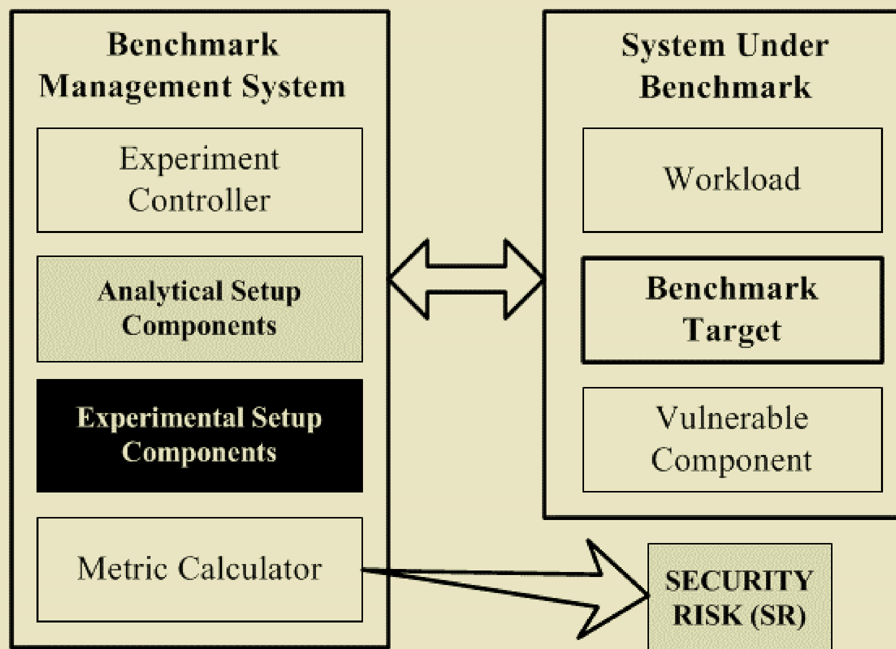

Naaliel Vicente Mendes

SECURITY BENCHMARKS FOR WEB SERVING SYSTEMS

PhD Thesis of the Doctoral Program of Science and Information Technology supervised by Professors Henrique Santos do Carmo Madeira and João António Pereira Almeida Durães and presented at the Department of Informatics and Engineering of the Faculty of Sciences and Technology of the University of Coimbra

September, 2015

Universidade de Coimbra

Department of Informatics Engineering

Faculty of Sciences and Technology

University of Coimbra

SECURITY BENCHMARKS FOR WEB SERVING SYSTEMS

Naaliel Vicente Mendes

PhD Thesis of the Doctoral Program of Science and Information Technology supervised by Professors Henrique Santos do Carmo Madeira and João Antonio Pereira Almeida Durães and presented at the Department of Informatics and Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

September 2015

FOREWORD

This research was developed at the Software and Systems Engineering Group of the Center for Informatics and Systems of the University of Coimbra (CISUC). This work was partially funded by the Portuguese Research Agency, *Fundação para a Ciência e a Tecnologia* (FCT), through the scholarship SFRH / BD / 61969 / 2009, and by the Portuguese Government/European Union through the R&D Unit 326/94 CISUC.

This work was supervised by Professors Doctor Henrique Santos do Carmo Madeira, Full Professor of the Department of Informatics and Engineering of the Faculty of Sciences and Technology of the University of Coimbra, and Doctor João Antonio Pereira Almeida Durães, Adjunct Professor of the Polytechnic Institute of Coimbra.

~ To my beloved wife and children ~

ABSTRACT

The assessment of the security level of computer systems in a standardized and regular manner (*security benchmarking*) has become a very relevant subject, especially for those who use computer systems to support critical business missions or to store confidential information. The concern about computer-based system security is totally justified: systems have become increasingly complex, interconnected, and pervasive, and their security have been threatened by many types of attacks. These attacks are unavoidable, as the root causes for them are tied up to human aspects that cannot be removed (intention to cause harm, intention to steal information, etc.), and the losses attacks can cause to their targets (when successful) can be very significant. This scenario of attack inevitability has led companies and governments to invest massively in the development of regulations and mechanisms aimed at the improvement of the security of computer systems (e.g., training developer teams, rapidly solving discovered vulnerabilities, using tools to detect and prevent attacks). Despite these efforts, successful attacks continue to happen, showing that computer systems remain insecure. This is why end-users, system administrators, and systems integrators (to mention just a few classes of users) consider security as an important decision factor when choosing which system to buy and use. These individuals are looking for the means to assess and compare the security of functionally-similar systems/components that will enable them to make a decision taking into account the assessment of security risk.

This thesis presents a novel, reproducible, risk-based methodology to benchmark the security of software-based systems. This is a generic methodology that can be instantiated to any class of software-based system. Our benchmark methodology uses the notion of risk in a quantifiable way to measure the security of systems, with **a single security metric (*SBench*)** to simplify the comparison of different systems (or different configurations of the same system), enabling users and system integrators to identify and select the most secure one, allowing as well the breakdown of this single metric for more detailed analysis. Our methodology follows the approach of benchmarks proposed in the field of performance and dependability, containing elements such as metrics, workload, and experimental setup, and defining a comprehensive set of procedures and rules to ensure the compliance with key properties such as repeatability.

Our security benchmark methodology cover the two complementary views of a

given system concerning security: the first takes into account concrete vulnerabilities effectively existing for that system (measures what is already known), and the second estimates the effects of possible yet-to-discover vulnerabilities (and, in fact, many attacks are based on previously unknown vulnerabilities). In fact, these views correspond to the two parts of our benchmark methodology: the static and the dynamic. The static part corresponds to a static analysis of the target system and uses the knowledge about the impact and exploitability of known vulnerabilities discovered for that component or system. The dynamic part corresponds to an experimental analysis of the system in runtime operation when subjected to attacks, while observing the behavior of the system in the presence of these attacks. The combination of the results of these two parts forms the security benchmark measure that enables users, administrators, integrators, and security specialists to identify the most secure among functionally equivalent software systems.

This thesis also exemplifies how to apply our security benchmark methodology for a particular and widely used system class (web serving systems), also describing the tools implemented to speed up the execution of the security benchmark. Due to their role in society and exposure to public at general, web serving systems are constantly targeted by attacks, making the implementation of a security benchmark for web serving system a very pertinent contribution.

This thesis presents case studies that demonstrate the feasibility, the usefulness and the validity of our security benchmark. Following our methodology, end-users will be able to estimate the security risk of given systems and, if needed, use the results to select the most secure one. The fact that our security benchmark methodology is designed to address any class of software-based systems, uses the notion of risk in the benchmark metric, applies an experimental approach to stress the security of systems, and provides procedures and rules that can guide the further development of representative security benchmark standards, make us sure that this is an effective and important contribution to both the industry and the academia.

Keywords:

Security; metrics; benchmarking; risk; security risk; software systems; web serving systems

RESUMO

A avaliação padronizada de segurança de sistemas computacionais (*benchmarking* de segurança) é um assunto bastante relevante especialmente para aqueles que usam sistemas computacionais para suportar negócios, missões críticas e armazenar informações confidenciais. A preocupação quanto a segurança de sistemas computacionais é totalmente justificada: sistemas computacionais têm-se tornado mais complexos, interconectados e ubíquos e a segurança destes sistemas tem sido ameaçada por diferentes tipos de ataques. Os ataques são inevitáveis, uma vez que suas causas estão ligadas a aspectos humanos que não podem ser eliminados (intenção de causar dano, intenção de roubar informações, etc.) e os danos que podem causar nos sistemas (quando bem sucedidos) podem ser muito significantes. O facto dos ataques serem inevitáveis tem feito com que empresas e governos invistam massivamente no desenvolvimento de regulamentos e mecanismos para melhorar a segurança de sistemas computacionais (e.g., treinamento de equipas de programadores de computador, resolução rápida de vulnerabilidades recém-descobertas, uso de ferramentas para detetar e prever ataques). Apesar destes esforços, ataques bem sucedidos continuam a acontecer, mostrando que os sistemas computacionais permanecem inseguros. Este é o motivo pelo qual utilizadores, administradores e integradores de sistemas (para mencionar apenas algumas classes de utilizadores) consideram segurança como um importante fator de decisão ao escolher qual sistema comprar e usar. Estes indivíduos estão a procura de meios que lhes permitam escolher sistemas baseado na avaliação do risco de segurança.

Esta tese apresenta uma metodologia inovadora, reproduzível e baseada na noção de risco para medir e comparar a segurança de sistemas computacionais (*benchmarking* de segurança). Esta metodologia é genérica, podendo ser aplicada em qualquer classe de sistema. Nossa metodologia de *benchmarking* de segurança usa uma abordagem quantitativa de risco para medir segurança, **com uma métrica única de segurança (*SBench*)** que simplifica a comparação de sistemas (ou diferentes configurações do mesmo sistema), ajudando utilizadores e integradores a identificar e escolher o sistema mais seguro, bem como permitindo o desdobrar desta métrica em indicadores que permitam análises mais detalhadas. Nossa metodologia segue a abordagem de

benchmarks propostas no campo do desempenho e da fiabilidade de sistemas, incluindo elementos como medidas, carga de trabalho e *setup* experimental, e definindo um conjunto detalhado de procedimentos e regras com o objetivo de tornar a metodologia repetível, uma importante propriedade de uma *benchmark*.

A nossa metodologia de benchmarking de segurança cobre as duas visões, que são complementares, de segurança de sistemas: a primeira leva em consideração as vulnerabilidades que efetivamente existem no sistema (mede o que já é conhecido), enquanto a segunda estima os efeitos de vulnerabilidades ainda a descobrir (de facto, muitos ataques são construídos sobre vulnerabilidades previamente desconhecidas). Na verdade, estas duas visões correspondem às duas partes da nossa metodologia: uma estática e uma dinâmica. A parte estática corresponde a uma análise estática do sistema alvo e usa informação do impacto e explorabilidade das vulnerabilidades que já foram descobertas naquele componente ou sistema. A parte dinâmica corresponde a uma análise experimental do sistema em tempo de execução, sujeitando-o a ataques e observando-o do ponto de vista de segurança. A combinação dos resultados destas duas partes forma a medida da *benchmark* de segurança que permite aos utilizadores, administradores, integradores e especialistas de segurança identificar o sistema mais seguro dentre aqueles que executam funções equivalentes.

Esta tese também exemplifica como aplicar a nossa metodologia de benchmark de segurança numa classe de sistemas utilizada amplamente (os *web serving systems*), descrevendo também as ferramentas implementadas para acelerar a execução da benchmark de segurança. Devido à sua função na sociedade e exposição ao público em geral, *web serving systems* estão constantemente sob ataques, o que faz da implementação de uma benchmark de segurança para *web serving systems* uma contribuição muito pertinente.

