism.	OS	3
Removing privileges of users over systems tables.	DBMS	3
Warning DBMS users, in an password change operation, that their new passwords are weak and cannot be accepted.	DBMS	3
Warning OS users, in an password change operation, that their new passwords are weak and cannot be accepted.	OS	3

6.3 Benchmark Metric and Execution

The purpose of the proposed benchmark is to allow the comparison among alternative software packages in terms of security capability. To this end, the benchmark provides two complementary outcomes: a *Security Mechanisms Compliance* metric (SMC) that portrays the level of compliance of the package in regard to the set of security mechanisms devised from the established security recommendations, and a gap analysis matrix that allows identifying exactly what are the mechanisms missing in each package (for implementing a given configuration).

Applying the benchmark is a process that consists of verifying which of the 112 security mechanisms are included in the software package, build a gap analysis matrix, and calculate the security compliance metric. First, the benchmark user must check whether each security mechanism is present on the software package being analyzed. This provides a list that can be used to build a gap analysis matrix that allows visually comparing several alternative packages in term of their overall capabilities compliance (in Section 6.4.3 we provide some examples of how to use such gap analysis matrix to draw important conclusions about the evaluated packages). The *security compliance metric SMC* is then computed as the sum of

the weights of all the security mechanisms present in the package. Note that this number must be interpreted carefully, as a higher value does not necessarily means a more secure product: it means that it offers more support for implementing security best practices in the context database infrastructures.

6.4 Experimental Evaluation

In order to demonstrate the possibilities of our tool, we used it to benchmark a set of software packages, and identify their characteristics and capabilities. In the following sections we describe the experiments and analyze the results obtained.

6.4.1 Software Packages Assessed

For the experimental evaluation we decided to consider a representative set of database solutions that can be found in the field. From the DBMS engines perspective, we selected two commercial DBMS engines, namely Oracle 10g and Microsoft SQL Server 2005, and two open source ones, namely PostgreSQL 8 and MySQL Community Edition 5. Oracle and SQL Server are two of the most widely used commercial DBMS, and these particular versions account for a representative number of installations in the field. PostgresSQL and MySQL account for the majority of DBMS installations that use open source software, and are very popular alternatives to commercial software.

From the operating system perspective, we used the same rationale, therefore choosing Microsoft Windows XP and Red Hat Enterprise Linux 5. Both operating systems are widely representative choices to support the DBMS mentioned above, but we are aware that several other alternatives would be interesting as well (e.g. Suse Linux and Microsoft Windows Server 2003, among many others).

Excluding Microsoft SQL Server 2005, that is only available over Windows platforms, the other three DBMS could be installed over both operating systems. The overall results of the evaluation of the seven different software packages are presented in Table 6.4.

6.4.2 Comparing the Software Packages

Besides using experts' knowledge, to apply the benchmark to the software packages selected we had to install them and analyze thoroughly their corresponding documentation. The goal is basically to evaluate if a given package has native support for each of 112 security mechanisms defined by the benchmark.

A fundamental difficulty was to determine what elements were provided by the software package as a whole in contrast to determining the elements provided by each product individually. Password policies are one example where the platform

influences the capabilities of the DBMS. For SQL Server 2005, password policies can be inherited from the operating system only if it is installed over Microsoft Windows 2003, and not if the system is based on Windows XP due to the lack of interfaces for this system. On the other hand, PostgreSQL can use the Pluggable Authentication Module (PAM) features of Linux, which comes in the standard installation of the Red Hat Enterprise Linux 5, and therefore is available for the package at both the OS and the DBMS levels. This kind of detail can make the process to be relatively costly in terms of information gathering, though the outcome justifies the work.

Table 6.4. Overall results of the experimental evaluation of the 7 different software packages.

Package N.	DBMS Engine	Operating system	MP	SMC	%
1	SQL Server 2005	Windows XP	79	131,5	76%
2	Oracle 10g	Red Hat Enterprise Linux 5	74	118,5	68%
3		Windows XP	73	118	68%
4	PostgreSQL 8	Red Hat Enterprise Linux 5	73	123	71%
5		Windows XP	68	114,5	66%
6	MySQL Community Edition 5	Red Hat Enterprise Linux 5	66	110	64%
7		Windows XP	66	110,5	64%

Table 6.4 presents the overall evaluation of the packages. The first and second columns identify the components of each package and the third column presents an identification number for the package (that will be used later in Table X to refer to each package). The fourth column presents the total number of mechanisms present (MP) in the package, and the fifth column presents the *Security Mechanisms Compliance* metric (SMC) of the benchmark (sum of the importance of all mechanisms present). Finally, the sixth column presents the metric in terms of a percentage of the maximum value possible for an ideal package including all the mechanisms.

Among the evaluated packages, the one that includes more security mechanisms is Package 1, SQL Server 2005 over Windows XP. This means that it has more native support for implementing security best practices for databases. Notice that the plain number of mechanisms present does not say much about the importance of such mechanisms. For example, Package 4 has a SMC higher than Package 2, even though it has less security mechanisms available. This happens because the security mechanisms present in Package 4 are generically considered more important than the ones present in Package 2.

Based on the SMC metric, the best package benchmarked is Microsoft SQL Server 2005 over Windows XP. A key aspect that supports this result is an overall better

integration with the operating system (allowing, for instance, using the Windows Update mechanism for keeping the DBMS software up to date with little intervention). The feature of *client application roles* (that allows to better support the development of applications with the ability to identify the end users behind database connections based on database authentication) and some extra backup features not present in the other DBMSs also contributed to this result. However, the score for all the packages is not that different, which suggests that, in general, these packages (operating systems and database engines) tend to implement the same type of security features and mechanisms (despite being open source or not). The worst scored package was MySQL Community Edition over Red Hat Enterprise Linux 5.

6.4.3 Software Packages Gap Analysis

This section presents and discusses the results from a gap analysis point-of-view, serving as an example of the full potential of the proposed tool. We start with an overall analysis of the set of mechanisms available and then move to the analysis of the mechanisms present in all packages, the mechanisms not available in any package, and, finally, the mechanisms available only in some of the packages.

6.4.3.1 Overall Analysis of the Mechanisms and Packages

The first observation regarding the overall analysis is the number of mechanisms related to each of the two software components that are part of a software package (i.e. the OS and the DBMS). As shown in Figure 6.1, more than a third of the 112 mechanisms identified are provided by the OS, which confirms what we suggested several times before, i.e. despite the DBMS engine being used, security is strongly tied to the capabilities of the underlying platform. Even more important is the fact that, for several DBMS, the provision of some security mechanisms is highly dependent on the operating system being used (e.g. some authentication features of PostgreSQL are only natively provided if the operating system has the Pluggable Authentication Module (PAM) installed, which is, for instance, available on Red Hat Enterprise Linux 5, but not on Windows XP). It is then clear that, from a security point of view, the two software components must be selected simultaneously.

The next important global observation is the general availability of the 112 mechanisms in the analyzed packages. Figure 6.2 presents the percentages of mechanisms available in all packages, mechanisms available in none of the packages, and mechanisms available in at least one package. As shown, little more than half of the mechanisms are supported by all the packages analyzed, which is much lower than what one would expect. Worse than that is the fact that 21% of

the mechanisms are not provided by any of the packages analyzed. This suggests that many security recommendations cannot be easily implemented (or additional software has to be acquired for their implementation) due to the inexistence of support from the DBMS and/or OS in all the packages analyzed.

Mechanism Type

Figure 6.1 Mechanisms by component of the analyzed packages.

Figure 6.3 breaks down the number of mechanisms supported by combinations of packages. Interestingly, a very high number of mechanisms appear on a minority of the packages (e.g. 20 mechanisms appear on only three or less packages). This suggests that these mechanisms, although provided by some packages, are not considered universally important (e.g. column level privilege settings).

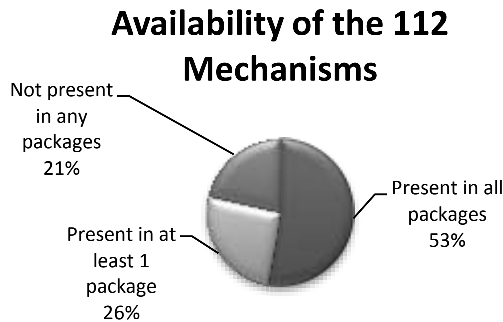


Figure 6.2. Availability of mechanisms

The last general observation is related to the total number of mechanisms provided by each software package (presented in Table 6.4). Although package number 1 clearly presents the biggest number of mechanisms, the actual number of mechanisms available in the seven packages does not vary considerably (79 in the

most and 66 in the least). This suggests that vendors follow some common trends when deciding what mechanisms should be made available in their products.

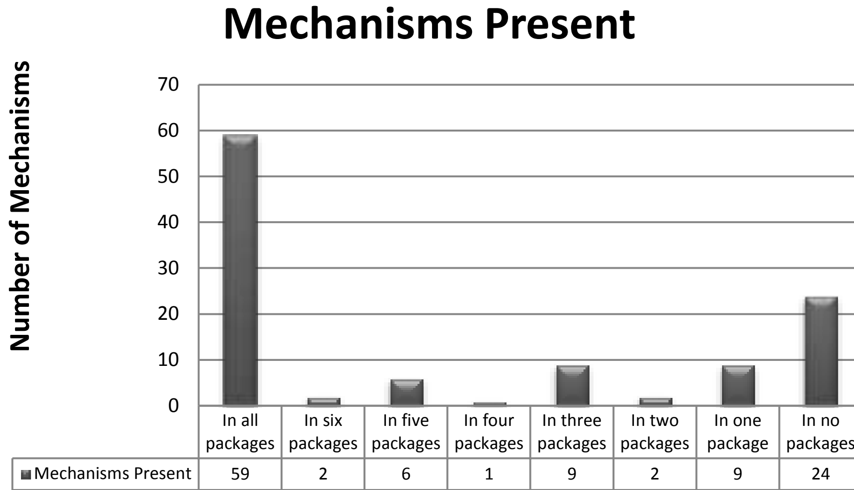


Figure 6.3: Number of mechanisms available across packages.

6.4.3.2 Mechanisms Available in All Packages

Table 6.5 presents the list of 59 mechanisms that are provided by all the packages. The first observation is that there are 28 DBMS mechanisms and 31 OS mechanisms in this group. This fact, together with the total number of mechanisms initially identified for the DBMS (70 mechanisms) and the OS (42 mechanisms) components, shows that the operating systems analyzed implement a higher percentage of the expected security mechanisms than the database engines, suggesting that operating systems vendors may be more concerned about helping the users in hardening their systems than the DBMS vendors are on helping DBAs to harden their database infrastructures. The operating system is in fact a more fundamental layer of software than the DBMS, as it is prepared to support a diversity of distinct purposes, contrary to the DBMS that serves a particular use. On the other hand, this does not justify more concern with security. In fact, although operating systems may also host critical information and services, the business purpose of DBMSs is specifically aimed at storing user information and data, which may be even more critical than a certain service that an operating system may provide. In summary, the security of both layers of software is equally important, and this disparity cannot be easily justified.

Note that mechanisms such as password settings, privilege settings, some installations choices, and the definition of some general operational parameters, are allowed by all packages, which confirms that these mechanisms are accepted as universal requirements for databases. Very few informational mechanisms, however, can be found in this group. For example, the easy verification of the current working state and configuration of the system is NOT a universal concern of DBMS and OS vendors.