Os casos de estudo apresentados demonstram a viabilidade, a utilidade e validade da nossa metodologia de *benchmark* de segurança. Ao seguir nossa metodologia, utilizadores poderão estimar o risco de segurança de sistemas e, se necessário, utilizar os resultados para escolher o sistema mais seguro. O facto da nossa metodologia ser projetada para qualquer classe de sistema, usar a noção do risco na medida da benchmark, aplicar uma abordagem experimental para testar a segurança de sistemas e prover procedimentos e regras que podem ajudar no desenvolvimento de um padrão de benchmark de segurança, faz-nos acreditar de que este trabalho é uma contribuição relevante, tanto para a indústria quanto para a investigação científica de âmbito académico.

Palavras-chaves:

Segurança; medidas; *benchmarking*; risco; risco de segurança; sistemas; *web serving systems*

LIST OF PUBLICATIONS

This thesis is supported by the following papers, peer reviewed and published in well-known conferences:

P1. Mendes, N., Durães, J., Madeira, H. "Security Benchmarks for Web Serving Systems," The 25th IEEE International Symposium on Software Reliability Engineering (ISSRE-2014), Naples, Italy, November 2014.

P2. Mendes, N., Durães, J., Madeira, H. "Benchmarking the Security of Web Serving Systems Based on Known Vulnerabilities," 5th Latin American Symposium on Dependable Computing (LADC-2011), São José dos Campos, Brazil, June 2011.

P3. Mendes, N., Durães, J., Madeira, H. "Evaluating and comparing the impact of software faults on web servers," Proc. of the Eighth European Dependable Computing Conference (EDCC-2010), Valencia, Spain, 2010.

P4. Mendes, N. "Risk-based Security Benchmarks for Web Serving Systems (Student Forum)," Proc. of the 39th Int. Conf. on Dependable Systems and Networks (DSN'09), Lisbon, Portugal, 2009.

P5. Mendes, N., Durães, J., Vieira, M. and Madeira, H., "Towards Assessing the Impact of Security Attacks in Web Servers (Fast Abstract)," Proc. of the 39th Int. Conf. on Dependable Systems and Networks (DSN'09), Lisbon, Portugal, 2009.

P6. Mendes, N. and Araújo Neto, A. and Durães, J. and Vieira, M. and Madeira, H., "Assessing and Comparing Security of Web Servers", 14th Pacific Rim International Symposium on Dependable Computing (PRDC'08), Taipei, Taiwan, December 2008.

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

P7. Vieira, M. and Mendes, N. and Durães, J. , "A Case Study on Using the AMBER Data Repository for Experimental Data Analysis",

SRDS 2008 Workshop on Sharing Field Data and Experiment Measurements on Resilience of Distributed Computing Systems (jointly organized with the 27th International Symposium on Reliable Distributed Systems, IEEE SRDS 2008), Napoles, Italy, October 2008

P8. Vieira, M. and Mendes, N. and Durães, J. and Madeira, H. , "The AMBER Data Repository", DSN 2008 Workshop on Resilience Assessment and Dependability Benchmarking (DSN-RADB08), IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2008), Anchorage, Alaska, June 2008

P9. Mendes, N. and Durães, J. and Vieira, M. and Madeira, H. , "Security Assessment and Testing Tools Information Repository", Workshop on Fault Tolerance (WTF2008) in the 26th Brazilian Symposium on Network Computing (SBRC2008), RJ, Brazil, May 2008

P10. Moraes, R. and Barbosa, R. and Durães, J. and Mendes, N. and Martins, E. and Madeira, H. , "Injection of faults at component interfaces and inside the component code: are they equivalent?", 6th European Dep. Comp. Conference – EDCC-6, Portugal, October 2006

P11. Mendes, N. and Moraes, R. and Martins, E. and Madeira, H. , "Jaca Tool Improvements for Speeding Up Fault Injection Campaigns", Brasilien Symposium on Software Engineering, Florianópolis, Brasil, October 2006

P12. Mendes, N. and Moraes, R. and Martins, E. and Madeira, H. , "Improving dependability of software components by using wrappers", in Proc. of the VII Workshop on Tests and Fault Tolerance, Curitiba, Brasil, May 2006

P13. Moraes, R. and Martins, E. and Mendes, N., “Fault Injection Approach based on Dependences to Validate a System”, First International Workshop on Testing and Quality Assurance for Component-Based Systems (TQACBS 2005), http://aquila.nvc.cs.vt.edu/compsac2005/workshop/tqacbs05/, Scotland, 2005

P14. Moraes, R. and Martins, E. and Poleti, E. and Mendes, N., “Using Stratified Sampling for Fault Injection”, Second Latin-American

Symposium on Dependable Computing (LADC 2005), http://www.lasid.ufba.br/ladc2005/index.html, Salvador, BA, Brasil, 2005.

ACKNOWLEDGMENTS

To my advisors, Prof. Henrique Madeira and Prof. João Durães, for believing in my potential and for being a model of perseverance to research work. To Prof. Henrique Madeira, a special thanks for providing the support I needed when I moved to Portugal in 2005, for the energy and words of encouragement aimed at keeping me motivated during this long journey, and for pointing out the many weaknesses I had, which helped me to improve my research skills in each phase of my PhD preparation. To Prof. João Durães, my gratitude for being such a rigorous reviewer and for his sincere friendship.

To Prof. Regina Moraes, one of the toughest and best undergrad Professors I had, for being my first research model and for believing that I would succeed in her scientific initiation project, and for her sincerity about the hard work I would face. I am grateful for her dedication in the CAPES-GRICE project that gave me the opportunity to join the CISUC research team at University of Coimbra and then pursue a PhD degree. To her, my appreciation for her bravery and hard work when facing her own challenges (which inspired me), for her friendship and demonstration of love to my wife and children, and for the "never give up" and "keep fighting" advice in the hardest periods of this work. I owe her the many good things that happened to me during this research experience.

To Professors Mario Rela and Marco Vieira for their concern about the progress of this research and for the recommendation letters that helped me to successfully apply to FCT research scholarship. I also thank these Professors, along with Professor Maria Emília Nogueira**,** for their dedication to the PhD classes.

To Luis Mauricio, Nuno Laranjeiro, Davi Baccan, Ivano Irrera, Nahgmeh Ivaki, Paulo Veras and other PhD students, for their friendship and for their comments about the seminal ideas of this work. To Luis Mauricio a special thanks for being tireless in our peer review meetings.

To the University of Coimbra (in particular to the Department of Informatics Engineering and the Center of Informatics and Systems of the University of Coimbra) and State University of Campinas (in particular to the Faculty of Technology, where I graduated) for providing all the infrastructure and resources

from the beginning of my formal studies (University of Campinas) to the conclusion of this PhD work (University of Coimbra).

To the members of the System Software Engineering Group for participating of the workshops where I presented the main ideas and status of this work and for the tough feedbacks that motivated me to better approach the research question of this work.

To the anonymous reviewers of my research papers for their insightful comments and advices that helped me to deliver a better research work.

To the members of the Coimbra Baptist Church for treating me and my beloved ones as part of their family and for providing the support I needed while I lived in Portugal.

To Olavo de Carvalho for his work and advices about philosophy, intellectual life, and modern science.

To Father Paulo Ricardo, Father Cesário, and Father Marcos for their spiritual advices that gave me more strength especially at the end of this journey.

To Maxwell and Lilian do Carmo, Luis Afonso, Fabio Barreto, Daniel Berça, and Evandro Madeira for the many joyful moments that we lived together and for the encouragement they provided to finish this research.

To Bruno Ferreira, Paula Cruz and other Portuguese friends for their friendship and for being available every time I needed them.

To my father, Aloizio Reis Mendes, and my mother, Aparecida Vicente Mendes, for taking care of me from the very beginning and supporting me with their prayers while I was abroad. To my lovely sister Queila, my grandmother Cantides, my mother-in-law Ailza and others members of my family for the warm welcome when I arrived to visit them. I deeply love you all.

To my lovely wife (and best friend) and children, the sources of my strength, for staying with me and for their support during this long path.

To God for bringing me into existence and for giving me the health I needed to work on this project and to take care of my beloved ones.

TABLE OF CONTENTS

1. INTRODUCTION 26

1.1 CONTEXT AND MOTIVATION 26

1.2 THESIS OBJECTIVE 29

1.3 THESIS CONTRIBUTIONS 31

1.4 THESIS STRUCTURE 33

2. BACKGROUND & RELATED WORK 35

2.1 BASIC CONCEPTS ON COMPUTER SYSTEM SECURITY 36

2.1.1 Early works on computer security 37

2.1.2 Security attributes 37

2.1.3 Vulnerability and attack definition and characterization 40

2.2 COMPUTER SYSTEM VULNERABILITIES & ATTACKS 46

2.2.1 Vulnerability root-causes and mitigation 46

2.2.2 Vulnerability and attack classification 49

2.2.3 Vulnerability and attack repositories 53

2.2.4 Vulnerabilities and attack statistics 54

2.2.5 Vulnerability risk assessment 57

2.3 SYSTEM BENCHMARING 61

2.3.1 Performance Benchmarks 63

2.3.2 Robustness Assessment and Benchmarks 65

2.3.3 Dependability Benchmarks 66

2.4 TOWARDS RISK-BASED SECURITY BENCHMARKS 74

2.4.1 Security metrics 74

2.4.2 Security assessment 76

2.4.3 Security Benchmarking Initiatives ... 81

2.5 CONCLUSION ... 86

3. BENCHMARKING THE SECURITY OF SOFTWARE-BASED SYSTEMS ... 89

3.1 BASIC CONCEPTS ... 90

3.2 SECURITY BENCHMARKING STRATEGY ... 92

3.3 STATIC PART ... 94

3.4 DYNAMIC PART ... 96

3.5 BENCHMARK COMPONENTS ... 101

3.6 BENCHMARK IMPLEMENTATION SPECIFICATION ... 104

3.6.1 Metric Calculator ... 105

3.6.2 Vulnerability Repository ... 106

3.6.3 Security Test Repository ... 109

3.6.4 Workload ... 112

3.6.5 Vulnerability Injector ... 113

3.6.6 Attackload ... 115

3.6.7 Security checker ... 120

3.6.8 Exploitability checker ... 123

3.7 CONCLUSION ... 125

4. BENCHMARK PROCEDURES AND RULES ... 127

4.1 BENCHMARK GENERAL PROCEDURE ... 127

4.2 BENCHMARK INSTRUMENTATION RULES ... 129

4.3 BENCHMARK DEPLOYMENT PROCEDURE ... 131

4.4 STATIC PART PROCEDURE ... 132

4.4.1 Preparation Stage ... 133

4.4.2 Vulnerability Extraction and Analysis Stage ... 133

4.4.3 Security Test Stage ... 133

4.4.4 Measurement Stage ... 134

4.5 DYNAMIC PART PROCEDURE 135
4.5.1 Preparation Stage *137*
4.5.2 Vulnerability Injection Stage *138*
4.5.3 Attackload Generation Stage *139*
4.5.4 Baseline and Attack Execution Stage *139*
4.6 BENCHMARK METRIC COMPOSITION PROCEDURE 143
4.7 BENCHMARK DISCLOSURE REPORT 143
4.8 CONCLUSION 144
5. BENCHMARK IMPLEMENTATION 146
5.1 BENCHMARK TARGET DEFINITION 147
5.1.1 Web Servers *148*
5.1.2 Web Application *148*
5.2 INSTANTIATION OF THE BENCHMARK RULES FOR WEB SERVING SYSTEMS 149
5.2.1 Vulnerability Repository *149*
5.2.2 Security Test Repository *149*
5.2.3 Workload *150*
5.2.4 Vulnerabilityload *151*
5.2.5 Attackload *151*
5.3 STATIC PART IMPLEMENTATION 152
5.3.1 Experiment Controller *152*
5.3.2 Vulnerability Repository Implementation *152*
5.3.3 Vulnerability Extractor and Analyzer (VEXA) *154*
5.3.4 Version and Configuration Detector Implementation *156*
5.3.5 Security Tester Implementation *157*
5.3.6 Static Risk Calculator *160*
5.4 DYNAMIC PART IMPLEMENTATION 161
5.4.1 Experiment controller *161*