Table 6.5 List of mechanisms available in all packages

Security Mechanism (*The package offers support for...*)	**Component Target**
Disabling access to extended stored procedures and functions	DB
Config. the system to always encrypt a remote connection to the DBMS	DB
Encrypting the connection of developer applications	DB
Removing system privileges of DBMS *userids*	DB
Restricting read/write privileges of a partition to a specific *userid*	OS
Automated installation of OS pending patches	OS
Configuring the DBMS to store credential information using a reliable encryption scheme	DB
Configuring the OS to store credential information using a reliable encryption scheme	OS
Defining all DBMS passwords during the installation phase	DB
Defining all OS passwords during the installation phase	OS
Relying the OS on an outside specialized authentication mechanism	OS
Warning OS users, in a password change operation, that their new passwords are weak and cannot be accepted	OS
A DBMS authentication procedure that requests only credential information to the remote users	DB
An OS authentication procedure that requests only credential information to the remote users	OS
Configuring the DBMS so only administrators have access to log information	DB
Denying login into the OS from a credential with more than a specified number of failed authentication attempts	OS
Forcing the OS users to change their passwords when they're older than a specified time frame	OS
Identifying systems privileges of DBMS *userids*	DB
Setting read/write/execution privileges over files	OS
Setting that a *userid* cannot login	OS
Setting who can change configuration files	OS
Setting who can change environment variables	OS
Using custom defined SSL certificates for encrypted connections	DB
Changing OS *userids* already in use	OS
Changing passwords of DBMS *userids* already in use	DB
Changing passwords of OS *userids* already in use	OS
Creating an OS *userid* with limited privileges	OS
Creating file systems partitions	OS
Identifying users with privileges over systems tables	DB
Making a backup copy of the database	DB
Storing the backup in a custom storage place	DB
Using a privilege limited *userid* to successfully load a DBMS process.	OS
Warning the administrator that there are OS vendor patches remaining to be applied	OS
Allowing the DBA to not use ANY and ALL expressions	DB
Allowing to explicitly state that a particular privilege cannot be delegated	DB
Changing listening TCP/UDP ports	DB
Changing remote identification information already in use. (e.g., SID)	DB
Configuring the system to always establish connections through the same TCP/UDP ports.	DB
Defining all remote identification information during the installation phase	DB
Disabling the generation of core_dump files	OS

Disabling the generation of trace files	DB
Preventing specifying sensitive information in configuration files. (e.g., not require specifying password in configuration files, etc.)	OS
Preventing the general use of sensitive information in systems variables	OS
Setting and discarding a complex password for a *userid*	OS
Setting the owner of files	OS
Specifying important events which occur in the OS that should generate a finger print	OS
Specifying privileges in a database level	DB
Specifying privileges in a table level	DB
Warning OS users that their passwords are older than a specified time frame	OS
Writing procedures that generate a trace for data changes	DB
Creating stored procedures	DB
Creating views	DB
Disabling a network protocol	OS
Identifying active protocols in the network stack	OS
Removing a database	DB
Selecting a different partition for OS log information	OS
Selecting a different partition than the main OS partition for DBMS log information	DB
Selecting a different partition than the main OS partition for the data files	DB
Setting/unsetting read/write/execute privileges over files	OS

6.4.3.3 Mechanisms Not Available in Any Packages

Table 6.6 shows the mechanisms that could not be found in any of the packages. The vast majority of the mechanisms in this group are specified by the actions of *Identifying (8)*, *Testing (4)*, *Warning (4)* and *Blocking (3).*

Identifying mechanisms are expected to easily provide general information about the system state and configuration. Not having these mechanisms forces the administrator to guess if a given setting is active or not, to create miraculous queries over poorly documented system tables, to analyze gigantic and cryptic configuration text files, or to read enormous manuals to find the information. Obviously, to help DBAs improving security, obtaining this kind of information should be as simple and intuitive as possible.

Testing mechanisms are mechanisms designed to verify either if some important operation was carried out successfully or if it will execute successfully when attempted (e.g. data backups and software updates, respectively). Testing is crucial to guarantee the system availability (either at the moment of execution of such maintenance task or in the future), but it is simply disregarded by developers of both operating systems and databases.

Warning mechanisms provide security related notifications. As these warnings may be a hindrance when the system is known to be working as expected, it should be possible to turn them off. However, when turned on they report information about important operations that should not occur normally. Providing such warning mechanisms is simply not considered in any of the packages analyzed. (e.g.

warning about outdated backups or about the modification of configuration parameters).

Blocking mechanisms are configuration options that result in some operations not being allowed. In this case, the blocking mechanisms that were not found in any package are related to privilege delegation. We believe that these mechanisms (although optional) are important because they allow the DBAs to better track how privileges are distributed within the database. For instance, whenever a particular user is the owner of some entity, he can decide who can access his entity and how. In critical security scenarios, however, the DBA may want to control this kind of delegation even about entities not owned by him, and this cannot be done in any of the DBMS analyzed unless the DBA owns all objects.

As can be seen in Table 6.6, most of these mechanisms are security specific and are not related to any major functional aspects of databases. As they simply do not provide any obvious functional advantage to DBAs that are not security experts, it seems that they are not considered as adding a significant *Return of Investment* value to the software. However, the importance of security in databases nowadays should be enough for vendors to consider these kinds of features from a perspective of not losing credibility in the future.

Table 6.6 List of mechanisms not available in any of the packages

Security Mechanism (*The package offers support for...*)	**Component Target**
Defining all OS *userids* during the installation phase	OS
Removing all privileges of users over all systems tables.	DB
Configuring the OS so only admins. have access to log information	OS
Identifying DBMS *userids* with default passwords	DB
Identifying default DBMS *userids*	DB
Identifying default OS *userids*	OS
Identifying OS *userids* with default passwords	OS
Testing the installation of DBMS new patches	DB
Testing the installation of OS new patches	OS
Warning the administrator that the last OS backup is not up-to-date anymore	OS
Blocking non-DBAs from delegating their privileges	DB
Blocking privileges not inherited from groups/roles	DB
Blocking the usage of ANY and ALL expressions in privileges granting	DB
Encrypting backups with a reliable encryption algorithm	OS
Identifying available functions that interact with the operating system	DB
Warning the administrator if any important configuration or file was modified	OS
Identifying available extended functions in general	DB
Identifying available functions that can be used to perform network operations	DB

Identifying available functions that can be used to read/write in the file system	DB
Identifying example databases	DB
Testing if a recently created backup correctly restores the database data to its corresponding state	DB
Testing if a recently created backup correctly restores the system to its corresponding state	OS
Warning administrators of ANY and ALL expressions used in privileges assignments	DB
Warning admin of users with the power of delegating their privileges	DB

6.4.3.4 Mechanisms Available in Some Packages

This group includes the mechanisms that exist in at least one package, but not in all of them (see Table 6.7). We can divide this group in two subgroups: the mechanisms that are *present in most of the packages* (four or more packages, corresponding to a total of 9 mechanisms) and the ones that are *present in just a few packages* (three or less packages, corresponding to a total of 20 mechanisms). These two subgroups seem to arise from two distinct situations.

Most mechanisms of the group *present in most packages* appear to be widely considered as important. In most cases, they are not present in some packages for very clear reasons, namely: specific platform migration decisions and feature inheritance from old versions. In other cases, vendors opted for excluding some mechanisms, but openly admit the lack of support (e.g. inexistence of groups/roles in packages 6 and 7). Note that, knowing if a particular mechanism is important for a particular environment should influence the decision of what is the best package for it.

Table 6.7 List of mechanisms available in some of the packages (X means that the mechanism is available in the corresponding package)

Security Mechanism (*The package offers support for...*)	**Component Target**	**Package 1**	**Package 2**	**Package 3**	**Package 4**	**Package 5**	**Package 6**	**Package 7**
Automated installation of DBMS pending patches	DB	X						
Defining all DBMS *userids* in the installation phase	DB	X						
Relying the DBMS on an outside specialized authentication mechanism	DB	X	X	X	X	X		
Warning DBMS users, in a password change operation, that their new passwords are weak and cannot be accepted	DB				X			
An authentication procedure for remote clients that identify individual end users instead of individual applications	DB	X						
Configuring the system to drop idle connections after a specific period of inactivity	DB		X	X	X	X	X	X
Configuring the system to require that remote clients have the correct server certificate installed	DB	X			X	X	X	X
Denying login into the DBMS from a credential with more than a specified number of failed authentication attempts	DB				X			
Forcing the DBMS users to change their passwords when they're older than a specified time frame	DB				X			
Specifying privileges in a row/value level	DB		X	X				

Changing DBMS *userids* already in use	DB	X			X	X	X	X
Making a backup copy of the OS which can be used to restore the environment to its current state	OS	X		X		X		X
Using a privilege limited *userid* to successfully install the DBMS.	OS		X		X		X	
Warning the admin that the last data backup is not up-to-date anymore	DB	X						
Warning the administrator that there are DBMS vendor patches remaining to be applied	DB	X						
Auditing a variety of important DBMS events	DB	X	X	X				
Auditing data changes	DB	X	X	X				
Config. the DBMS so only DBAs have access to audited information	DB	X	X	X				
Configuring the system to always establish connections through the same TCP/UDP ports during the installation phase.	DB	X			X	X	X	X
Defining listening TCP/UDP ports during the installation phase	DB		X	X	X	X		X
Preventing the installation of a database example during installation	DB		X	X	X	X	X	X
Removing quotas over systems areas	DB	X	X	X				
Setting privileges to groups or roles	DB	X	X	X	X	X		
Specifying important events which occur in the DBMS that should generate a finger print	DB	X	X	X				
Specifying privileges in a column level	DB	X						
Warning DBMS users that their passwords are older than a specified time frame	DB				X			
Identifying users with quotas over systems areas	DB	X	X	X				
Selecting a different partition than the main OS partition for auditing info	DB	X	X	X				
Setting/unsetting access privileges over peripherals	OS		X		X		X	

The mechanisms of the group *present in just a few packages*, on the other hand, do not seem to be considered universally important. Take, for instance, *setting privileges at row level*, only available in packages 2 and 3. It seems that it is not seen as a relevant feature, as this kind of privilege filter is usually carried out by the client applications themselves. However, it might happen that client applications do not use this feature exactly because it is not usually available, and not the other way around. Using a feature implemented directly by the DBMS is often more reliable than implementing them at the application layer. Therefore, providing these mechanisms allow the development of systems that are less error prone than the ones that have to implement specific tailored solutions.

In order to understand if there is any pattern behind the distribution of the mechanisms provided by a subset of packages, we explicitly analyzed the number of common mechanisms in each possible combination between the seven packages. This analysis is presented in Table 6.8. When looking to the mechanisms from this point of view, the fact that Packages 1, 2 and 3 provide uniquely 7 mechanisms, and Package 1 provides uniquely 6 mechanisms stands out. On the first case, most of these mechanisms are related to auditing, which is only provided by the commercial DBMSs analyzed (Oracle and SQLServer). Open source databases do not usually provide these mechanisms. In the second case, SQL Server database stands out by providing a few features that no other DBMS provides (e.g. some types of backup warnings, more installation options, column level privilege settings and a few automatic updates facilities). This helps confirming the fact that this

DBMS has most security mechanisms implemented out-of-the-box, as was portrayed by the analysis presented in Section 6.4.3.1.

Table 6.8 Mechanisms available only in specific sets of packages

Set of packages	Number of mechanisms provided uniquely by this set
Packages 2,3,4,5,6,7	2
Packages 1,2,3,4,5	2
Packages 1,4,5,6,7	3
Packages 2,3,4,5,7	1
Packages 1,3,5,7	1
Packages 2,4,6	2
Packages 1,2,3	7
Packages 1,4	1
Packages 2,3	1
Package 1	6
Package 4	3

In summary, all the security mechanisms identified in this work should be seen as being important, even if they are not usually used by most applications. Taking into account the current situation, where the set of mechanisms implemented by each available package is defined by factors not necessarily linked with the requirements of the end users, the analysis presented in our work seems to be very useful in helping clarifying and deciding which package, or set of packages, are fit for a particular target environment.

6.5 Conclusion

This chapter revisited the problem of security qualification in transactional systems infrastructures, first discussed in the context of the security benchmark for transactional systems infrastructures proposed in Chapter 4. The goal here was to design a benchmarking tool able to help analyzing and selecting specific software components that would help in securing complex infrastructures like transactional systems, where the identification of vulnerabilities and actual attack paths is not an easy problem. The need for such tool arises from the fact that a transactional system infrastructure can only have a proper security assessment after deployment, leaving the problem of selecting the components that will be part of this infrastructure unsolved.

The proposed methodology allows assessing the effectiveness of software packages, considering the security mechanisms that they make available for the administrator to secure the infrastructure, and favoring the ones that help the most in such task. We evaluated a set of real software packages regarding their ability for helping securing live installations, and were able to put into evidence a very

large set of security characteristics that the most representative DBMS engines have today. We also found a set of mechanisms that are not included in any of the benchmarked engine, demonstrating that our tool is able to provide relevant information of the assessed targets (namely, a matrix to support gap-analysis). The list of absent mechanisms we identified is particularly interesting, as it shows that the set of security mechanisms included in the evaluated software packages vary only slightly, being mostly the same in each version. It may be the case that the inexistence of procedures like the one we proposed in this chapter makes it difficult for software vendors to become aware of which security mechanisms would help the administrators in the field.

It is important to remember, however, that the list of mechanisms presented in this chapter was directly derived from the list of security recommendations devised in Chapter 4. Therefore, it may also suffer from the deficiencies already pointed out in that case (e.g. incompleteness and/or deprecation by change of technologies). As a matter of fact, the list of mechanisms brings no additional security information, as it is simply another perspective from the same knowledge that we already had in the original list of recommendations. We believe that this is one of the great merits of this methodology, to demonstrate how to reason about security in a consistent and methodical manner, taking security information provided by reliable experts in one end and, by assuming that this information is correct and sound, examining all the consequences of such information, deriving important conclusions and interpretations that allow it to be used in a variety of distinct perspectives.

7

Conclusions and Future Work

The importance of benchmarking of computer systems in general is growing with the diversity of solutions and software implementations. This is a natural consequence of the importance that computer systems are having in our society, given by the boost in efficiency and productivity that they provide to every single area of our lives. With our growing dependence on computing systems, the necessity of considering their security becomes unavoidable.

This thesis brings two major contributions to the fields of benchmarking and security in general. The first is a generally applicable security benchmarking framework suited for the definition of security benchmarks in any application domain. The framework is based on the observation that the course that research on benchmarking has been taking over the last years does not seem appropriate when considering security aspects. In essence, the research on benchmarking had its roots on performance of computer systems, where the goal was to have a measure of how efficient the system was at executing tasks. The most successful general model for performance benchmarking was based on the idea of modeling the work and the stress that the system under test would be subjected to in the form of a typical *workload*, and such workload allied with a set of performance metrics (e.g. number of tasks execute per amount of time) would allow a fair comparison of different systems. TPC and SPEC benchmarks are the most notable organizations that provide recognized standard performance benchmarks based this approach.

However, in the last decade, the research community noticed that performance benchmarks were not sufficient to realistically compare systems, at least not in a variety of practical scenarios. In fact, the results of performance benchmarks are

only correct when the system operates under no degradation effects, essentially under ideal circumstances. Assuming that systems fail, and that the overall execution environment is not ideal, dependability benchmarking appears as an approach to evaluate how a system degrades under faults. In other words, dependability benchmarking is the idea of measuring the degradation when operating under faulty conditions. The general model followed to accomplish this goal was based on the adaptation of the performance benchmarking model, by adding two elements: a faultload, which represents the set of faults that the system would typically suffer during its normal lifetime, and a set of dependability metrics.

Resilience benchmarking appears today as the last research endeavor in benchmarking, advancing the idea of the faultload to a *changeload* in which the assumption is that the problems that systems will face in the field are much more broad than typical faults, ranging from physical resources stress and limitations to workload fluctuations. Resilience benchmarking research is also starting to deal with the fact that computer systems are becoming progressively more self-adaptive, and that adaptations mechanisms are designed exactly to deal with the complex working conditions in which systems operate. Evaluating the performance and effectiveness of these adaptations mechanisms is a very complex problem that is still being researched.

With the success of this benchmarking approach, based on workloads, faultloads, and changeloads, one could expect the same idea to also apply to security. In fact, the Amber consortium (Assessing, Measuring and Benchmarking Resilience) delivered a research roadmap that clearly promoted the idea of identifying representative “attackloads” and security metrics, with the goal of defining security benchmarking using this approach.

Throughout this thesis, and particularly in Chapters 2 and 3, we presented several reasons that show why the traditional benchmarking approach is not the ideal one when it comes to security. The main argument is that the information that we get from identifying a vulnerability in a system, and therefore a potential attack, is not the same information we get when we subject a system to faults. This comes from a fundamental differences between faults and attacks that unavoidably has to be taken into account when comparing systems. For example, although the triggering of a fault may have a certain distribution probability, the triggering of an attack is much more complex to define as it depends on a malicious person that may or may not have interest in attacking the system. Accounting for the exploitation of known vulnerabilities must be completely different from accounting for the triggering of faults.

A proper security benchmarking approach must necessarily take into consideration a set of aspects that normally are not taken into consideration in other types of benchmarks: there are lots of uncertainties about the system, the environment, and the attackers. We believe that modeling unknown security problems in the same way we model known/detectable vulnerabilities is an error that ultimately leads to useless benchmarks or misleading conclusions. From a high-level view, we may say that the framework proposed in thesis essentially provides a way for reasoning about *how to correctly rationalize security aspects when the goal at hand is fair comparison*.

The framework itself was built upon two main ideas, and therefore was conceived with two main phases: security qualification and trustworthiness benchmarking, both deeply discussed in chapter 3. Basically, security qualification deals with the actual detectable security problems and results in a binary response, either a system under benchmarking is acceptable for use or it is unacceptable. The detectable security problems that a system may have, can actually be divided in two groups: 1) the system should not be obviously insecure, meaning that any severe vulnerability that opens the system to attacks renders it unacceptable; and 2) the lack of the mechanisms required by the domain for the security tasks (e.g. authentication for a withdraw operation in a bank system) also renders it unacceptable.

Trustworthiness benchmarking is the process of distinguishing the systems considered acceptable by security qualification. In this case, we examine the system under evaluation looking for evidences that show how good the design of the system is, therefore allowing to compare the probability of different systems having security problems. This is where we account for the uncertainty factors related with the security of the systems. The proposal and the extensive study of alternative approaches for defining useful trustworthiness benchmarks was actually the second major contribution of this thesis.

Chapters 4 and 5 were dedicated to the study of methodologies, approaches and actual implementations of the security benchmarking framework, with emphasis on trustworthiness benchmarking, for two fairly representative use cases, as discussed next.

Complex environments, where a diversity of people, hardware, software and configuration options interact towards a single goal. We studied this scenario in the form of a transactional system infrastructure. In this situation, as the possible configurations and circumstances are too many to account for, the most interesting usage of trustworthiness benchmarking is to help tracking the state of the system

and suggest ways for enhancing its security, basically by addressing the questions of what are the most important areas that should be improved and threats that should concern the administrators. The assessment of four real database infrastructures allowed demonstrating the capabilities of the benchmark. Several analysis and discussions about the security properties of the environments become evident, and such evidence can clearly be the justification required for systems modifications and even more drastic actions. Another lesson we obtained from the application of the benchmark was that its mere execution already provides a very significant amount of information to the administrator. One aspect demanded by the benchmark is the administrator to obtain information about the actual state of his infrastructure, which is something that not all administrators are able to do. The application of the benchmark also provided a very large amount of information to the administrators in the form of what were the configurations and the security mechanisms that they were neglecting or were not aware of, and what were the potential consequences of the existing configuration state.

A targeted well-bounded and controlled use-case where the goal is to select the most secure software implementation among several alternatives that implement the same specification. This scenario is the case of a typical business application working upon an already existing transactional system infrastructure. In this case, the threats can be more tightly specified and be much more detailed and precise. For this scenario, we first studied the design of an automated trustworthiness benchmark based on static code analysis tools. Using a series of experiments, we found that the metrics that can be designed based on such tools do really correlate with the security quality of the targets, and this was a very important result. This conclusion was particularly solid, because we evaluated the results against the evaluation of six different security experts, which manually reached out the same conclusions of our tool. However, we identified a series of limitations of the approach, namely that the dependence on tools that were not designed exactly for this goal would make the approach loose effectiveness in the future. The solution to this problem was to design and propose a general methodology to accomplish the exact same thing, but eliminating the problems that the static code analyzers had. This general approach was exercised and explained from the start to end, and was also partially validated in a small scale experiment that demonstrated that the approach is sound and may lead to effective long term solutions to the problem of trustworthiness benchmarking of web applications.

While chapters 4 and 5 focused on trustworthiness benchmarking approaches, in Chapter 6 we studied a very specific problem that arise from the combination of the two scenarios just described. The problem comes specifically from the fact that

such complex infrastructure does not conform easily to a security qualification specification. With several complementary systems and configurations, pinpointing the security characteristics that necessarily make the infrastructure unacceptable is not easy, as an administrator can always compensate single vulnerabilities or missing security mechanisms with additional overlapping defense systems. At the same time, that does not mean that any set of components within this infrastructure can be considered acceptable. Chapter 6 is specifically devoted to the development of a process that helps in analyzing the security mechanisms that a complex software package, like a DBMS engine plus an operating system, can provide to a complex infrastructure. Based on the trustworthiness benchmark proposed in Chapter 4, we built an assessment tool that can be used evaluate how much a particular software helps securing the infrastructure.

We evaluated the tool by actively applying it to seven representative software packages, which allowed finding several characteristics about the packages. Results show that there is a common set of security mechanism that is implemented by most packages, while several important mechanisms have no support at all on the packages analyzed. The reasons for this are open for debate, but we can conjecture that it has to do with a tradition of copying what has already being proposed in the field and has proven to work, without rethinking the whole features from scratch. When these systems are comprehensively analyzed, the missing features become highlighted. We believe that the analysis we did in this experiment is of utmost importance for database administrators and could be of great interest for vendors to improve the security characteristics of future software products and packages.

Future Work

This thesis is far from closing the problem of security benchmarking, and many future research topics can be envisaged, including:

- *Implementation of the framework for other domains*. This thesis was dedicated to the application of the framework specifically for transactional systems. One of the lessons of this application is that the two constituting parts of a transactional system cannot be trivially benchmarked simultaneously because each part has a different set of security goals. We believe that the study of the framework in the context of other domains would further improve our knowledge on how the different security goals of systems can affect the benchmark design.
- *More effective methods of developing and creating the components of a benchmark.* Most of the work required for the definition of the benchmarks demanded a lot of manual inspection and analysis, along with discussions

and inputs from security experts. Even though it is impossible to avoid completely the security knowledge needed for designing the benchmark from coming from security experts, the execution of several processes and definitions could be partially automated. Some examples:

  - The description of the security recommendations could be formalized, allowing for the automated analysis of the potential effects of the pessimistic scenarios whenever these recommendations were not being applied.
  - It is possible that the design of the security tests could be made automatically if the description of the practices was more formal. The automation of a partial set of security tests would already be an advancement of the usability of the test set.

- *Devising more effective ways for identifying the impact recommendations and security mechanisms.* Some of the proposed methods required the identification of the security impact of mechanisms and recommendations. We solved it by obtaining the consensual judgment of several distinct security experts and practitioners. However, opinions can always be biased, even for large samples of people. It would be extremely valuable to have more impartial and effective ways of determining the security impact of such elements.
- *Developing an automated tool capable of performing the benchmark proposed in Section 5.3*. Our expectation is that this particular tool would be much more efficient and precise than the benchmark based on static analysis tools. This would be a natural consequence of the fact that this new tool would be designed with the exact goal of performing trustworthiness benchmarking, while the static code analysis based benchmarking is taking advantage of a collateral effect - errors, something that should progressively disappear with their improvement.
- *Approaches to properly validate trustworthiness benchmarks*. As discussed in Section 4.5, validating the trustworthiness benchmark proposed in Chapter 4 is a extremely complex problem for which we do not have an easy solution. The main problem is that the most obvious metrics that could be used to confirm if the results of the benchmark are correct suffer from external effects that make them not suitable for comparison. In fact, the security incidents that could demonstrate if one threat vector is better protected than another one depend not only on the capabilities of the attackers, which are considered by the benchmark in the form of the security mechanisms in place, but also on the intentions of the

attackers, which by design are not considered in the benchmark because they are external variables. Therefore, we need to study methods for validating the benchmarks results without requiring the systems to be effectively attacked.