5.4.2 Workload Implementation 163
5.4.3 Vulnerability Injector Implementation 165
5.4.4 Attackload Implementation.. 167
5.4.5 Security Checker Implementation.. 169
5.4.6 Dynamic Risk Calculator 171
5.5 BENCHMARK RESULT CONSOLIDATION................................ 171
5.6 CONCLUSION 172
6. CASE STUDIES 174
6.1 BENCHMARKING THE SECURITY OF WEB SERVERS............ 175
6.1.1 EXPERIMENTAL SETUP 175
6.1.2 SYSTEMS UNDER BENCHMARK.. 175
6.1.3 BENCHMARK RESULTS & ANALYSIS 177
6.1.4 STATIC PART RESULTS 177
6.1.5 DYNAMIC PART RESULTS... 179
6.1.6 BENCHMARK RESULTS.. 184
6.2 BENCHMARKING THE SECURITY OF WEB SERVING SYSTEMS BASED ON KNOWN VULNERABILITIES... 185
6.2.1 SYSTEMS UNDER BENCHMARK.. 186
6.2.2 VULNERABILITY REPORT RESULTS..................................... 188
6.2.3 COMPONENT BENCHMARK RESULTS................................. 188
6.2.4 SYSTEM BENCHMARK RESULTS.. 189
6.3 BENCHMARK PROPERTIES VALIDATION 191
6.3.1 Representativeness 192
6.3.2 Repeatability... 193
6.3.3 Portability... 193
6.3.4 Non-intrusiveness 194
6.3.5 Feasibility... 195
6.4 CONCLUSION 196

7. CONCLUSION 198

8. REFERENCES 203

LIST OF FIGURES

Figure 2-1. Intrusion composite fault (Powell, Stroud, and others 2003) 43

Figure 2-2. IBM X-Force Report – Vulnerability Disclosure Growth by Year ... 56

Figure 2-3. CVSS categorization of vulnerabilities captured by X-Force Team during the year of 2011 61

Figure 2-4. Main components of a dependability benchmark (Figure taken from (Marco Vieira 2005)).. 67

Figure 2-5. System Under Benchmark and Benchmark Management System Interaction (Durães, Vieira, and Madeira 2004)... 68

Figure 2-6. Software fault injection and system observation (Duraes and Madeira 2006).. 69

Figure 3-1. Relationship between Benchmark Management System (BMS) and System Under Benchmark (SUB) in a Security Benchmark Execution............... 91

Figure 3-2. Attacks executed against a Vulnerable Component 98

Figure 3-3. Security Risk Calculator.. 106

Figure 3-4. Vulnerability Repository Data Model... 108

Figure 3-5. Security Repository Data Model 111

Figure 3-6. Attack injector components 118

Figure 3-7. Security checker components 122

Figure 4-1. Security Benchmark and Instrumentation Components 129

Figure 4-2. Benchmark Dynamic Part Components... 136

Figure 4-3. Inspection and Attack Execution Stage 141

Figure 5-1. Web Application Architecture Example.. 148

Figure 5-2. Experiment Controller of the Static Part.. 153

Figure 5-3. VEXA Information flow... 155

Figure 5-4. Nikto web server scanner components 158
Figure 5-5. Security tester – Example of Nikto security tests 159
Figure 5-6. Experiment Controller Dynamic Part 162
Figure 6-1. Security Benchmark Results for Web Serving Systems using different configuration 191
Figure 6-2. Benchmark Results for Web Serving Systems with different technologies... 192

LIST OF TABLES

Table 2-1. Mapping Security Errors to the Seven Pernicious Kingdoms and to the OWASP Top 10 Vulnerabilities *(Tsipenyuk, Chess, and McGraw 2005)* 48

Table 2-2. Examples of Vulnerabilities Databases... 55

Table 3-1. Top 10 Vulnerability Found unpatched in Web-based Systems (Symantec 2015)... 97

Table 3-2. Missing Function Call Extended (Jose Fonseca 2011) 116

Table 3-3. Software faults types 117

Table 3-4. Confidentiality Impact Assessment Rules 123

Table 3-5. Integrity Impact Assessment Rules... 124

Table 3-6. Availability Impact Assessment Rules.. 124

Table 5-1. HTTP Protocol Attacks 168

Table 6-1. System Under Benchmarking 176

Table 6-2. Static Part Benchmark Results.. 177

Table 6-3. Vulnerabilities discovered by Nikto 178

Table 6-4. Total Attacks & Vulnerability & Benchmark Duration.................... 180

Table 6-5. Loss of Availability Results.. 181

Table 6-6. Loss of Integrity Results 183

Table 6-7. Dynamic Part Results: Security Risk 184

Table 6-8. Web Server Ranking 185

Table 6-9. Vulnerability Distribution By System Component........................... 187

Table 6-10. Web Serving Systems Groups... 188

Table 6-11. Security Risk By System Component... 190

Table 6-12. Security Benchmark Results for Repeatability validation 194

Table 6-13. Benchmark Execution Time 196

CHAPTER 1

1. INTRODUCTION

This PhD Thesis presents a novel, risk-based methodology to measure and compare the security of software-based systems (termed here as *security benchmark*). This methodology uses the notion of risk to compute the benchmark metric (*security risk*) by taking into account the impact and occurrence probability of attacks targeting vulnerabilities present in the system under benchmarking. Our methodology allows users to build security benchmarks to assess and compare the security risk of functionally equivalent systems, and includes a well-defined set of rules to guide users to define and validate security benchmarks. In this thesis, we also provide an example of security benchmark for web serving systems and a case study with real web serving system components that demonstrates the applicability and usefulness of our benchmark approach. This chapter addresses the context and motivation of our research work, and includes a description of its research goals. This chapter comprises an overview of the thesis contributions and concludes with a brief description of the remaining chapters of this thesis.

1.1 CONTEXT AND MOTIVATION

Computer-based systems are becoming increasingly important and critical in our society. These systems have transformed the way people interact and make businesses, providing a place to store, share and disseminate all sort of information, a real-time communication mechanism to interact with one another and with institutions, and a platform to conduct business. In the last decade, the use of computer-based systems has grown globally, especially after the explosive growth of the World Wide Web in the last decade of the twentieth century. Statistics indicate that nearly one third of the world population already access the World Wide Web (Group 2011; U.S. Census Bureau 2009) in 2009. Certainly, this percentage is higher today. The World Wide Web (www) is currently the

basis for many public systems (e.g., access to governance services, public utility requisitions, e-commerce, banking transactions, stock market operations, social networks, enterprise and government management and so on), forming in fact a critical support of modern society where security and reliability are important for the general public. This clearly means that if a weak point (termed in the security field as *a vulnerability*) existing in a critical computer-based system is successfully exploited, it could lead to credibility and financial losses of institutions (e.g., due to credit card theft), and even exposure of critical infrastructure to risk of disruption.

Examples of successful vulnerability exploitations targeting computer systems are quite abundant in the real world. In October 2014, JP Morgan Chase, the US largest bank by assets, reported that its systems were compromised in mid-August, with customer contact information stolen from 76 million households and 7 million small business data (Goldstein, Perlroth, and Corkery 2014). This security breach was apparently the result of a weak authentication mechanism in an unpatched server that was used as the entry point, as a stolen password allowed hackers to get access to restricted system areas and gain high-level access to more than 90 bank servers. In March 2013, a known vulnerability in open Domain Server Name resolvers (US-CERT 2006) led to what security specialists called the biggest cyber attack of its kind in history (D. Lee 2013): a Denial of Service attack against Spamhaus's Domain Name System servers (an Internet company that tracks spam providers). These servers were flooded with 300 gigabits of data per second, effectively rendering it unavailable. These attacks were so intense that they brought down the Internet performance in some parts of Europe. In January 2011, The Canada Government disconnected three government agencies from the Internet, as attackers exploited vulnerabilities in their systems by installing malware (malicious software) that sent classified information back to the attackers (Weston 2011). Also in 2011, 24.6 million accounts of Sony Online Enterprise were stolen (including 12,700 non-U.S. credit or debit card numbers) due to exploited vulnerabilities in Sony on-line game services (SOE 2011). The cost of these attacks against Sony was initially estimated to more than 155 million of USD dollars, and recently Sony UK was fined 394,500 dollars by the UK privacy regulator (Bodoni 2013) as Sony could have prevented the breach by keeping software up-to-date and ensuring that passwords were secure. In the United Kingdom alone, the Ponemon Institute found out that the most expensive incident in 2009 cost £3.9 million, whereas in the United States it cost $31 million (Titterington 2010). In 2012, the average cost of cyber crimes for 56 organizations in the United Stated (that resulted from 102 successful attacks per week) was $8.9 million, with a range of US$ 1.4 to 46 million (Ponemon 2012). In May 2015,

IBM and Ponemon Institute released a new data breach cost study covering 350 companies spanning 11 countries. In this study, the average total data breach cost is $3.79 million (Ponemon 2015), with cyber-attacks representing 47% of the root cause of a data breach and an average cost per record of US$ 170. To counter these attacks and avoid even greater financial and credibility losses, vulnerabilities in computer-based systems should be avoided or countered at any cost and security should be treated as a top priority.

Despite the investment on security by governments and enterprises (e.g., (CNCI 2008)), attacks against IT systems have increased in the last years (IBM X-Force 2012) and have reached mission-critical systems that could affect humans lives (Sanger 2012; Arthur 2012). According to (Titterington 2010), several security firms have reported an increase in security incidents. The Computer Security Institute (CSI 2012), for example, reported that the number of organizations infected by malware during 2013 was 64%. A report released by IBM X-Force team indicates that 2011 was the year of security breaches (IBM X-Force 2012), with 7,000 vulnerabilities being disclosed during that year. This is mainly due to the increasing complexity of systems, leaving room for more vulnerabilities that can be exploited by attackers.

Many techniques and methods have been proposed to increase the security of software-based systems, i.e., to achieve a level of protection to deter attacks (from tools such as anti-virus, firewalls, intrusion detection systems to security recommendations in the form of security practices, checklists and security standards). However, the only assured way of knowing if the security of a given system was indeed increased is by evaluating it. In other words, security evaluation is always required even when other security practices and mechanisms are employed. This adds to the relevance of having a methodology for security evaluation.

Security evaluation methods and tools are important to identify vulnerabilities in computer-based systems, helping software vendors, buyers, users, and administrators, among others, to identify which components of the system are more prone to be attacked, to prioritize which part of the system should be secured first. Also, one common practice among software buyers and end-users is to rely on a set of security evaluation methods (e.g., ISO 17799 2005; NIST-SP800-12 1995; CC Protection Profiles 2012; M. Vieira and Madeira 2005; Mendes et al. 2008) and tools (Acunetix 2012; IBM Appscan 2012; Nikto2 2015; Curphey and Arawo 2006; SecTools 2014) to help them to get an estimation of the security level of a given system based on a set of security requirements, tests, or software maturity (e.g.: is the system free from known vulnerabilities? Are the

common attacked ports properly closed?). However, current security evaluation methods and tools have well-known limitations. For example, they provide a fragmented view of the security level of the system and there are few methods to evaluate the security of software-based systems in a systematized and standardized way that can be used by end-users to measure their security for comparative purposes (*security benchmark*).

The development of comprehensive and easy-to-use approach to benchmark the security of competing systems/component is then a real need in modern society. To help in this effort, this thesis presents a new contribution to the field of security evaluation and benchmarking of software based systems, helping to reduce the lack of security methods to measure and compare software security.

1.2 THESIS OBJECTIVE

This thesis proposes a novel methodology to support the development of functional and effective security benchmarks that can be applied over any class of software-based system. This methodology uses security risk as the benchmark metric, with a single metric (*SBench*) that enables users to compare system security. To the best of our knowledge, the notion of security risk has not been used for the definition of security benchmark metrics (although the notion of risk has been widely used in security evaluation, outside the benchmarking scope) and our decision to use it is based on the fact that this metric is able to translate into a single number the risk of vulnerabilities present in a software system. The purpose of this number (metric) is to indicate the security level of the system under benchmark, helping users to identify which system to use when faced with the need to select one system among functionally equivalent ones.

This thesis exemplifies the proposed security benchmark methodology by providing a security benchmark for web serving systems, also describing the tools implemented to speed up the benchmark execution. Web serving systems form the basis of many services, such as e-commerce and banking systems. These systems are heterogeneous and complex, based on several discrete components. This internal complexity potentiates the existence of vulnerabilities that might be exploited by attackers. Because these systems are naturally connected to the Internet, and thus exposed to many users and attackers, any internal vulnerability becomes a real threat to security (e.g., (OWASP 2013; B. Martin et al. 2010)). Therefore, the web-serving scenario as case study of our benchmark methodology is relevant. In fact, in this thesis we have the purpose of demonstrating the applicability of the benchmark prototype by conducting case studies to measure and compare the security of real web serving systems. Additionally, this thesis

and the research work supporting it provide to the community results, tools and an increase in knowledge concerning security at general, and in web serving systems in particular.

As mentioned, our benchmark metric (*SBench*) is estimated based on the security risk of vulnerabilities present in the system under benchmark. This is done by computing the risk related to vulnerabilities that are already discovered (*known vulnerabilities*) and by estimating the effect of not-yet discovered vulnerabilities (*unknown vulnerabilities*) on the system. This in fact corresponds to the two parts of our security benchmark methodology: static and dynamic parts. The assessment of the risk of known vulnerabilities (static part) corresponds to a static analysis of the target system and uses the knowledge about the impact and exploitability of vulnerabilities discovered in the field for that system to measure the security risk. These known vulnerabilities are obtained from two sources: (i) public repositories such as vulnerabilities databases and specialized web sites (e.g., (NVD 2014; OSVDB 2014; US-CERT 2014)); and (ii) results from security tests usually proposed by security experts. One important aspect is that vulnerability impact and exploitability are estimated considering the criteria defined by the Common Vulnerability Scoring System (CVSS) (Mell, Scarfone, and Romanosky 2007): this vulnerability framework has been widely used by large enterprises to characterize the risk of software vulnerabilities.

The assessment of the effects of unknown vulnerabilities (dynamic part) corresponds to an experimental approach where robustness attacks are conducted to observe the behavior of the system. This approach is properly detailed in Chapter 3, but it is worth pointing out that we do not propose a way to identify unknown vulnerabilities. Our experimental approach (already applied in dependability benchmarks to test the tolerance of system to software faults) stress the system with attacks, observe the impact of these attacks, and then estimate the security risk in case if these attacks were successful (i.e., the attack compromised at least one of the security attributes of confidentiality, integrity, and availability). In practice, this is done using two complementary steps: (1) stressing the system with malicious input parameters and multiples attacks (e.g., Denial of Services attacks, Buffer Overflow) directed against components that interact with end-users; and (2) mounting attacks against representative vulnerabilities that are injected in a component that is not included in the benchmark target (but interacts with it). By representative we mean the injection of vulnerabilities that are usually found (and consequently more exploited) in the target system.

The purpose of injecting vulnerabilities and attacking them is to anticipate if a security breach (a successful attack that leads to a security compromise) in a

component may affect the security of the whole system under benchmark. For example, by injecting vulnerabilities in the web application (it is plausible to assume that applications may have vulnerabilities) we can assess if the attacks launched over such vulnerabilities can compromise the system. One important aspect here is that the vulnerability and attack injection approach is actually the technique that allow us to assess the effects of unknown vulnerabilities to the system. As attacks exploit vulnerabilities injected in a component that is different from the benchmark target (i.e., the component is outside the perimeter of the benchmark target), any security compromise of the benchmark target during the execution of such attacks was caused by the presence of one or more unknown vulnerabilities (weak points). This idea was already applied in fault injection field to assess the behavior of fault tolerant system in the presence of erroneous software components and the goal here is to apply this concept to the security field, using the attack injection technique proposed in (J. Fonseca, Vieira, and Madeira 2009) as detailed later on.

1.3 THESIS CONTRIBUTIONS

Although we recognize the existence of security benchmarking initiatives in literature as important contributions to the definition of security benchmarks (in Chapter 2, we present these works and discuss their intent and characteristics), we believe that our security benchmark methodology is unique in the sense that it incorporates the notion of security risk into the benchmark metric, takes advantage of an experimental approach to stress the security of the system with representative vulnerabilities and attacks and to assess the effect of unknown vulnerabilities, and uses the elements and approach of performance and dependability benchmarks applied to the security field. These elements and approach are relevant because they have been successfully applied to build realistic and repeatable benchmarks in other fields and guided us to build a realistic security benchmark for web serving systems.

The fact that our benchmark uses the notion of risk, applies an experimental approach and do so in a repeatable and reproducible way, make us to believe that our security benchmark methodology is the best option considering the most recent security evaluation initiatives that ultimately seeks to help users to select the most secure among functionally equivalent software. In fact, there are security assessment methodologies for web servers (e.g. (CIS 2008)), but they are very far from what can be considered a benchmarking, especially to what concerns the fulfillment of benchmark properties such as representativeness, portability, scalability and the translation of the security level of systems in a single metric. Also, the software product evaluation method proposed in (Das, Sarkani, and

Mazzuchi 2012) is focused only on known vulnerabilities present in the system, with no experimental approach to test the security behavior of the target system and limited to certain application domains. The security-benchmarking framework described in (Neto 2012) uses a security qualification to target known vulnerabilities that do not consider the individual risk of known vulnerabilities to the whole system. Also, the benchmarking approach they use to target hidden vulnerabilities is not based on an experimental approach, but on the identification of the characteristics that the system has to avoid the effect of system threats.

One of the novelties of our methodology is the extension of the benchmarking concept that has been successfully applied to the performance and dependability fields to the comparison of security features of web serving systems/components. This involves the research of new benchmark components, such as the vulnerability repository, the security test repository, the attackload (adapting the vulnerability and attack injection technique proposed in (J. Fonseca, Vieira, and Madeira 2009)), and the security risk metric.

The main contributions of this PhD Thesis are as follows:

- The proposal of an innovative and easy-to-follow methodology for defining and implementing security benchmarks, based on both static and dynamic aspects of the target systems, including known vulnerabilities and an estimation of potential damage from unknown vulnerabilities. This methodology can be adapted to any class of software-based system.
- A single security metric (SBench) to simplify the comparison of different systems (or different configuration of the same system), allowing as well the breakdown of this single metric for more detailed analysis.
- A practical approach to inject vulnerabilities and execute attacks to observe the behavior of the systems regarding yet to be discovered vulnerabilities.
- A prototype security benchmark for web serving systems implemented following the proposed benchmark methodology. This benchmark implementation enables users and system integrators to identify the most secure from a set of functionally equivalent web serving systems, and is also aimed at helping benchmark implementers to build security benchmarks.
- The proposal of supporting tools to speed up the extraction of information and analysis of vulnerabilities, to inject vulnerabilities and attacks against web serving systems components in order to assess their behavior

concerning security, to monitor and assess the impact of attacks and vulnerability injection against web serving systems.

- A comprehensive case study benchmarking the security of widely used web-serving systems providing useful data to the community on web serving systems actually used in the field. This case study was used to validate our security benchmark methodology and to demonstrate that it can effectively be used to benchmark the security of these systems, providing results that allowed us to clearly identify the most secure among the evaluated systems. We believe that providing the computer industry and user communities with examples of risk-based security benchmarks, in addition to the validation of those benchmark proposals, is a significant step to the definition of a general security benchmark standard accepted by companies and governments.

In conclusion, this thesis is a contribution on how to build realistic security benchmarks, on how to define and use security risk as a benchmark metric, and on how benchmark the security of real web serving systems. Another natural contribution is the advance of security benchmarking state-of-the-art.

1.4 THESIS STRUCTURE

The structure of this thesis is organized as follows:

Chapter 2 presents the theoretical foundations for this research work and existing work related with this thesis, focusing on the state of the art on security measurement and benchmarking.

Chapter 3 provides a detailed overview of the benchmark definition methodology. This also includes the description of security benchmark elements, with special emphasis on the description of our security metric composition.

Chapter 4 details the benchmark procedures and rules that should be taken in account to implement and deploy our security benchmark.