References

Alberts, C. and Dorofee, A. 2002. Managing Information Security Risks: The OCTAVE Approach. Boston, MA: Addison-Wesley.

Alberts, C., Dorofee, A., Stevens, J. and Woody, C. 2005. OCTAVE-S Implementation Guide, Version 1.0. Retrieved sept. 2012 from http://www.cert.org/octave/octaves.html

Alexander, I. 2003. Misuse Cases: Use Cases with Hostile Intent. IEEE Software, vol. 20, no. 1, pp. 58–66.

Allard, T., Anciaux, N., Bouganim, L., Guo, Y., Folgoc,L.L., Nguyen, B. Pucheral, P. Ray, I., Ray,I. and Yin, S. 2010. Secure personal data servers: a vision paper. Proc. VLDB Endow. 3, 1-2 (September 2010), 25-35.

Almeida, R. and Vieira, M. 2011. Benchmarking the resilience of self-adaptive software systems: perspectives and challenges, 6th International Symposium on Software Engineering for Adaptive and Self-managing Systems (SEAMS'11), Waikiki, Honolulu , HI, USA, Pages 190-195

Almeida, R. and Vieira, M. 2012a. Changeloads for Resilience Benchmarking of Self-Adaptive Systems: A Risk-Based Approach, 9th European Dependable Computing Conference (EDCC'12), Sibiu, Romania, Pages 173-184.

Almeida, R. and Vieira, M. 2012b. Changeloads: a Fundamental Piece on the SASO Systems Benchmarking Puzzle, 1st International Workshop on Evaluation for Self-Adaptive and Self-Organizing Systems (Eval4SASO), Lyon, France.

Almonaies, A.A., Alalfi, M.H., Cordy, J.R. and Dean, T.R. 2011. Towards a framework for migrating web applications to web services. In Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research (CASCON '11). IBM Corp., Riverton, NJ, USA, 229-241.

Amirtahmasebi, K., Jalalinia, S.R. & Khadem, S. 2009. A survey of SQL injection defense mechanisms. ICITST 2009. London, UK.

Antunes, J. and Neves, N. F. 2012. Recycling Test Cases to Detect Security Vulnerabilities, Proceedings of the 23nd Annual International Symposium on Software Reliability Engineering (ISSRE), Dallas, USA.

Antunes, J., Neves, N. F., Correia, M., Veríssimo, P. and Neves, R. 2010. Vulnerability Discovery with Attack Injection, IEEE Transactions on Software Engineering, Vol. 36, No. 3, pages 357-370, May/June 2010.

Antunes, N. and Vieira, M. 2010. Benchmarking Vulnerability Detection Tools for Web Services. ICWS 2010. Miami, USA.

Antunes, N. and Vieira, M. 2009. Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services, PRDC'09. China.

Arlat, J. and Crouzet, Y. .2002. Faultload Representativeness for Dependability Benchmarking, DSN 2002, Washington, DC, US.

Ayewah, N., Pugh, W. , Morgenthaler, J., Penix, J. and Zhou, Y. 2007. Evaluating static analysis defect warnings on production software. *ACM SIGPLAN-SIGSOFT 2007* . California, USA.

Bales, D. 2001. Java Programming with Oracle JDBC. O'Reilly Media; 1st edition.

Balzarotti, D., Cova, M., Felmetsger, V. V. and Vigna, G. 2007. Multi-module vulnerability analysis of web-based applications. In Proceedings of the 14th ACM conference on Computer and communications security (CCS '07). ACM, New York, NY, USA, 25-35.

Barbacci, M. et al. 2003. Quality Attribute Workshops (QAWs), Third Edition, CMU/SEI-2003-TR-016.

Barnum, S. 2007. An Introduction to Attack Patterns as a Software Assurance Knowledge Resource, OMG Software Assurance Workshop.

Baumhardt, F. 2006. Common Criteria - It Security Certification, Or Shiny Sales Sticker ?, (IN)SECURITY ARCHITECTURE. Last Access: Sept 2008. URL http://blogs.technet.com/fred/archive/2006/03/02/421014.aspx

Bellovin, S. and Bush, R. 2009. Configuration management and security. *Selected Areas in Communications, IEEE Journal on,* 27(3), 268-274.

Bertino, E., Jajodia, S. and Samarati, P. 1995. Database security: Research and practice. *Information Systems Journal*, Volume 20, Number 7.

Bishop, M. and Gates, C. 2008. Defining the Insider Threat. *Proceedings of the Cyber Security and Information Intelligence Research Workshop*, Oak Ridge, Tennessee, EUA.

Bondavalli, A. et al. 2009. D3.2: Final Research Roadmap, formal deliverable AMBER Project – Assessing, Measuring and Benchmarking Resilience, IST – 216295 AMBER, EU FP7 program.

Booch, G., Rumbaugh, J. and Jacobson, B. 2005. Unified Modeling Language User Guide, The (2nd Edition) (The Addison-Wesley Object Technology Series). Addison-Wesley Professional, May 2005.

Bowen, P., Hash, J. and Wilson, M. 2006. Information Security Handbook: A Guide for Managers, NIST Special Publication 800-100. National Institute of Standards and Technology, U.S. Dept of Commerce.

Caralli, R. A., Stevens, J. F., Young, L. R. and Wilson, W. R. 2007. The OCTAVE Allegro Guidebook, v1.0, Software Engineering Institute, Carnegie Mellon, May 2007, available at http://www.cert.org/octave/allegro.html, October 2010.

Cardellini, V., Casalicchio, E., Colajanni, M. and Yu, P.S.. 2002. The state of the art in locally distributed Web-server systems. ACM Comput. Surv. 34, 2 (June 2002), 263-311.

Castano, S., Fugini, M. G., Martella, G. and Samarati, P. 1994. *Database Security*. ACM Press Books, Addison-Wesley Professional.

Cenzic. 2009. Application security trends report Q3-Q4 2009. http://www.cenzic.com.

CGI Security. 2010. The Cross-Site Scripting (XSS) FAQ. http://www.cgisecurity.com/xss-faq.html.

Chapman, I., Sylvain, M., Leblanc, P. and Partington, A. 2011. Taxonomy of cyber attacks and simulation of their effects. In Proceedings of the 2011 Military Modeling & Simulation Symposium (MMS '11). Society for Computer Simulation International, San Diego, CA, USA, 73-80.

Chen, L., Feng, D., Shi; Z., Zhou; F. 2009. Using Session Identifiers as Authentication Tokens. Communications 2009. ICC '09. IEEE International Conference on , vol., no., pp.1-5, 14-18.

Chess, B and West, J. 2007. Secure Programming with Static Analysis. Addison-Wesley. ISBN 978-0-321-42477-8.

CIS Benchmarks. 2012. Center for Internet Security Configuratioin Benchmarks. Retrieved in sept. 2012 from https://benchmarks.cisecurity.org/en-us/?route=downloads.multiform.

CLUSIF. 2004. MEHARI (Information risk analysis and management methodology) V3, Concepts and Mechanisms.

Commission of the European Communities. 1993. *Information Technology Security Eval. Manual (ITSEM*).

Common Criteria. 1998. *Commercial Database Management System Protection Profile (C.DBMS PP)*, Issue 1.

Common Criteria. 1999. *Common Criteria for Information Technology Security Evaluation: User Guide*.

Common Criteria. 2000. *Database Management System Protection Profile (DBMS PP)*, Issue 2.1.

Computer Internet Security (CIS), 2008 “Benchmark/Tools”, www.cisecurity.org Last Access: Sept 2012

Cybenko, G., Kipp, L., Pointer, L. and Kuck, D. 1990. Supercomputer performance evaluation and the Perfect Benchmarks. SIGARCH Comput. Archit. News 18, 3b (June 1990), 254-266.

Da-sheng; W., Sheng-yu; W. 2010. Dynamically maintain the teaching examples of triggers and stored procedures about the course of database application. Education Technology and Computer (ICETC), 2010 2nd

International Conference on , vol.1, no., pp.V1-525-V1-527, 22-24 June 2010

Daswani, N. Kern, C. and Kesavan, A. 2007. Foundations of Security: What Every Programmer Needs to Know, Apress, Berkely, CA.

DBench. 2000. Dependability Benchmarking Project. http://spiderman-2.laas.fr/DBench/

Defense Information Systems Agency. 2001. *Database - Security Tech. Implem. Guide*, V8, R1.

Denning, P. J. 1976. Fault tolerant operating systems. ACM Computing Surveys (CSUR) 8 (4): 359–389. doi:10.1145/356678.356680. ISSN 0360-0300.

Department of Defense. 1985. *Trusted Computer System Evaluation Criteria*.

Dept. of Defense Standard. 1985. Department of Defense Trusted Computer System Evaluation Criteria, DOD 5200.28-STD.

Diallo, M. H., J. Romero-Mariona, et al. 2006. A Comparative Evaluation of Three Approaches to Specifying Security Requirements. REFSQ'06, Luxembourg

Dorfman, M. S. 2007. Introduction to Risk Management and Insurance (9th Edition). Englewood Cliffs, N.J: Prentice Hall.

Dunlop, A.N. 1994. The Status of Parkbench, In Proceedings of the 6th RAPS Workshop, CERFACS, Toulouse.

Eisenberg, A. 1996. New standard for stored procedures in SQL. SIGMOD Rec. 25, 4 (December 1996), 81-88.

FindBugs. 2011. Java static code analisys tool. Retrieved April 2011 from http://findbugs.sourceforge.net/

Fonseca, J. and Vieira, M. 2008a. Mapping Software Faults with Web Security Vulnerabilities. *IEEE/IFIP International Conf. on Dependable Systems and Networks (DSN 2008),* USA.

Fonseca, J. Vieira, M. and Madeira, H. 2008b. Online detection of malicious data access using DBMS auditing. Proceedings of the 2008 ACM Symposium on Applied Computing (SAC). Brazil.

Fonseca, J. Vieira, M. and Madeira, H. 2009. Vulnerability & attack injection for web applications. IEEE/IFIP International Conf. on Dependable Systems and Networks (DSN 2009). Portugal.

Fonseca, J., Vieira, M. and Madeira, H. 2007. Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks. 13th IEEE Pacific Rim Dependable Computing Conference (PRDC 2007), Melbourne, Victoria, Australia.

Forums Benchmarked. 2011 JavaBB. www.javabb.org, JForum. jforum.net, JGossip. jgossip.dev.java.net, JSForum. jsforum.sourceforge.net, mvnForum. mvnforum.com, Yazd forum. www.forumsoftware.ca

FP7 – 216295. 2010. AMBER - Assessing, Measuring, and Benchmarking Resilience. http://www.amber-project.eu

Friginal, J, de David, A., Ruiz, J-C. and Gil, P. 2009. Attack Injection for Performance and Dependability Assessment of Ad-Hoc Networks, 12th European Workshop on Dependable Computing, 2009, Toulouse (France).

Friginal, J, de David, A., Ruiz, J-C. and Gil, P. 2010. Attack Injection to Support the Evaluation of Ad Hoc Networks, 29th International Symposium on Reliable Distributed Systems (SRDS), New Delhi (India), Pages 21-29.

Friginal, J, de David, A., Ruiz, J-C. and Moraes, R. 2011. Using Dependability Benchmarking to Support ISO/IEC SQuaRE, 17th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC), Pasadena (USA), Pages 28-37.

Gefen, D. 2002. Reflections on the dimensions of trust and trustworthiness among online consumers. SIGMIS Database 33, 3 (August 2002), 38-53

Gegick, M. and Williams, L. 2005. Matching attack patterns to security vulnerabilities in software-intensive system designs. SIGSOFT Softw. Eng. Notes 30, 4 (Jul. 2005)

Gegick, M. and Williams, L. 2007. On the design of more secure software-intensive systems by use of attack patterns. Inf. Softw. Technol. 49, 4 (Apr. 2007), 381-397

Gencel, C. and Demirors, O. 2008. Functional size measurement revisited. ACM Trans. Softw. Eng. Methodol. 17, 3, Article 15 (June 2008), 36 pages.