Chapter 5 describes the implementation of our benchmark methodology for web serving systems, covering the entire development process of the supporting tools. This includes the extraction and analysis of real vulnerabilities, the injection of vulnerabilities and attacks, the management of security benchmark experiments, and the measurement of the benchmark metric.

Chapter 6 exemplifies the use of the benchmark implementation with a case study and shows the benchmark results and its validation. The case study uses real web serving systems, and the validation targets the benchmark properties of

representativeness, repeatability, and portability, among others.

Chapter 7 concludes this thesis and presents directions for future work to advance the security benchmark approach proposed in this thesis.

CHAPTER 2

2. BACKGROUND & RELATED WORK

"Research is the search for knowledge through objective and systematic method of finding solutions to a problem" (Kothari 2008)

This chapter presents background and research work related to our own and relevant to the definition, development and validation of security benchmarks for web serving systems. This chapter also includes the descriptions of key concepts regarding security following the organization described below:

Section 2.1 describes the concepts of computer system security, briefly describing early works on computer security, and then it presents the primary and secondary attributes used to verify whether or not a system is in a secure state. It also describes the concepts of vulnerability and attack, including the notions of vulnerability life-cycle and vulnerability risk.

Section 2.2 presents works concerning the characterization and representativeness of vulnerabilities and attacks, including an overview about vulnerability causes and classification schemes. We also describe existing repositories containing information and statistics on vulnerabilities, and then we provide examples of vulnerability scoring methods. Some of these works form the conceptual background of our security benchmark methodology.

Section 2.3 details works on performance, robustness and dependability benchmarking. These research topics are relevant to our work as we use them in the definition of the main elements of our security benchmark methodology (metrics, procedures and rules, experimental setup, workload, and so on).

Section 2.4 focuses on security assessment, which includes security metrics, and assessment techniques and methods. In this context, we also present security benchmarks initiatives that have been proposed in recent years.

Section 2.5 concludes this chapter.

2.1 BASIC CONCEPTS ON COMPUTER SYSTEM SECURITY

Computer system security is the idea of engineering systems so that it continues to function correctly under malicious attacks (definition adapted from (McGraw 2004)). The goal of computer system security mechanisms is to prevent, detect, and recover from attacks (Bishop 2003). A computer system **attack** is any action aimed at compromising the security of a system (Stallings 1999). In this sense, **attack prevention** refers to the measures set in place to counter attacks. Successful prevention means that an attack will fail. **Attack detection** refers to the measures aimed at detecting an attack attempt. **Attack recovery** aims to bring the system back to a secure state after a successful attack. Recovery has two forms: (1) to recover the missing or modified resources and (2) to fix the vulnerability that enabled a given attack. The attack prevention and detection capabilities are more important to our work as they directly affect the security level of a system. The deployment of firewalls and intrusion detection systems are examples of measures that have been employed to make attack attempts unsuccessful. The use of backup tools is very important to minimize the impact of attacks over the business, but these tools do not play the role of countering attacks and making systems more secure. This distinction is important as the focus of our work is on the assessment of the security level of software-based systems.

The object of protection in a computer system is the **asset**. An asset is anything that has value to an organization (software, hardware, people, data, etc.) and which therefore requires protection. In what regards computer systems, an asset can be one of several software and hardware components along with the data that is kept, manipulated, or transmitted (depending on the type of the system and its mission). In the context of information security, the **data** is an important asset that is stored, locally, or distributed across several computers (Kaufman 2009). If we consider that these data can be highly sensitive and confidential (such as credit card and social security numbers, or financial data of companies and governments, etc.), it is very clear that the consequences of data theft can be very negative. This notion is important because our security benchmark methodology focus on the security deficiencies that affect the logical part of a software-based system. The security of the areas where a computer infrastructure is located (physical part) and procedures to reduce the risk of theft, fraud, or misuse of

facility (human part), which are described in (ISO 17799 2005), are not the object of study of the present thesis.

2.1.1 Early works on computer security

The topic of computer security is not new. In 1976, papers regarding security, design of software protection, operational practices, and auditing were already numbered by the thousands. (Browne 1976) provided an overall perspective of 134 papers and clearly demonstrated that security was a major concern since the creation of the first mainframe computers. As a point of clarification, it is worth noting that the first computer generation was primarily designed for military purposes, and its data was classified and should be protected against the enemy at any cost. The ENIAC (Eletronic Numeric Integrator and Computer), announced in 1946, was one of the first all–purpose electronic computers and was designed to compute artillery firing tables for the Ballistic Research Laboratory of the United States Army (Eckert 1964; Weik 1961; McCartney 1999).

The Computer Security Laboratory of the Computer Science Department at the University of California, Davis made available a list of works produced on the early 70s and 80s in the computer security field. This list was then made available by the Computer Security Resource Center of the National Institute of Standards and Technology at (NIST 1998). It is very clear from these works that the design of a secure system (Schiller 1975), the development of security controls for computer systems (Ware 1970; Nibaldi 1979), and the analysis of vulnerabilities (Karger and Schell 1974) were among the concerns of former computer security researches. In summary, 66 years after the creation of the first electronic computer and a huge technology progress, security continues to be a topic of concern for governments, military, enterprises, and academia.

2.1.2 Security attributes

The security of a computer system can be described in terms of *primary* and *secondary* attributes. The difference between these two categories is that the former refers to the key attributes a system must have to be secure, while the secondary security attributes are the instantiation of these primary attributes to a given area and are generally associated with human users (or with components that act as users such as proxy agents, or web services).

The primary are confidentiality, integrity, and availability. According to (Avizienis et al. 2001), security is the concurrent existence of these primary attributes. A system is not secure if attackers are able to obtain restricted content, or to modify it, or make it unavailable. The partial loss of at least one of these

attributes is enough for the system enter an unsecure state.

The primary security attributes are described as follows:

- **Confidentiality** refers to the protection of functionality and data against unauthorized access (Bishop 2003). Confidential data access or confidential data transmission requires that unauthorized disclosure of one or more specific items will not occur (Walton, Longstaff, and Linger 2009). Access control mechanisms support confidentiality. One access control mechanism for preserving confidentiality is cryptography, which scrambles data to make it incomprehensible to unintended viewers (Bishop 2003).
- **Integrity** refers to the trustworthiness of data or resources, assuring that the actions and data are correct (Bishop 2003). Integrity requires that authorized changes are allowed, all changes must be detected and tracked, and changes must be limited to specific scopes (Walton, Longstaff, and Linger 2009). Integrity is defined as a property of the object, not of the mission. Integrity includes data integrity (the content of the information) and origin integrity (the source of data, often called authentication). Integrity mechanisms fall into two classes: *prevention* and *detection*. Prevention is aimed at maintaining the integrity of the data by blocking any attempts to change data in unauthorized ways. Detection is aimed at information that data integrity is no longer trustworthy (Bishop 2003).
- **Availability** refers to the readiness of the system to provide the expected service, i.e., to the ability to use the information or resource desired (Avizienis et al. 2001). Availability requires that a resource is usable despite attacks. In terms of security, a malicious user may arrange to deny access to data or to a service by making it not available. One avenue that availability mechanisms can use is to seek atypical events that might lead the system to become unavailable or unresponsive (Bishop 2003).

The secondary security attributes are described as follows:

- **Accountability** refers to the record of any security-related action that should also be available even if the user is no longer connected (Goertzel et al. 2006). In other words, this refers to the availability and integrity of the identity of the person who performed an operation (Avizienis et al. 2001).
- **Authenticity** refers to the integrity of a message contents and origin, possibly of some other information as well, such as time of emission

(Avizienis et al. 2001).

- **Authentication** is the process of establishing the user´s identities before they can access an application. As an example of action of authentication mechanisms, the system should allow a requested program to be executed only if the user has previously been identified as a trusted user (Stoneburner, Hayden, and Feringa 2004).
- **Authorization** refers to the access control to specific contents or components based on user privileges. Although several users may have access to a given system (i.e., they have personal credentials to access the system), authorization ensures that only the right users will get the information for the requested process (Walton, Longstaff, and Linger 2009).
- **Privacy** refers to the ability to define control over how his/her information will be disclosed (visualized or accessed by others) (Walton, Longstaff, and Linger 2009). One example is the social network sites that allow users to define who will access their personal content.
- **Non-repudiation** refers to data transmission that cannot be refuted by either part after an agreement has been established. (Avizienis et al. 2001) considers *non-repudiability* as the availability and integrity of the identity of the message sender (non-repudiation of the origin), or of the receiver (non-repudiation of reception). For example, an e-mail system with non-repudiation is the one that ensures that the recipient of a message cannot deny receiving it and that the sender cannot deny sending it.

Computer security has also been defined or specified in other terms. One possible definition is based on guidelines and checklists instead of attributes. These guidelines can later be checked in a similar way as quality procedures control. Security can also be equated in terms of techniques in place to help system administrators to observe and protect the target system against security incidents (e.g., firewalls, intrusion detection systems, and similar).

An example of computer security specified in the form of guidelines was written by Matt Bishop. (Bishop 2004) argues that computer security relies on three fundamental components:

- **Requirements.** These describe the needs of the user or institution in terms of security. Each organization may have its security goals and this should be clarified through the collection of security requirements. An

example of security goals is data protection against unauthorized access.

- **Policy.** This specifies the measures and steps to be taken to achieve the intended security goals. The policy consists then on a set of statements that specifies what is allowed and what is not. If the system is always in allowed states, and users can only perform actions that are allowed, then the system is *secure*. On the contrary, if the system can enter a disallowed state, or if user can execute a disallowed action, then the system is *nonsecure*.
- **Mechanisms.** These identify the tools, procedures, and other ways used to ensure that the policy is enforced. Security mechanisms can be technical (e.g., vulnerability scanners that identify known vulnerabilities) or operational (e.g., procedures aimed at protecting classified information).

2.1.3 Vulnerability and attack definition and characterization

In the context of software, a vulnerability is an instance of a mistake in the specification, development, or configuration of software such that its execution can violate the explicit or implicit security policies (Krsul 1998). (Andy Ozment 2007) proposed two changes in this definition. The term "mistake" is aimed to mean an incorrect result (Radatz, Geraci, and Katki 1990), and not an "error" as proposed originally. The terms "explicit" and "implicit" highlight the fact that every system has a security policy, even if it was not written. The definition proposed by (Krsul 1998) with the changes presented in (Andy Ozment 2007) is the one we take into account in the context of this thesis.

Other definitions for vulnerability can be found in literature. (Bishop 1999) defines vulnerability as a bug that enables users to violate the security policy. In (D. D. Clark et al. 1991), vulnerability is defined as a weakness in a system that can be exploited to violate the system intended behavior. (Arbaugh, Fithen, and McHugh 2000) sees vulnerability as a flaw or defect in a technology or its deployment that produces an exploitable weakness in a system, results in behavior that has security or survivability implications. Despite the fact that these definitions are not uniform, all of them treat vulnerability as a weakness that affects system security.