Gray, J. 1993. Database and Transaction Processing Performance Handbook. *The Benchmark Handbook for Database and Transaction Systems* (2nd Edition), Morgan Kaufmann.

Gray, J. and Reuter A. 1992. Transaction Processing: Concepts and Techniques. Morgan Kaufmann. First edition.

Gray, Jim. 1993. “Database and Transaction Processing Performance Handbook.” The Benchmark Handbook for Database and Transaction Systems (2nd Edition), Morgan Kaufmann, 1993.

Harbitter, A and Menasc, D. 2002. A methodology for analyzing the performance of authentication protocols. ACM Trans. Inf. Syst. Secur. 5, 4 (November 2002), 458-491.

Helmer, G., Wong, G. 2007. Software fault tree and coloured Petri net based specification, design and implementation of agent-based intrusion detection systems, International Journal of Information and Computer Security, v.1 n.1/2, p.109-142.

Hevery, M. and Abrons, A. 2009. Declarative web-applications without server: demonstration of how a fully functional web-application can be built in an hour with only HTML, CSS & Javascript Library. In Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications (OOPSLA '09). ACM, New York, NY, USA, 801-802.

Hibernate. 2011. Hibernate persistence framework. Retrieved April 2011 from www.hibernate.org.

Hoffman, I. 2008. Rising Popularity of Web Application Development. Articlesbase. Retrieved Sept 2012 from http://www.articlesbase.com/software-articles/rising-popularity-of-web-application-development-512839.html

Hoglund, G. and McGraw, G. 2004. Exploiting Software: How to Break Code. Boston, MA: Addison-Wesley.

Honsa, J. D. and McIntyre, D.A. 2003. ISO 17025: Practical Benefits of Implementing a Quality System. Journal of AOAC International 86 (5): 1038–1044.

Howard, M. and LeBlanc, D. 2002. Writing Secure Code. Second Edition, Microsoft press.

Howard, M. and Leblanc, D. E. 2002. *Writing Secure Code*. 2nd. Microsoft Press.

IEC. 2012. International Electrotechnical Commission. www.iec.ch

Im, G. P. and Baskerville, R. 2005. A longitudinal study of information system threat categories: the enduring problem of human error. SIGMIS Database 36, 4 (October 2005), 68-79.

INFOSEC Research Council. 2005. *Hard Problem List*. Retrieved march, 2012 from http://www.cyber.st.dhs.gov/docs/IRC_Hard_Problem_List.pdf

Integrigy. 2007. An Introduction to SQL Injection Attacks for Oracle Developers. White paper. http://www.integrigy.com /security-resources/whitepapers/ Integrigy_Oracle_SQL_Injection_Attacks.pdf

IntelliJ IDEA. 2011. Retrieved April 2011 from http://www.jetbrains.com/idea

ISO. 2012. International Organization for Standardization. www.iso.org

Jackson, W. 2007. Under attack: Common Criteria has loads of critics, but is it getting a bum rap?. Government Computer News. Last Access: Sept. 2008. URL http://www.gcn.com/print/26_21/44857-1.html

Jahl, C. 1991. The information technology security evaluation criteria. In Proceedings of the 13th international conference on Software engineering (ICSE '91). IEEE Computer Society Press, Los Alamitos, CA, USA, 306-312.

Jansen, W. 2009. *Directions in Security Metrics Research.* NISTIR 7564. Retrieved march 2012 from http://csrc.nist.gov/publications/drafts/nistir-7564/Draft-NISTIR-7564.pdf

Jaquith, A. 2007. .Security Metrics: Replacing Fear,. Uncertainty, and Doubt. Addison Wesley.

Jelen, G. and Williams J. 1998. A Practical Approach to Measuring Assurance. *14th Annual Computer Security Applications Conference*, Phoenix, USA.

Johnston, R.G. 2010. Being Vulnerable to the Threat of Confusing Threats with Vulnerabilities. Journal of Physical Security. Volume 4, Issue 2.

Jovanovic, N., Kruegel, C. and Kirda, E. 2006. Precise alias analysis for static detection of web application vulnerabilities. Proceedings of the 2006 ACM SIGPLAN PLAS 2006, Ottawa, Ontario, Canada.

Kanoun, K. and Spainhower, L. 2008. Dependability Benchmarking for Computer Systems. Wiley-IEEE Computer Society Press.

Kanoun, K., Arlat, J., Costa, D. J.G. , DalCin, M. , Gil, P. Laprie, J.-C., Madeira H. and Suri, N. 2001. DBench (Dependability Benchmarking)", in Supplement of the Int. Conference on Dependable Systems and Networks (DSN-2001), (Göteborg, Sweden), DEPPY Workshop, pp. D.12-15, Chalmers University of Technology, Göteborg, Sweden.

Karabacak, B. and Sogukpinar, I. 2005. ISRAM: information security risk analysis method, Computers & Security 24 (2) 147-159.

Kaufman, C., Perlman, R. and Speciner, M. 2002. Network Security: Private Communication in a Public World (2nd Edition). Prentice Hall PTR

Kumaraguru, P. et al. 2007. Getting users to pay attention to anti-phishing education: evaluation of retention and transfer. In Proceedings of the anti-phishing working groups 2nd annual eCrime researchers summit (eCrime '07). ACM, New York, NY, USA, 70-81.

Littlewood, B. et al. 1993. Towards Operational Measures of Computer Security. *Journal of Computer Security*. v2. pp.211-229.

Littlewood, B., Popov, P., Strigini, L. and Shryane, N. 2010. Modeling the Effects of Combining Diverse Software Fault Detection Techniques. *IEEE Trans. Software Eng*. 26(12).

Liu, H. and Tan, H.B.K. 2006. An Approach to Aid the Understanding and Maintenance of Input Validation. Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on , vol., no., pp.370-379, 24-27

Livshits, V. and Lam, M. 2005. Finding security vulnerabilities in java applications with static analysis. *14th USENIX Security Symposium*, Baltimore, MD, USA.

Long, J. 2007. Google Hacking for Penetration Testers. Syngress. ISBN 978-1-59749-176-1

Lyu, M. 1996. Handbook of Software Reliability Engineering. *IEEE Comp. Society Press*, McGraw-Hill.

Manadhata, P. K, Tan, K. M. C., Maxion, R. A. and Wing, J. M. 2007. *An approach to Measuring a System's Attack Surface.* Carnegie Mellon University, Technical Report CMU-CS-07-146, August 2007.

Marsh, S., Dibben, M. .2005. Trust, Untrust, Distrust and Mistrust – An Exploration of the Dark(er) Side". iTrust 2005, Paris, France.

Martinez-Moyano, I. J., Rich, E., Conrad, S. H. , Andersen, D. F. 2006. Modeling the Emergence of Insider Threat Vulnerabilities. *Informs Winter Simulation Conference*, Monterey, CA.

Mate Bacic, E. 1990. The Canadian trusted computer product evaluation criteria. Computer Security Applications Conference. Proceedings of the Sixth Annual. pp.188-196, 3-7.

McClure, S. 2009. *Hacking Exposed: Network Security Secrets and Solutions*. McGraw-Hill. ISBN 978-0-07-161374-3

McDermott, J. 2000. Attack Net Penetration Testing. In The 2000 New Security Paradigms Workshop (Ballycotton, County Cork, Ireland, Sept. 2000), ACM SIGSAC, ACM Press, pp. 15-22.

McDermott, J. 2001. Abuse-Case-Based Assurance Arguments. In: Proc. 17 th Annual Computer Security Applications Conference (ACSAC'01), IEEE Computer Society Press.

McDermott, J., Fox, C. 1999. Using Abuse Case Models for Security Requirements Analysis. In: Proc. 15th Annual Computer Security Applications Conference (ACSAC'99), IEEE Computer Society Press.

McGraw, G. 2006. Software Security: Building Security In. Addison-Wesley Professional.

McKnight, D. H. and Chervany, N. L. 2006. The meanings of trust. TR, University of Minnesota, Carlson School of Management, 1996.

Mendes, N., Araújo Neto, A., Durães, J., Vieira, M. and Madeira, H. 2008. Assessing and Comparing Security of Web Servers. Proceedings of the Pacific Rim Dependable Computing Conference (PRDC 2008). Pages 313-322

Mendes, N., Durães, J. and Madeira, H. 2012. Benchmarking the Security of Web Serving Systems Based on Known Vulnerabilities. LADC 2011: 55-64.

Messmer, E. 2012. Black Hat: Oracle database vulnerabilities exposed again. Computer World UK Magazine. Retrieved in September 2012 in http://www.computerworlduk.com/news/security/3372534/black-hat-oracle-database-vulnerabilities-exposed-again/

Microsoft Corporation. 2011a. *Microsoft SQL Server 2005*. Retrieved august, 2011, from http://www.microsoft.com/sqlserver/en/us/default.aspx

Microsoft Corporation. 2011b. *Microsoft Windows XP*. Retrieved august, 2011, from http://windows.microsoft.com/en-US/windows/products/windows-xp

Monga, M. Paleari, R. and Passerini, E. 2009. A hybrid analysis framework for detecting web application vulnerabilities. 2009 ICSE SESS.

Morrison, M., Morrison, J. and Keys, A. 2002. Integrating web sites and databases. Commun. ACM 45, 9 (September 2002), 81-86.

Nadeem, M., Williams, B. J. and Allen, E. B. 2012. High false positive detection of security vulnerabilities: a case study. In Proceedings of the 50th Annual Southeast Regional Conference (ACM-SE '12). ACM, New York, NY, USA, 359-360.

National Cyber Security Division (NCSD), 2008. Common Attack Pattern Enumeration and Classification, http://capec.mitre.org/ Last Access: Sept 2008

Nikolić, I. 2009. Distinguisher and Related-Key Attack on the Full AES-256. CRYPTO 2009. Santa Barbara, California, USA.

Oliveira, R., Laranjeiro, N, and Vieira. M. 2011. A Composed Approach for Automatic Classification of Web Services Robustness. In Proceedings of the 2011 IEEE International Conference on Services Computing (SCC '11). IEEE Computer Society, Washington, DC, USA, 176-183

Open Web Application Security Project (OWASP). 2007. *OWASP top 10*. Retrieved august, 2012 from http://www.owasp.org/index.php/Top_10_2007

Oracle Corporation. 2011a. *MySQL Community Edition 5*. Retrieved august, 2011, from http://www.oracle.com/technetwork/database/express-edition/overview/index.html

Oracle Corporation. 2011b. *Oracle 10g Express Edition*. Retrieved august, 2011, from http://www.oracle.com/technetwork/database/express-edition/overview/index.html

OWASP. 2010. SQL Injection prevention Cheat Sheet, http://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet

Parker, D. B. 2002. Toward a New Framework for Information Security. In The Computer Security Handbook, 4th ed., New York, NY: John Wiley & Sons.

Patrick, A. S., Long, A. C., and Flinn, S. 2003. Human factors of security systems: A brief review.

Pauli, J. J. and Engebretson, P. H. 2008. Hierarchy-Driven Approach for Attack Patterns in Software Security Education. In Proceedings of the Fifth international Conference on information Technology: New Generations (April 07 - 09, 2008). ITNG. IEEE Computer Society, Washington, DC, 1156-1157

Pavlovic, D. 2011. Gaming security by obscurity. In Proceedings of the 2011 workshop on New security paradigms workshop (NSPW '11). ACM, New York, NY, USA, 125-140.

Payne, S. C. 2006. *A Guide to Security Metrics*. SANS Institute Information Security Reading Room.

Payton, A. M. 2006. Data security breach: seeking a prescription for adequate remedy. In Proceedings of the 3rd annual conference on Information security curriculum development (InfoSecCD '06). ACM, New York, NY, USA, 162-167.

Pernul, G. and Luef, G. 1992. Bibliography on database security. *ACM SIGMOD Rec.*, Volume 21, Issue 1.

PhD Thesis Complementary Info. 2012. Available at http://eden.dei.uc.pt/~mvieira/ThesisComplAfonso.zip

PostgreSQL Global Development Group. 2011. *PostgreSQL 8*. Retrieved august, 2011, from http://www.postgresql.org

Ram, P., Do, L. and Drew, P. 1999. Distributed transactions in practice. SIGMOD Rec. 28, 3 (September 1999), 49-55.

Ray, I. and Chakraborty, S. 2004. A Vector Model of Trust for Developing Trustworthy Systems. ESORICS 2004. France.

Red Hat. 2011. *Red Hat Enterprise Linux 5.* Retrieved august, 2011, from http://www.redhat.com/rhel/

Reuter, A. 2008. Is there life outside transactions?: writing the transaction processing book. SIGMOD Rec. 37, 2.

Roberts, N. H., Vesely, W.E., Haasl, D.F., and Golberg, F.F. 1981. Fault Tree Handbook, U.S. Nuclear Regulatory Comission, NUREG-0492. Washigton, D.C

Russell, D. and Gangemi, G.T. 1991. Computer Security Basics. O'Reilly Media. First edition.

Saad-Khorchef, F. Rollet, A. and Castanet, R. 2007. A framework and a tool for robustness testing of communicating software. In Proceedings of the 2007 ACM symposium on Applied computing (SAC '07). ACM, New York, NY, USA, 1461-1466.

Said, H. E., Guimaraes, M. A., Maamar, Z. and Jololian, L. 2009. Database and database application security. In Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education (ITiCSE '09). ACM, New York, NY, USA, 90-93.

Saitta, P., Larcom, B. and Eddington, M., 2008. Trike threat modelling tool", URL: http://www.octotrike.org,. Last Access: Sept 2008

Saitta, P., Larcom, B. and Eddington, M. 2005. Trike v.1 Methodology Document [draft], http://dymaxion.org/trike/ Last Access: Sept 2008

Sandia National Laboratories. 2010. *The Information Design Assurance Red Team*. Retrieved august 2010 from http://idart.sandia.gov

Sawyer, Tom. 1993. Doing Your Own Benchmark. The Benchmark Handbook for Database and Transaction Systems (2nd Edition), Morgan Kaufmann.

Schell, R. & Heckman, M. 1987. *Views for multilevel database security*. IEEE Trans. on Software Engineering.

Schell, R. and Heckman, M. 1987. *Views for multilevel database security*. IEEE Trans. on Software Engineering.

Schmidt, H. 2010. Threat- and Risk-Analysis During Early Security Requirements Engineering. Availability, Reliability, and Security. ARES '10 International Conference on, vol., no., pp.188-195, 15-18

Schneier, B., 1999. Attack Trees. Dr Dobbs Journal of Software Tools 24. URL: http://www.schneier.com/paper-attacktrees-ddj-ft.html Last access: Sept. 2008.

Schulte, W. 2012. Ten years of automated code analysis at Microsoft (invited industrial talk). In Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012). IEEE Press, Piscataway, NJ, USA, 1001-1001.

Schweitzer, D. 2006. Factory Settings -- Insecure by Default. COmputerWorld Magazine. Retrieved Sept 2012 from http://www.computerworld.com/s/article/110699/Factory_Settings_Insecure_by_Default

Seacord, R. 2006. Secure Coding in C and C++. Upper Saddle River, NJ: Addison-Wesley.

Shahriar, H and Zulkernine, M. 2012. Mitigating program security vulnerabilities: Approaches and challenges. ACM Comput. Surv. 44, 3, Article 11 (June 2012), 46 pages.

Shahzad, M., Shafiq, M. Z. and Liu, A. X. 2012. A large scale exploratory analysis of software vulnerability life cycles. In Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012). IEEE Press, Piscataway, NJ, USA, 771-781.

Sherriff, M. and Williams, L. 2006. Defect Density Estimation Through Verification and Validation. *The 6th Annual High Confidence Software and Systems Conference*, Lithicum Heights, MD, pp. 111-117.

Shoulman, A. 2009. *Top Ten Database Security Threats*. Imperva, white paper. Retrieved august 2010 from http://www.imperva.com/go/wp10/

SIEMENS. 2003. CRAMM - CCTA Risk Analysis and Management Method - User Guide version 5.0, Insight Consulting.

Siponen, M. T. and Oinas-Kukkonen, H. 2007. A review of information security issues and respective research contributions. SIGMIS Database 38, 1 (February 2007), 60-80

SPEC. 2012. Standard Performance Evaluation Corporation. Retrieved Sept 2012 from http://www.spec.org.

Stallings, W. 2010. Cryptography and Network Security: Principles and Practice. Prentice Hall. 5th Edition.

Stevens, J. 2005. Information Asset Profiling. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. Retrieved Sept 2012 from
http://www.sei.cmu.edu/publications/documents/05.reports/05tn021.html

Stevens, W. Myers, G. Constantine, L. 1974. Structured Design. IBM Systems Journal, 13 (2), 115-139.

Stoneburner, G., Goguen, A. and Feringa, A. 2002. Risk management guide for information technology systems. Last Access: Sept 2008 URL: http://csrc.nist.gov/publications/nistpubs/800-30/sp800-30.pdf.

Sullivan, K., Clarke J. and Mulcahy B. P. 2010. Trust-terms Ontology for Defining Security Requirements and Metrics. *4th European Conference on Software Architecture* (ECSA 2010). Copenhagen, Denmark.

Swiderski, F. and Snyder, W. 2004. Threat Modeling, Microsoft Press, Redmond, WA.

Toma, C.L. 2010. Perceptions of trustworthiness online: the role of visual and textual information. In Proceedings of the 2010 ACM conference on Computer supported cooperative work (CSCW '10). ACM, New York, NY, USA, 13-22.

Tomatis, N., Brega, R., Rivera, G., Siegwart, R. 2004. "May you have a strong (-typed) foundation" why strong-typed programming languages do matter. Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on , vol.4, no., pp. 3429- 3434 Vol.4

Torgerson, M. 2007. Security Metrics for Communication Systems. *12th International Command and Control Research and Technology Symposium*, Newport, Rhode Island.

Transaction Processing Performance Council. 2012. Retrieved Sept 2012 from http://www.tpc.org

Transaction Processing Performance Council. 2002. TPC Benchmark W, Standard Specification, Version 1.8, 2002, available at: http://www.tpc.org/tpcw/.

Transaction Processing Performance Council. 2005. TPC Benchmark C, Standard Specification, Version 5.4. available at: http://www.tpc.org/tpcc/.

Transaction Processing Performance Council. 2011. TPC Benchmark App, Standard Specification, Version 1.3, 2011, available at: http://www.tpc.org/tpc_app/.

van der Steen, A. J. 1989. Proposals for standard benchmark programs for supercomputers. In Proceedings of the Conference on CONPAR 88

(UMIST, Manchester, United Kingdom). C. R. Jesshop and K. D. Reimartz, Eds. Cambridge University Press, New York, NY, 621-634.

van der Steen, A. J. 1993. The benchmark of the EuroBen group. In Computer Benchmarks, J. J. Dongarra and W. Gentzsch, Eds. Elsevier Advances In Parallel Computing Series, vol. 8. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, 165-175.

Verendel,V. 2009. Quantified security is a weak hypothesis: a critical survey of results and assumptions. In *Proceedings of the 2009 workshop on New security paradigms workshop* (NSPW '09). ACM, New York, NY, USA, 37-50.

Vieira, M. 2005a. Dependability benchmark for Transactional Systems. PhD Thesis. University of Coimbra.

Vieira, M. and Madeira, H. 2003. A Dependability Benchmark for OLTP Application Environments. *29th International Conference on Very Large Data Bases*, VLDB2003, Berlin, Germany.

Vieira, M. and Madeira, H. 2005b . Towards a security benchmark for Database Management Systems. Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks, DSN2005, Yokohama, Japan.

Vieira, M. and Madeira, H. 2009. From Performance To Dependability Benchmarking: A Mandatory Path. Performance Evaluation and Benchmarking: Transaction Processing Performance Council Technology Conference (TPCTC).

Vraalsen, F., Mahler, T., Lund, M. S., Hogganvik, I., den Braber, F. and Stølen, K. 2007. Assessing Enterprise Risk Level: The CORAS Approach. In Advances in Enterprise Information Technology Security, D. Khadraoui and Francine Herrmann, Idea Group Reference.

Wang, X. et al .2005. Finding Collisions in the Full SHA-1, CRYPTO 2005. Santa Barbara, California, USA.

Weber-Jahnke, J. H. and Price, M. 2007. Engineering Medical Information Systems: Architecture, Data and Usability & Security. In Companion to the proceedings of the 29th International Conference on Software Engineering

(ICSE COMPANION '07). IEEE Computer Society, Washington, DC, USA, 188-189.

Whittaker, J. 2003. Why secure applications are difficult to write. In *Security & Privacy, IEEE* , vol.1, no.2, pp. 81-83.

Wool, A. 2004. A quantitative study of firewall configuration errors. Computer, vol. 37, pp. 62-67.

Yan, J., Blackwell, A., Anderson, R. and Grant, A. 2000. The Memorability and Security of Passwords -- Some Empirical Results. Tech. Report 500, Computer Lab, Cambridge.

Yang; J. 2011. A classification evaluation model for software trustworthiness based on trustworthiness evolution. Business Management and Electronic Information (BMEI), 2011 International Conference on , vol.1, no., pp.222-227.

Yasca. 2011. Retrieved april 2011 from http://www.scovetta.com/yasca.html

Zanero, S., Carettoni, L. and Zanchetta, M. 2005. Automatic Detection of Web Application Security Flaws, *Black Hat Briefings*.

Zhao, L., Sakr, S., Zhu, L., Xu, X. and Liu, A. 2012. An architecture framework for application-managed scaling of cloud-hosted relational databases. In Proceedings of the WICSA/ECSA 2012 Companion Volume (WICSA/ECSA '12). ACM, New York, NY, USA, 21-28.

Zsifkov, N. and Campeanu, R. 2004. Business rules domains and business rules modeling. In Proceedings of the 2004 international symposium on Information and communication technologies (ISICT '04).

Zwillinger, D. 1995. Standard Mathematical Tables and Formulae, Chapman&Hall/CRC. ISBN 0849324793.

Annex A

Security Recommendations Tests, Weights and Analytical Results

Table A.1 Security recommendations devised from the analysis of the CIS documents

#	SECURITY Recommendation (CIS)	Recommendations in CIS documents			
		M	O8	O10	S
	ENVIRONMENT				
1	Use a dedicated machine for the database	1	1	1	28
2	Avoid machines which also run critical network services (naming, authentication, etc)	1	1	1	1
3	Use Firewalls: on the machine and on the network border	1	3	3	1
4	Prevent physical access to the DBMS machine by unauthorized people				1
5	Remove from the network stack all unauthorized protocols		1	1	1
6	Create a specific user to run the DBMS daemons	1	1	1	
7	Restrict DBMS user access to everything he doesn't need	1	4	4	3
8	Prevent direct login on the DBMS user account	2	1	3	3
	INSTALLATION SETUP				
9	Create a partition for log information	2	1	1	1
10	Only the DBMS user should read/write in the log partition	1			
11	Create a partition for DB data	1	1	1	2
12	Only the DBMS user should read/write in the data partition	1			
13	Separate the DBMS software from the OS files	1	2	2	2
	Remove/Avoid default elements:				
14	»»»Remove example databases	1			1
15	»»»Change/remove user names/passwords	1	4	4	2