An **attack** is any action aimed at compromising the security of a system (adapted from (Stallings 1999)). An attack (also termed in this thesis as **vulnerability exploitation**) usually targets one or more vulnerabilities present in a system. If the vulnerability being exploited is in a network environment (Internet, intranet,

etc.), then an attack can also be named as **cyberattack** (adapted from (Liu and Cheng 2009)). A **threat** is an attacker that is motivated and capable of exploiting a vulnerability (Schneider 1999). If an attack exploits a vulnerability and executes a set of commands (like the cross-site scripting and SQL Injection attacks that use an input parameters to inject malicious code) or penetrate the system with a malicious program, then, the commands are referred to as the **payload**. The difference between payload and **exploits** is that the later refer to the tools used to conduct an attack.

One important distinction to be made is that between **attempted attacks** and **successful attacks**. The former refers to attacks that are unable to penetrate the system and cause harm. The latter refers to attacks that successfully compromise the security of the system. When an attack is successful, the consequence is a **security breach**, i.e., at least one of the security attributes of confidentiality, integrity, and availability were compromised.

The notion of successful attacks is a key aspect of our security benchmark methodology, as security measurements are taken into account only when attacks compromise the security attributes of the system. In this sense, it is quite important to emphasize the aspects that, in our view, contribute to the occurrence of successful attacks, which are as follows:

- **Attacks can be executed remotely.** An attacker can be an inside agent, being able to attack the system locally, accessing protected files and executing tools using higher privileges. However, the advent of Internet not only allowed the intercommunication and access of system, but also allowed attacks to be executed from a remote location, where laws may not effectively restrict attacks. Examples of physical and remote attacks can be found at (Easttom and Taylor 2010).
- **Attacks can be executed in different forms.** The diversity of attacks is related to the diverse nature of vulnerabilities, which can be located in any part of the design, specification, implementation, configuration and deployment of systems. Although one can learn from attacks that were executed and properly detected by detection tools, it is very hard to predict new forms of attacks. For example, if an attacker is skilled enough to change the way a known exploit is normally launched, this new form of attack may bypass intrusion detection systems in place. Descriptions of common methods for attacking systems are available in Common Attack Pattern Enumeration and Classification (CAPEC 2014) and (Hoglund and McGraw 2004). Examples of attack methods introduced in (Barnum and Sethi 2007) are HTTP response splitting, Structure Query Language

injection, Cross-site Scripting in HTTP query string.

- **Attackers have the advantage.** There are four reasons why attackers have the advantage over defenders (M. Howard et al. 2003): (1) the defender must defend all points; the attacker can choose the weakest point; (2) the defender can defend only against known attacks; the attacker can probe for unknown vulnerabilities; (3) the defender must be constantly vigilant; the attacker can strike at will; (4) the defender must play by the rules; the attacker can play dirty.
- **It is hard to prosecute attackers.** Maybe the most effective way to defend against criminal attacks is to put the attacker in jail (without a computer and Internet connection). A report about the crimes committed by attackers is available at (Easttom and Taylor 2010). However, cyber security laws may not be enforced and different countries may be used to launch attacks and avoid prosecution.

At this stage, it is quite clear that the existence of vulnerabilities is the most important enabling factor to the occurrence of successful attacks, allowing attackers to penetrate the system and execute malicious actions. These vulnerabilities can be present in a system in a variety of forms, as detailed in section 2.2.1, including software faults (code defects), missing configuration, and improper components integration. For example, a software fault that, once activated, allows the leak of confidential information is, in fact, a vulnerability, as this compromises the security attribute of confidentiality. This notion is important to us as in our methodology we inject software faults that lead to vulnerabilities and attack them while observing the security behavior of the system under benchmark. The remainder of this subsection focuses on the relation of software faults and vulnerabilities. Then, we describe the several conditions of a vulnerability from its creation to its elimination and we finish by describing the important concept of vulnerability risk.

2.1.3.1 Faults, vulnerabilities and security breaches

A vulnerability is the security equivalent to a fault in the dependability field (Brocklehurst et al. 1994). A **fault** is the adjudged or hypothesized cause of an error (Avizienis et al. 2001). An **error** is that part of the system state that may cause a subsequent failure: a **failure** occurs when an error reaches the service interface and alters the service (Avizienis et al. 2001). In the security context, the obvious analogy to system failure is a **security breach** (an event where the behavior of the system deviates from the security requirements) (Brocklehurst et al. 1994). Security breaches are made possible by vulnerabilities, such as the

improper validation of an input parameter in the system and these are clearly special types of faults. (Avizienis et al. 2004) terms vulnerability as an internal fault that enables an external fault to harm the system, a definition more tied to the dependability field.

A possible alternative to *successful attack* would be intrusion. An **intrusion** is a malicious operation fault that originates externally to the system boundaries (Powell, Stroud, and others 2003). Even a malicious faulty interaction executed by an insider can thus be classified as an intrusion since the goal is to conduct an operation that is unwanted by the owner of that resource. In fact, an intrusion can be considered as an attack that compromised the security of the system. Figure 2-1 presents an intrusion (successful attack) as a combination of two factors: (1) an attack that attempts to exploit a vulnerability in the system and (2) the existence of at least one vulnerability.

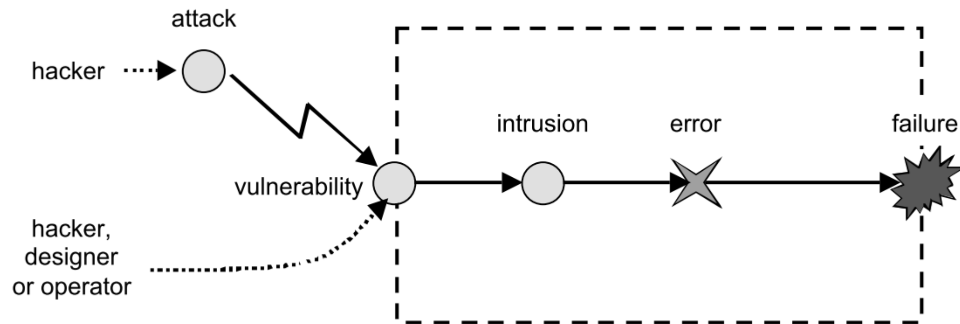

Figure 2-1. Intrusion composite fault (Powell, Stroud, and others 2003)

A system free of requirements, design, implementation, and deployment faults is obviously free of vulnerabilities. This does not mean that this system is not attackable. This simply means that attacks will not be successful against this system, i.e., that the security attributes of confidentiality, integrity, and availability will remain intact even in the presence of the worst attack attempt. However, the inherent complexity of systems and development processes makes it very hard to deploy systems completely free of vulnerabilities (Hatton 2007).

2.1.3.2 Vulnerability life-cycle

The **vulnerability life-cycle** is the time period between the moment a vulnerability is created (born) and the moment that the vulnerability is removed (dies). During its lifecycle, a vulnerability can be in one of the following states (Andy Ozment 2007):

- *Unknown vulnerability.* The vulnerability exists in the software/system

but has not yet been detected.

- *Secret Vulnerability.* The vulnerability has been detected, but it has not been informed to the vendor, the public, etc. If the person that discovered the vulnerability is an attacker, he or she may be exploiting the vulnerability.
- *Disclosed Vulnerability.* The vulnerability has been discovered, and the person who discovered it, informed the vendor or a disclosure institution. One important distinction here is that existing between **vulnerability discovery** and **vulnerability disclosure**. The former refers to the detection of a vulnerability by a party that keeps this information from developers and users. This party may use the knowledge about the vulnerability as an advantage to attack and compromise the security of the system involved. The latter refers to vulnerabilities that were discovered and reported to software developers or vendors/fabricants. The vendor may or may not decide to fix the vulnerability based on the risk that it poses to the system, however the vendor knows about the existence of the vulnerability. To help fabricants get notified about discovered vulnerabilities, CERT and popular vulnerability databases usually offer a platform to users to report vulnerabilities. The idea is to help the vendor to release a vulnerability patch before the vulnerability is made public.
- *Public Vulnerability.* The vulnerability has been detected and made public through a patch, a public forum, or the media.
- *Scripted Vulnerability.* There is an automation tool/script to exploit the vulnerability. As mentioned before, these tools are also termed as **exploits** in literature.

In this thesis, we use the notion of known and unknown vulnerabilities. However, we consider known vulnerabilities those that are either disclosed or in public domain. Also, we consider secret vulnerability as an unknown vulnerability, since the fabricant is not aware of the existence of such vulnerability. This distinction is important because they correspond to the view of security that we address in our security benchmark methodology.

2.1.3.3 Vulnerability risk

In order to understand the definition of vulnerability risk, it is necessary to firstly describe risk in a more general way. W. Lowrance considers risk as the measure of the probability and severity of adverse effects (Lowrance 1976) while (Rowe 1977) defines risk as the potential for realization of unwanted negative

consequences of an event. From these definitions, it is clear that there are two key characteristics associated with risk (Kirkpatrick, Walker, and Firth 1992): uncertainty (an event may or may not happen) and loss (an event may have unwanted consequences or losses).

The notion of risk is extensively applied in the security field. Typical security risk equations found in literature indicate that the risk posed by a particular threat is equal to the probability of the threat occurring multiplied by the damage potential (Meier et al. 2003). This damage potential refers to the consequence to a computer-based system if an attack were to occur. A typical security risk equation is as follows:

$$Risk = Probability * Damage\ Potential$$

For the two terms of the risk equation stated above, it is generally employed a scale that determines the value of each one of the factors. This scale is generally divided in bands to generate a High, Medium, or Low risk rating. An example of scale is as follows:

- **Probability:** 0-10 scale, where 0 represents a threat that will not occur and 10 represents the certainty of occurrence.
- **Damage Potential:** 0-10 scale, where 0 indicates minimal damage and 10 represents a catastrophic damage.

An important aspect is the risk posed by the vulnerabilities present in the system. The notion of vulnerability risk is based on the fact that each vulnerability has a likelihood of being exploited and, if successfully exploited, will compromise the system security at a certain degree (vulnerability impact). The existence of countermeasures or constraints to make the vulnerability exploitation more difficult is directly related to the probability of successful exploitation (Meier et al. 2003). The more difficult to exploit a given vulnerability, the lower is the probability of successful exploitation. The risk that a known or unknown vulnerability poses to a system can be very dangerous. According to (Goertzel et al. 2006), vulnerabilities jeopardize intellectual property, consumer trust, business operations and services, and a broad spectrum of critical applications and infrastructures, including everything from process control systems to commercial application products.

We define **vulnerability risk** as a measure of the probability and impact of vulnerability exploitation. This is an important topic as our benchmark methodology uses the notion of vulnerability risk to estimate the benchmark metric. **Vulnerability impact** refers to the perceived impact of the successful

attack and is related to the loss of confidentiality, integrity, and availability (security attributes). If an attack completely compromises the security attributes, then the vulnerability impact should be scored with the highest degree. It is important to note that the impact of any vulnerability refers to the effects in the system of the vulnerability exploitation.

2.2 COMPUTER SYSTEM VULNERABILITIES & ATTACKS

In this section, we present several works focused on the characterization and the representativeness of vulnerabilities and attacks. We start by describing the vulnerability root-causes, an important step in the elimination of the security deficiencies of a system security. Then, we provide relevant examples of vulnerability and attack classifications and repositories. After that, we present statistics about vulnerabilities and attacks, with the intent of demonstrating their prevalence and providing the evidence that supports the development of security benchmarking methods. We conclude this section by approaching the topic of vulnerability risk as this notion is used to compute the benchmark metric of our security benchmark methodology.

2.2.1 Vulnerability root-causes and mitigation

The subject of **vulnerability root-causes** is important to help designers, developers, and administrators to avoid faults that can lead to vulnerabilities. **Vulnerability mitigation** refers to the actions in place to identify and correct vulnerabilities in the system in order to increase its security. These notions are important to our thesis as this helps users to improve system security and obtain a better security measurement in our security benchmark methodology.

According to (Neves et al. 2006), vulnerabilities are usually created during the development phase of the system, or during operation. These vulnerabilities can be introduced accidentally or deliberately, with or without malicious intent. However, vulnerability can also be caused by an improper system design. For example, if a proper authorization requirement is not defined and designed, developers may develop code that correctly implements a flawed design, leading to a system that allows unauthorized access in restricted areas. The integration of third-party off-the-shelf components is another cause for vulnerabilities. Component integration may also introduce vulnerabilities in the larger system due to interface mismatches that may be exploited by attacks (Weyuker 1998). This is usually the result of components designed or developed with no security concerns in mind, as well as the lack of proper security testing prior to the component integration into the software product.

Several vulnerability **root-causes models** have been proposed in the last years (e.g., (Piessens 2002)). (Tsipenyuk, Chess, and McGraw 2005) proposed a taxonomy of security errors named as *the seven pernicious kingdoms*. In this context, “security errors” refer to programming errors that lead to vulnerabilities. The seven-kingdom taxonomy focuses on collecting common errors and explaining them in a way that makes sense to software developers. The main benefit of this approach is to make developers aware of these errors so that they can develop software free of vulnerabilities. The seven kingdoms are as follows: input validation and representation (metacharacters, alternate encodings, and numeric representations), API abuse (the caller fails to honor its end of the contract), security features (authentication, access control, confidentiality, cryptography, and privilege management), time and state (deadlock, insecure temporary file), errors, code quality (memory leak, null deference), encapsulation (drawing strong boundaries around things to avoid, for example, system information leak), environment. To demonstrate how to classify security errors into the proposed kingdoms, the authors used information from the 19 Deadly Sins of Software Security (M. Howard, LeBlanc, and Viega 2005) and the OWASP (The Open Web Application Security Project) Top 10 Most Critical Web Application Vulnerabilities (OWASP 2012), as can be seen in Table 2-1.

Although security awareness has increased over the years, it is still very difficult to prevent vulnerabilities in the system originated from code defects/software errors. More specifically, studies have been conducted to estimate the number of average defects per thousand lines of code. (Hatton 2007) estimated one defect per 10 thousand executable code lines for mission and business critical software. The 2009 Coverty report uncovered one defect in every four thousand lines of Open source code (Coverity 2009). In 2011, Coverity analyzed over 37 million lines of code from 45 of the most active projects in Coverity Scan (Coverity 2011). The average defect density, or the number of defects per thousand lines of code, across the top 45 active open source projects in Coverity Scan is 0.45. This considerable reduction over the last two years in the defect density is due to the fact that open source software developers are more aware of the common defects that they made in their codes.

In the context of software security, (A. Ozment and Schechter 2006) found out that, for OpenBSD operating system, software defects that could compromise system security and that were introduced prior to the release of the initial version have an average lifetime of 2.6 years. In this study, the density of vulnerabilities per thousand lines of code ranged from 0 to 0.033. Although this seems to be a very low number for an open-source operating system, it is important to remark

Table 2-1. Mapping Security Errors to the Seven Pernicious Kingdoms and to the OWASP Top 10 Vulnerabilities *(Tsipenyuk, Chess, and McGraw 2005)*

KINGDOMNS	19 SINS	OWASP TOP 10
Input Validation and Representation	Buffer overflows, command injection, cross-site scripting, format string problems, integer range errors, SQL Injection	Buffer overflow, cross-site scripting flaws, injection flaws, unvalidated input
API Abuse	Trusting network address information	-
Security Features	Failing to protect network traffic, failing to protect and store network data, failing to use cryptographically strong random numbers, improper file access, improper use of SQL, use of weak password-based systems, unauthenticated key exchange	Broken access control, insecure storage
Time and State	Signal race conditions, use of "magic" URLS and hidden forms	Broken authentication and session management
Errors	Failure to handle errors	Improper error handling
Code Quality	Poor usability	Denial of service
Encapsulation	Information leakage	-
Environment	-	Insecure configuration management

that a single risky vulnerability, if successfully exploited, can totally compromise the security of the whole system.

In order to help developers to avoid software errors that can lead to serious vulnerabilities in software, (SANS Institute 2012), (MITRE Corp. 2012), and many top software security experts in the US and Europe proposed a list of the Top 25 Most Dangerous Software Errors (B. Martin et al. 2010). Some of these vulnerabilities (e.g., improper neutralization of special elements used in a SQL Command), if successfully exploited, allow attackers to completely take over the software, steal data, or prevent the software from working at all.

Initiatives to eliminate or reduce the number of software vulnerabilities present in a system are shown in literature in a variety of forms, which are:

- **Security training.** Designers and developers should have a security mindset (Schneier 2009). A developer that ignores the common software bugs that could lead to vulnerabilities will certainly write a vulnerable

code. Guidance on how to design a secure computer system, or how to write a secure code can be found at (Venkat Pothamsetty 2005) and (Thompson and Chase 2007).

- **Vulnerability scanners** are tools aimed at detecting vulnerabilities present in the system code. If the source code is available, there are tools that can search for common vulnerabilities patterns in the code. A review of automated tools for security is described in (McGraw 2008). Examples of code vulnerability finder tools are (HP Fortify 2012), (Ounce Labs 2012) and (Pixy 2012). If the source code is unavailable, there are tools such as penetration testing that look for vulnerabilities by analyzing the result of attacks. If an attack compromises system security, then vulnerability should exist.
- **System security patches** and **security configuration** are usually provided by vendors to fix known vulnerabilities of widely used software. It is an elementary precaution to keep a system patched and properly configured to avoid attacks exploiting known vulnerabilities. In general, default configuration is not the best approach to keep a server protected. Web servers with default configuration, for example, may have template web sites with Cross-site scripting vulnerabilities. More specifically, Apache Tomcat 4.1.0 through 4.1.39 contains an example of calendar application that contains XSS vulnerabilities (CVE-2009-0781). To help end-users with system security configuration, the US government created the National Checklist Program (NCP 2012), which provides detailed low level guidance on setting the security configuration of operating systems and applications.

2.2.2 Vulnerability and attack classification

There are many works published addressing specific domains and root causes of vulnerabilities, and there is a considerable diversity of vulnerability classifications. These classifications allow specialists to collect statistics, to perform trend analysis, to check the correlation with exploits, and to evaluate the effectiveness of countermeasures. In the context of our work, these classifications allows us to organize vulnerability information collected from the field and study the impact and exploitability of vulnerabilities under the same group.

An overview of vulnerability classifications can be found in (Meunier 2008) and they are summarized as follows:

- **Classification by software development lifecycle.** This aims to

categorize vulnerabilities according to the moment when they were introduced in the software lifecycle (feasibility study, requirements definition, design, implementation, integration and testing, and operations and maintenance). The downside of this approach is that a vulnerability can be introduced in multiple points of the development cycle and this classification is not always applicable. Examples of software development lifecycle classifications are found at (Dowd, McDonald, and Schuh 2006), (J. D. Howard and Meunier 2002), and (Piessens 2002).

- **Classification by genesis.** This approach aims to categorize vulnerabilities according to security flaws (the conditions or circumstances that lead to, for instance, denial of service, unauthorized disclosure, unauthorized modification of data, etc.). According to this approach the parts of software code or configuration that can cause a security compromise are used to classify the vulnerabilities. Examples of these classifications can be found at (S. Weber, Karger, and Paradkar 2005) and (Landwehr et al. 1994).
- **Classification by errors or mistakes.** This aims to categorize vulnerabilities according to human errors or mistakes, considering error cause (e.g., validation error, domain error), impact (e.g., execution of code, access target resource), and fix (missing entity). The limitation here resides in the fact that it not always possible to determine the exact point of a vulnerability. An example of this classification is found at (Du and Mathur 1998).
- **Classification by enabling attack scenario.** This approach aims to categorize vulnerabilities according to the attacks mounted to exploit vulnerabilities. For example, “cross-site scripting” (“XSS”) is an attack scenario, and “XSS vulnerabilities” are vulnerabilities enabling the injection of scripting code into web-based content. One concern here is that in certain cases a category is more directed to the consequence of the attack (e.g., denial of service) and is not helpful to guide users to identify the cause of the vulnerability. An example of this classification in the field of network protocol is present in (V. Pothamsetty and Akyol 2004).

As exemplified, there are several approaches to classify vulnerabilities and each one of them has inherent limitations. In our view, the most important factor of a vulnerability classification is the ability it provides to users and developers to quickly identify system weaknesses (errors, mistakes, etc.) that might result in a vulnerability. Using the knowledge about the origin of a vulnerability (e.g., an improper input validation, etc.), developers can make their code more secure and

avoid the causes leading to vulnerabilities in the future. This means that, in our opinion, the most useful and relevant classifications are those that classify vulnerabilities by genesis and by errors and mistakes. One important contribution in this direction (and that we apply in the present thesis to classify vulnerabilities) is the Common Weakness Enumeration (CWE), which is maintained by MITRE sponsored by the U.S. Department of Homeland Security (CWE 2012). CWE is a dictionary of software weakness type that has become widely used to classify the causes of software vulnerabilities. Each weakness has an identifier (CWE- ID), where ID is the identification of the weakness in the CWE platform. For each weakness, there is a web page containing several details about the software weakness, including the common consequences, the likelihood of exploit, detection methods, examples, and etc. Taking CWE-89 as example, it is possible to obtain the full description of the weakness, along with time of introduction (architecture and design, implementation, operation), applicable platforms, modes of introduction, common consequences to the security attributes (confidentiality, integrity, and availability), likelihood of exploit, enabling factors for exploitation, detection methods, and demonstrative examples. It is worth noting that there are other efforts to enumerate and classify vulnerabilities, such as the PLOVER (Preliminary List of Vulnerability examples of Researchers), (Christey 2005), (R. A. Martin and Barnum 2008), and (R. A. Martin, Christey, and Jarzombek 2005). However, none of them has the details and the community acceptance of CWE.