16	»»»Change remote identification names (SID, etc...)		3	1	
17	»»»Change TCP/UDP Ports		1	1	1
18	»»»Do not use default SSL certificates	1			
19	Separate production and development servers		1	1	
20	No developer should have access to the production server		5	5	
21	Use different network segments for production and development servers		1	1	1
	Verify all the installed DBMS application files:				
22	»»»Check and set the owner of the files	1	2	3	
23	»»»Set read/running permissions only to authorized users	4	18	22	14
	OPERATIONAL PROCEDURES				
24	Keep the DBMS software updated	3		1	1
25	Make regular backups	1			4
26	Test the backups	1		1	
	SYSTEM LEVEL CONFIGURATION				
27	Avoid random ports assignment for client connections (firewall configuration)		1	1	
28	Enforce remote communication encryption with strong algorithms	1	1	11	3
29	Use server side certificate if possible	1		1	
30	Use IPs instead of host names to configure access permissions (prevents DNS spoofing)		1	1	
31	Enforce strong user level authentication	2	6	8	4
32	Prevent idle connection hijacking		2	2	
33	Ensure no remote parameters are used in authentication	1	2	1	
34	Avoid host based authentication		1	1	
35	Enforce strong password policies	1	2		2
36	Apply excessive failed logins lock		1	1	
37	Apply password lifetime control		1	1	
38	Deny regular password reuse (force periodic change)		2	2	
39	Use strong encryption in password storage	3			
40	Enforce comprehensive logging	1	2	1	
41	Verify that the log data cannot be lost (replication is used)		2	2	1
42	Audit sensible information		14	19	25
43	Verify that the audit data cannot be lost (replication is used)		1		1
	Ensure no "side-channel" information leak (don't create/restrict access):				
44	»»»From configuration files		2	1	
45	»»»From system variables	1			
46	»»»From core_dump/trace files		8	8	1
47	»»»From backups of data and configuration files		1	1	4
	Avoid the interaction between the DBMS users and the OS:				
48	»»»Deny any read/write on file system from DBMS used	2	3	2	
49	»»»Deny any network operation (sending email, opening sockets, etc...)		4	3	
50	»»»Deny access to not needed extended libraries and functionalities	1	11	11	54
51	»»»Deny access to any OS information and commands	2			
	APPLICATION LEVEL CONFIGURATION AND USAGE				
52	Remove user rights over system tables	1	23	25	1
53	Remove user quotas over system areas		3	1	
54	Implement least privilege policy in rights assignments		9	10	6
55	Avoid ANY and ALL expressions in rights assignments	1	3	3	
56	Do not delegate rights assignments	1	3	3	3

57	No user should have rights to change system properties or configurations	3	4	4	2
58	Grant privileges to roles/groups instead of users		1	1	3
59	Do not maintain the DB schema creation SQL files in the DB server		1		
Total number of recomendations		**48**	**166**	**183**	**177**

Table A.2 Complementary DoD configuration best practices

#	COMPLEMENTARY BEST PRACTICES (DoD)	Group
1A	Monitor de DBMS application and configuration files for modifications	Operational Procedures
2A	Do not use self signed certificates	System Level Config.
3A	Protect/encrypt application code	Appl. Level Config./Usage
4A	Audit application code changes	Appl. Level Config./Usage
5A	Employ stored procedures and views instead of direct table access	Appl. Level Config./Usage

The following table presents the individual weights given by the experts, the relative importance to the attack surface and the cumulative importance for each best practice. For each contributor, E stands for engineer and A for academic.

Table A.3 Best Practices Weights

Best Practice	E1	E2	A3	A4	E5	A6	E7	A8	A9	Relative Weight	Cumul. Weight
4	4	4	4	4	4	4	4	4	4	5,26%	5,26%
3	4	4	4	4	4	4	4	3	4	4,73%	9,99%
19	4	4	4	3	4	4	4	4	3	4,21%	14,19%
28	3	4	4	3	4	4	4	4	4	4,21%	18,40%
57	3	4	4	3	4	4	4	4	4	4,21%	22,60%
2	3	4	3	3	4	4	4	4	4	3,68%	26,28%
24	3	3	4	4	3	4	4	4	4	3,68%	29,96%
39	4	3	4	3	3	4	4	4	4	3,68%	33,64%
35	4	3	4	2	3	4	4	4	4	3,63%	37,27%
15	4	3	4	4	3	3	3	4	4	3,15%	40,42%
1	3	4	3	2	4	4	4	3	4	3,10%	43,52%
6	2	4	4	2	4	4	4	2	3	3,00%	46,52%
52	2	3	4	3	3	4	3	4	4	2,58%	49,10%
25	4	4	3	3	1	4	4	3	2	2,52%	51,61%
20	3	4	3	3	4	3	4	3	3	2,10%	53,72%
23	3	3	4	3	3	3	4	3	4	2,10%	55,82%
18	3	3	3	2	3	3	4	4	4	2,05%	57,87%

31	4	4	3	2	4	3	3	3	3	2,05%	59,92%
8	2	3	2	3	3	4	4	3	4	2,00%	61,92%
29	2	4	3	2	4	3	4	3	3	2,00%	63,91%
51	2	4	3	2	4	3	3	3	4	2,00%	65,91%
32	3	4	2	1	4	3	3	4	3	1,99%	67,90%
36	3	3	3	2	3	3	4	3	4	1,52%	69,43%
54	3	3	4	3	3	2	3	4	3	1,52%	70,95%
33	4	3	3	2	3	4	3	2	3	1,47%	72,42%
37	3	2	3	1	2	3	4	3	4	1,41%	73,84%
10	2	3	3	1	3	4	4	3	1	1,41%	75,25%
12	2	3	3	1	3	4	4	3	1	1,41%	76,66%
42	2	2	3	2	2	4	4	3	3	1,37%	78,02%
41	3	1	1	1	1	4	4	2	2	1,24%	79,26%
22	3	3	4	2	3	3	3	3	3	1,00%	80,26%
34	3	3	4	2	3	3	3	3	3	1,00%	81,26%
5	3	3	2	2	3	3	4	3	3	0,95%	82,21%
48	2	3	4	2	3	3	3	3	3	0,95%	83,15%
21	3	3	2	3	3	3	4	1	3	0,94%	84,09%
47	2	2	4	3	2	3	3	3	3	0,89%	84,99%
38	3	2	3	1	2	3	4	3	3	0,89%	85,88%
55	3	3	4	1	3	1	3	3	2	0,88%	86,76%
46	2	2	4	3	2	3	3	2	3	0,84%	87,60%
50	2	2	4	2	2	3	3	3	3	0,84%	88,44%
7	2	2	3	2	2	3	4	2	3	0,79%	89,23%
44	2	2	2	3	2	4	3	2	3	0,79%	90,02%
45	2	2	2	3	2	4	3	2	3	0,79%	90,81%
49	2	2	4	2	2	3	3	2	3	0,79%	91,59%
26	3	3	2	2	1	2	4	2	3	0,78%	92,38%
40	4	1	1	2	1	3	3	3	2	0,77%	93,15%
43	2	2	3	1	2	3	4	2	2	0,73%	93,88%
9	3	1	1	2	2	3	4	2	1	0,72%	94,60%
4A	1	1	4	1	1	3	3	2	2	0,71%	95,32%
11	2	1	1	2	2	3	4	2	1	0,67%	95,98%
17	2	1	2	1	1	2	4	2	2	0,62%	96,60%
13	1	1	1	1	1	2	4	1	2	0,60%	97,20%
56	3	3	3	2	3	3	3	3	3	0,47%	97,67%
30	2	3	2	1	3	3	3	3	2	0,31%	97,98%
1A	2	3	2	2	3	2	3	3	2	0,26%	98,24%
53	2	2	3	2	2	1	3	3	3	0,26%	98,50%

58	3	2	1	3	2	2	3	2	3	0,26%	98,76%
27	2	3	1	1	1	3	3	1	3	0,24%	99,00%
2A	2	2	3	1	2	1	3	3	2	0,20%	99,20%
14	1	1	2	3	1	3	3	2	1	0,19%	99,39%
5A	2	2	2	3	2	2	3	2	2	0,16%	99,55%
16	2	2	2	1	2	3	3	2	2	0,15%	99,70%
59	2	2	1	2	2	3	3	2	2	0,15%	99,85%
3A	3	2	2	1	2	2	3	1	2	0,15%	100,00%

Table A.4 Complete list of tests.

#	TEST	Fail
	ENVIRONMENT	
1	If the machine is turned off, does any service other than the database become unavailable? Is there any process running on the machine which is not demanded by the DBMS, the OS or the machine maintenance/security?	Yes
2	If the machine is turned off, does any critical network service, like naming, directory or authentication services, becomes unavailable?	Yes
3	Is there a firewall on the network border? Is there a firewall running on the DBMS machine? Are both firewalls properly configured by experienced staff with solid network knowledge? [9, 14, 16]	No
4	Is it possible to an unauthorized person to physically access the machine without supervision at any given time?	Yes
5	List the protocols available in the network stack in the OS of the DBMS machine. For each protocol, is there a clear justification for its availability?	No
6	List the DBMS processes in the OS. For each process, is the user running it used to run any other process at any time?	Yes
7	Locate the DBMS processes user. Does that user have administration rights? Does it can run applications not DB related? Does it have read rights on any file not necessary to the DBMS processes?	Yes
8	Locate the DBMS processes user. Can you login in the OS with it? (assume you know its password)	Yes
	INSTALLATION SETUP	

9	Locate the log files of the DBMS and identify their file system partition. Are there any other files in this partition besides the logs?	Yes
10	Locate the log files of the DBMS and identify their file system partition. Does that partition have exclusive read/write rights for the DBMS user?	No
11	Locate the data files of the DBMS and identify their file system partition. Are there any other files in this partition besides the data files?	Yes
12	Locate the data files of the DBMS and identify their file system partition. Does that partition have exclusive read/write rights for the DBMS user?	No
13	List all OS users which work only with the DB. List all OS regular users (not DB users). List all DBMS applications and OS applications that are necessary for the OS users that work with the DB. Does any regular user can access any DBMS application listed? Does any DB user can access any application not in one of the lists?	Yes
14	List all DBMS databases. Install a fresh copy of the DBMS in a test machine without any customization and then list its DBMS databases. Is there any database in both lists which isn't required for the DBMS?	Yes
15	List all DBMS accounts. Install a fresh copy of the DBMS in a temporary machine without any customization and then list its DBMS accounts. Is there any account in both lists?	Yes
16	List any identification names a remote user must know to connect to the DBMS. Install a fresh copy of the DBMS in a temporary machine without any customization and then list the identification names a remote user must know to connect to this DBMS instance. Is there any name in both lists?	Yes
17	List any TCP/UDP ports a remote user must know to connect to the DBMS. Install a fresh copy of the DBMS in a temporary machine without any customization and then list the TCP/UDP ports a remote user must know to connect to this DBMS instance. Is there any port in both lists?	Yes
18	List all SSL certificates used with the DBMS. For each one, was it created by experienced staff with that specific purpose? [2, 4]	No
19	Is there any kind of development or testing being done in the production server?	Yes
20	Does any developer have a valid DBMS account or OS account in the production server?	Yes

21	List the sub-net mask of the IP address of the production and the development servers. Are they the same? Are both servers reachable from one other through a path with only layer 2 network equipments (hubs, switches, etc…)?	Yes
22	List all files installed with the DBMS application. For each file, is its owner correctly set as the DBMS user?	No
23	List all files installed with the DBMS application. For each file, are its rights correctly configured according to its purposes?	No
	OPERATIONAL PROCEDURES	
24	Check your DBMS version. Check the latest DBMS version available from the vendor which is an update to your version. Are they different? Is there any recommendation from the vendor against the use of your version?	Yes
25	Is a carefully thought out, documented backup procedure regularly executed? If the person in charge suddenly quit, is it easy for anyone else to resume its task?	No
26	Is the backup data regularly tested after it is generated? Is a recovery procedure regularly fully simulated? Is the backup data stored in a secure place other than the DB server?	No
1A	Is there any procedure (like checking the files hashes) employed to regularly identify if any of the DBMS application files or configuration files have been change by someone unauthorized?	No
	SYSTEM LEVEL CONFIGURATION	
27	During a connection procedure, does the server assign a full range random local port for the remote user to connect?	Yes
28	Establish a connection from any remote user to the server, capture the underlying network traffic and ask for a security expert to analyze it. Is the connection being secured with a recognized encryption protocol like TLS?	No
29	Does the user connection require the knowledge of a server certificate?	No
30	List all configuration files/parameters of the DBMS. Is a host name used on any parameter?	Yes
31	For each registered DBMS user, was it created for a specific application /purpose/person? Is the authentication procedure used in the applications recognizably secure? Does it use a standard algorithm or protocol? [13, 14]	No

32	Establish a connection with the DBMS and let it stay idle. Is the connection severed in a reasonable amount of time?	No
33	Is any specific information other than a username and password obtained from the client host during the authentication procedure?	Yes
34	List all authentication methods used with the DBMS. For each one, does it depend only on the host?	Yes
35	Was a clear policy defined (and documented) about how passwords would be changed, when they must be changed, how they should be retrieved if lost and what rules they must obey? Does it comply with standard recommendations from security experts? [13, 17]	No
36	Try authenticating several times with a wrong password. Is there a try when the account becomes permanently locked?	No
37	Advance the server clock an unreasonable number of months. Authenticate to the server. Are you forced or recommended to change the password?	No
38	Try changing your password to the same password. Did you succeed?	Yes
39	Locate the table or file where the passwords are stored and ask for a security expert to analyze it. Are the passwords stored as some recognizably standard hash algorithm? [13, 14]	No
40	Is logging turned on? Is the log level set to report at least database errors and client connections? Is there a clearly justified reason for it not to be set to a higher level?	No
41	Are the logs periodically checked? Are the logs also included in the backup procedures? Is the space of the partition where the logs are written monitored?	No
42	Are the following operations traceable: creation and destruction of users, objects and sessions, failed and successful logins, rights assignments and data changes on critical tables?	No
43	Is the trace data stored in a different area than the database? Does that area have its read/rights permissions correctly set? Is the space of the partition where it is stored monitored?	No
44	For each configuration file, analyze its permissions. Is it readable only by authorized users?	No
45	For each system variable, does it contain sensitive information (any which should be private) and can be seen by all OS users?	Yes
46	Are core_dump or trace files being generated for failed processes and are they generally visible in the OS?	Yes

47	Does the editor used to update configuration files generate backups of the edited files and do they remain available for reading afterwards?	Yes
48	For each function and extended functionality available, does it allow a user to access a file on the file system?	Yes
49	For each function and extended functionality available, does it allow a user to do any kind of network operation?	Yes
50	For each function and extended functionality available, is its availability clearly required? Is it impossible to do the same task without it?	No
51	For each function and extended functionality available, does it allow a user to gather any info about the OS? Does it allow a user to run any OS command?	Yes
2A	For each certificate used in the servers, is it bought from a trusted company, which has root certificate already installed in the most common browsers and operating systems?	No
APPLICATION LEVEL CONFIGURATION AND USAGE		
52	Make a list of all system tables (not created for use with applications). For each one, check if there is any user with some permission (read or write) over it. Are those permissions clearly justified and necessary?	No
53	Make a list of all system databases. For each element on the list, check if there is any user with some permission over it. Is this permission clearly justified and necessary?	No
54	For each non-DBA user, list all its permissions. For each permission, does it have a clear justification? Is it impossible for the user to work without it?	No
55	For each non-DBA user, list all its permissions. For each permission, is it of type ANY or ALL, which would automatically propagate to other objects of the same type?	Yes
56	For each non-DBA user, list all its permissions. For each permission, does it allow that user to grant it to another user?	Yes
57	For each non-DBA user, list all its permissions. For each permission, does it allow that user to change some system configuration which is either critical or valid to the whole DB?	Yes
58	For each non-DBA user, list all its permissions. For each permission, does the user inherit it from a group or role he is assigned to?	No

59	List all documents and files that contain any schema information. For each one, is it stored in the DB server?	Yes
3A	Is the production application code being stored in a trusted repository (like a Concurrent Versioning System), with proper authentication, or being closely controlled and checked against malicious modification (e.g. encrypted)?	No
4A	Is it possible to identify unequivocally, at all times, for all application code, who made each modification and programming?	No
5A	Are all data modification operations being applied through carefully programmed stored procedures instead of direct updates? When reading data from critical tables, are the unnecessary data fields being filtered through views or other means?	No

In the following table, P stands for test passed, F for test failed and U for unknown (which is treated as failed test).

Table A.5 Analytical results of the infrastructures evaluated

Test Number	Case 1	Case 2	Case 3	Case 4
1	P	P	F	F
2	P	P	P	P
3	P	F	P	F
4	P	P	F	P
5	P	F	F	F
6	F	F	F	F
7	F	F	F	F
8	P	P	P	P
9	F	F	P	F
10	F	F	F	F
11	P	F	P	F
12	F	F	F	F
13	F	F	F	F
14	F	P	P	P
15	F	P	P	P
16	F	P	P	P
17	F	F	P	F

18	P	P	F	P
19	F	F	F	F
20	P	F	P	F
21	F	P	F	F
22	P	U	F	F
23	F	F	F	F
24	F	P	P	F
25	P	P	F	P
26	F	F	F	F
1A	F	F	F	F
27	P	U	U	U
28	F	F	F	F
29	F	F	F	F
30	P	P	P	P
31	P	F	P	F
32	F	F	F	F
33	P	P	P	P
34	P	P	P	P
35	P	F	F	F
36	P	F	F	F
37	F	F	F	F
38	F	F	F	F
39	P	P	P	P
40	P	F	P	F
41	P	F	P	F
42	F	F	F	F
43	F	F	F	F
44	P	P	F	F
45	P	P	P	P
46	F	P	P	P
47	P	P	P	P
48	P	P	P	P
49	P	P	F	P
50	U	P	F	F
51	U	P	F	U
2A	P	F	P	F
52	P	F	P	F
53	P	F	P	P
54	P	F	F	P

55	P	P	F	P
56	P	P	F	P
57	P	P	P	P
58	F	F	F	F
59	P	F	P	P
3A	F	F	F	F
4A	F	F	F	F
5A	F	F	F	F

Annex B

Pessimistic Scenarios

Table B.1 Complete list of pessimistic scenarios

Recommendations	Pessimistic Scenarios
Use a dedicated platform for the database	The DBMS platform hosts other applications which may have security vulnerabilities
Avoid platforms which also run critical network services (naming, authentication, etc)	The DBMS platform hosts a directory, naming or similar high critical network service
Install and properly configure a firewall on the network border	The network does not have a border firewall, leaving all network fully accessible to internet traffic
Install and properly configure a firewall on the host OS	The OS does not have a local firewall leaving any listening process fully accessible to the local area network
Prevent physical access to the DBMS platform by unauthorized people	The platform is physically stationed in a place where non-authorized personnel have regular access
Remove from the network stack all unauthorized protocols	The OS has several network protocols installed which are non-essential and which characteristics and consequences are not fully understood
Create a specific user to run the DBMS daemons	The OS *userid* used to run the DBMS daemons are used for other daemons and tasks as well
Restrict DBMS user access to everything he doesn't need	The OS *userid* used to run the DBMS daemons has privileges over non-necessary OS parts (configuration files, for instance)
Prevent direct login on the DBMS user account	It is possible to try to login in the OS using the *userid* of the DBMS daemon
Create a partition for log/auditing information	The log/auditing information is placed in the same partition as the OS
Only the DBMS user should read/write in the log/auditing partition	Any OS *userid* can read/write in the log/auditing information

Create a partition for DB data	The data files are hosted in the same partition as the OS
Only the DBMS user should read/write in the data partition	Any OS *userid* can read/write the DBMS data files
Remove example databases	Any potential attacker know the innerworkings and exact details of at least one database within the DBMS
Change/remove default user names	Any potential attacker knows at least one DBMS *userid* that can be used to login in the database
Change default passwords	Any potential attacker knows at least one *userid*/password pair that can be used to login in the database
Change default remote identification names (SID, etc...)	Any potential attacker knows the remote identification names used by the database
Change default TCP/UDP Ports	Any potential attacker knows exactly to what ports the DBMS process is listening
Do not use default SSL certificates	All attackers have access to the private key of the certificate in use
Separate production and development servers	Developers run untested/developmental code over real live production data
No developer should have access to the production server	Developers have partial or total control and access over the production data
Use different network segments for production and development servers	The developers work and access the server though the same local network segment where the production server is hosted
Check and set the owner of all the DBMS files	One or more OS users are owner of the DBMS files
Set read/write/running permissions of the DBMS files to authorized users	All OS users have read/write/running permissions over all DBMS files
Keep the OS software updated	The OS has known vulnerabilities which are not patched with vendor updates
Keep the DBMS software updated	The DBMS has known vulnerabilities which are not patched with vendor updates
Make regular backups	There is no updated copy of the production data in a separate storage
Test the backups	The backup files might be corrupted or being incorrectly generated
Monitor de DBMS application and configuration files for modifications	It is impossible to know if the configuration files or DBMS application files have been tampered with
Avoid random ports assignment for client connections	DBMS configuration makes it impossible to configure the external firewall as to not accept external connection requests to a large range of unspecified ports
Enforce remote communication encryption with strong algorithms	Remote clients of the database use exchange data in clear
Use server side certificate	There is no reliable way for a remote client to be sure he is connection to the correct server instead of a “rogue” one
Use IPs instead of host names to configure access permissions	The server automatically accepts connections from computers identified by a particular DNS

Enforce strong user level authentication	The authentication mechanism used is not well understood, may be flawed and does not pinpoint the specific person that is connected
Prevent idle connection hijacking	Connections to the server are never terminated automatically
Ensure no remote parameters are used in authentication	It is possible to test the reaction of the system to an additional parameter during authentication
Avoid host based authentication	The server automatically accepts connections from specific hosts which are not complete under control of the administrator
Enforce strong password policies	DBMS users may choose any password they like, no specific rules are enforced
Apply excessive failed logins lock	Anyone may try to login in the DBMS any number of times
Apply password lifetime control with forced change	Users and applications may use the same password indefinitely
Use strong encryption in password storage	Stored password information in the database is cleartext
Enforce comprehensive logging	Nothing done in the system and DBMS is recorded anywhere
Verify that the log data cannot be lost or tampered with	The log data is unprotected, unreplicated and may be susceptive to unidentified modifications
Audit sensitive information	No operation done over the data within the database is recorded anywhere
Verify that the audit data cannot be lost or be tampered with	The audit data unprotected, unreplicated and may be susceptive to unidentified modifications
Ensure no “side-channel” information leak through configuration files	Configuration files are generally visible and contain sensitive information like passwords
Ensure no “side-channel” information leak through system variables	OS system variables (like the processes list) contain sensitive information like passwords
Ensure no “side-channel” information leak through core_dump/trace files	Core_dumps and trace files from sensitive processes are created and kept scattered within the file system
Ensure no “side-channel” information leak through backups of data and configuration files	Backups of data and configuration files are kept in a location generally visible and unmonitored
Deny any read/write on file system from DBMS used	Applications regularly create, read and manipulate local files though DBMS commands
Deny any network operation (sending email, opening sockets, etc...)	Applications regularly access the network through DBMS commands
Deny access to not needed DBMS extended libraries and functionalities	It is not known what extended functionalities are available
»»»Deny access to any OS information and commands	Applications regularly executes OS commands
Do not use self signed certificates	Any attacker can create another server certificate with the exact same information as the one in use
Remove users privileges over system tables	DBMS users have knowledge and access to internal control information, and may alter the DBMS engine behaviour
Remove user quotas over system areas	DBMS users have the possibility of writing new objects in a system area

Implement least privilege policy in privileges assignments	DBMS users may read and alter critical data which they should have access to
Avoid ANY and ALL expressions in privileges assignments	DBMS users may read and alter critical data which they should have access to, and may create and modify database elements
Do not delegate privileges assignments	DBMS users can transfer its own privileges to other untrusted users
No user should have privileges to change system properties or configurations	DBMS users can alter or influence the DBMS environment and behaviour
Grant privileges to roles/groups instead of users	DBMS users have specific unknown privileges which are not reflected as privileges of any defined role
Do not maintain the DB schema creation SQL files in the DB server	OS users have complete information about the database internal structure
Protect/encrypt application code	Application code may be altered by unknown individuals under certain uncontrolled circunstances
Audit application code changes	It is generally not possible to know which individual made which modifications to application code
Employ stored procedures and views instead of direct table access	DBMS users may read and alter critical data which they should not have access to