We can also find a diversity of attacks classifications in the literature. The following are some examples:

- **Network and computer attacks taxonomy.** (Hansman and Hunt 2005) proposed a taxonomy for network and computer attacks. The proposed taxonomy consists of four dimensions. The first dimension covers the attack vector (means by which the attack reaches the target) such as viruses, worms, network attacks, physical attacks. The second dimension classifies the target of the attacks such as computer, operating system, application, network, and so on. The vulnerabilities and the exploits attacks uses are classified in the third dimension. Any additional effect of the attack or malicious component that is installed in the attack (e.g., Trojan horse termed by the authors as payload) is addressed in the fourth dimension.
- **Attack-centric and defense-centric taxonomy.** Attack-centric taxonomies based on the objective of the attackers (e.g., steal information, bring the target down), while those based on defender goals (e.g., avoid information disclosure, keep the target available in the presence of

attacks) are defense-centric. (Killourhy, Maxion, and Tan 2004) proposed a defense-centric taxonomy based on attacks manifestations. This consists of building an experimental setup to emulate realistic attacks, observe the effects of these attacks, and then categorize the possible ways to defend the system against the attacks.

- **Signature classification.** This refers to taxonomies based on the pattern by which an attack is detected. The evidence category refers to the patterns that are left behind by an attacker (presence of certain files, permission on certain files) that can be evaluated by inspecting the state of the attacked system. The interval and duration of an attack may also be evaluated and are covered in the sequence signature category. An example of this classification is found at (Kumar 1995).
- **Attack-effect classification.** This refers to taxonomies that classify attacks based on the intended effect of the act (e.g., elevation of attack privileges). Examples of this classification are found at (J. D. Howard and Longstaff 1998; Lindqvist and Jonsson 1997; Ranum 1997; D. J. Weber 1998).
- **Attack threat classification.** This refers to classifications that categorize the potential attacks that a system may be targeted by. (M. Howard and LeBlanc 2002) proposed a classification scheme named as STRIDE that characterizes known threats in accordance with the motivation of the attacker, for example: spoofing (e.g., stealing of user identity on systems), tampering (modification of system data without authorization), repudiation (e.g., denial of access of authorized users to a given system), information disclosure (reading of private content without proper authorization), denial of service, and elevation of privileges. Note that this kind of classification may be part of a treat modeling effort aimed at mapping possible threats against a system, such as the Trike methodology described at (Saitta, Larcom, and Eddington 2005). Although useful, these approaches do not guarantee that all possible threats are identified and categorized, since the threat analysis relies on the experience of the security specialist to map potential attacks and threat categories.

In the context of our work, a useful attack classification is the one that allows users to understand the effect of an attack to the system, as in the case of attack-effect classification. Although the knowledge of the attack vector or signature of an attack may be useful, our main concern resides in identifying the most harmful attacks. An important contribution in this regard is the Common Attack Pattern Enumeration and Classification (CAPEC 2014), also

maintained by MITRE and sponsored by the U.S. Department of Homeland Security. CAPEC is an attack dictionary that contains comprehensive information about the nature, characteristic and effects of attacks. Examples of attack information provided by CAPEC are attack description and category, execution flow, exploit techniques, solutions and mitigations, and also typical severity and likelihood of exploit. In our work, we take advantage of attack implementation techniques provided by CAPEC to guide users to build the experimental part of our security benchmark methodology.

2.2.3 Vulnerability and attack repositories

The MITRE Corporation (MITRE Corp. 2012) along with the National Cyber Security Division of the U.S. Department of Homeland Security (NCSD 2012) has conducted an important initiative to identify and enumerate computer system vulnerabilities: the Common Vulnerability and Exposures (CVE 2014). The goal is to provide a unique way to identify and characterize vulnerabilities and information regarding the steps that should be followed by software owners to fix and patch the affected system version and or configuration.

CVE has become a standard on information security concerning the vulnerability names. For each vulnerability reported to CVE, it is generated a CVE Identifier (CVE-YEAR-ID), where YEAR is the year of vulnerability report and ID is the identification of the vulnerability for a given year. CVE Identifiers enable security practitioners and researchers to access the full characterization of vulnerabilities, including its description, affected system, versions, and configuration, and the solutions or available workarounds. The creation of a CVE Identifier involves the CVE Numbering Authority (CNA), represented by the MITRE Corporation along with software vendors (e.g., Apple Inc., Oracle, Microsoft, IBM Corporation, Google), third-party contributors (CERT/CCCC, JPCERT/CC), and security researchers.

CVE-2012-0671 is a CVE vulnerability reported in 2012 with the ID 0671 and refers to an Apple QuickTime vulnerability (affecting versions before 7.7.2) that allows remote attackers to execute arbitrary code and to cause a denial of service. The National Vulnerability Database (NVD) makes available the full description of each CVE Identifier at http://web.nvd.nist.gov. One of the first recorded vulnerabilities is identified as CVE-1999-0095. That particular vulnerability consists on having the debug command in Sendmail enabled, allowing attackers to execute commands as root. To date, more than 65000 vulnerabilities were already recorded.

The Open Source Vulnerability Database (OSVDB) is another example of

vulnerability repository that identifies vulnerabilities using CVE naming system. The difference between OSVDB and CVE is that OSVDB is open to anyone that wants to report vulnerabilities and CVE is not. For the sake of comparison, OSVDB currently covers 120,980 vulnerabilities, spanning 198,976 products and submitted by 4735 users (measurement collected in September, 2015).

One important platform that has been used to keep and share vulnerabilities information is vulnerability databases. They contain detailed information regarding vulnerability reported by users or fabricants, including vulnerability description, the list of affected systems and versions, available patch and methods, and the level of impact and exploitability. Vulnerability databases initiatives have been sponsored by governments (such as the National Vulnerability Database), enterprises (such as Secunia) and specialists interested in studying the trend of reported vulnerabilities in systems (such as the Open Source Vulnerability Database). Some examples of vulnerabilities databases are presented in Table 2-2.

Exploit databases and tools are also available on the web. (MilW0rm 2010), which was shutdown in 2010 due to the lack of resources to maintain the initiative, was a very popular repository of exploits, providing examples of attacks and making them available for anyone willing to use them. Milw0rm can still be reached at web.arquive.org. (Metasploit 2015) hosts one of the largest databases on exploits, including hundreds of remote exploits, auxiliary modules, and payloads (“Metasploit Auxiliary Module & Exploit Database”) and it also provides a penetration testing tool that helps users to better understand how attacks are executed.

2.2.4 Vulnerabilities and attack statistics

Vulnerability and attack trends have gained ground in recent years. The main benefit of the analysis vulnerability and attack trends is that it enables designers and developers to focus on the most representative vulnerability types and helps security firms to develop more effective measures to counter these attacks.

CVE has been tracking the software errors that lead to publicly reported vulnerabilities, and it periodically reports on the trends on a limited scale (Christey and Martin 2007). In 2007, CVE published a report indicating that vulnerabilities found in web application rose sharply; buffer overflow vulnerabilities was the number one issue reported by operating system (OS) vendor advisors, followed by integer overflow.

IBM Corporation issues in a regular basis a security report about the trend of security incidents: the IBM X-Force Trend and Risk Report (IBM 2014). The

Table 2-2. Examples of Vulnerabilities Databases

Type	Vulnerability Database	URL
Government	US CERT Vulnerability Notes Database	http://www.kb.cert.org/vuls
	Common Vulnerability and Exposures (CVE)	http://cve.mitre.org/
Vendors	BugTraq	http://www.securityfocus.com
	SANS	http://www.sans.orgv
	X-Force	http://xforce.iss.net/
	Security tracker	http://securitytracker.com/
	Symantec's Security Response Database	http://www.symantec.com/avcenter/
	AtStake	http://www.atstake.com
	Secunia	http://secunia.com/
Open-Source	OWSAP	www.owasp.org/
	Cassandra	https://cassandra.cerias.purdue.edu/
	The Open Source Vulnerability Database (OSVDB)	http://osvdb.org/

IBM X-Force Research and Development team has the role of analyzing trends in attacks behaviors and they claim that 2011 was the year of the security breaches (IBM X-Force 2012). Law enforcements, governments, social network communities, retail, entertainment, banks, non-profits, the Fortune 500, and even security companies were attacked. IBM-XForce team obtains security information from IBM Managed Security Services (MSS). MSS monitors tens of billions of events per day in more than 130 countries, working continuously 24 hour a day, providing a unique understanding of the cyber threat landscape. The top high-volume of signatures (vulnerabilities) collected in 2011 by MSS tool is as follows: 1) SQL Injection, 2) HTTP Suspicious Unknown content, 3) SQL SSRP Slammer Worm, 4) SNMP Crack, 5) HTTP Get Dot Dot Data, 6) Cross-site Scripting, 7) SSH Brute Force, 8) HTTP Unix Passwords, 9) Shell Command Injection, and 10) Proxy Bounce Deep. A detailed analysis of this result is available in (IBM X-Force 2012). The groups Anonymous and Lulzsec were major players in SQL injection tactics and continue to improve their skills with new injection attack vectors. Additionally, there are automated SQL injection attacks like LizaMoon that scan the Internet for vulnerable hosts.

There has also been a significant increase in vulnerability disclosures in recent years. **Error! Not a valid bookmark self-reference.** shows the evolution of vulnerability disclosures from 1996 to 2011 as reported by IBM X-Force Team, with nearly 9000 vulnerability disclosures in 2010 and 7000 in 2011. According to the data provided by the IBM X-Force 2011 Report, The web applications is a type of application where vulnerabilities are prone to exist: 41% of all vulnerabilities disclosed in 2011 were found in web applications, and in 2010, 49% of all vulnerabilities disclosures were found in web applications. SAN Top Cyber Security Risk Reports also confirms this result (SANS Trends 2009). According to SAN report, the number of vulnerabilities being discovered in application-level software is far greater than the number of vulnerabilities discovered in operating systems.

In 2011, of all the vulnerabilities for which patches that were issued to remove vulnerabilities, 91% were patched on the very same day of the public it disclosures (IBM X-Force 2012). This demonstrates that software vendors are treating security as a top priority, taking immediate action to provide users the proper means to defend their systems against vulnerabilities. However, and still according to X-Force, there were 29 cases during 2011 where it took more than a week for a major software vendor to fix a publicly disclosed vulnerability that had an exploit. This particular case is the worst scenario possible for users, since an attacker is aware of the vulnerability, has the means to mount the attack and there

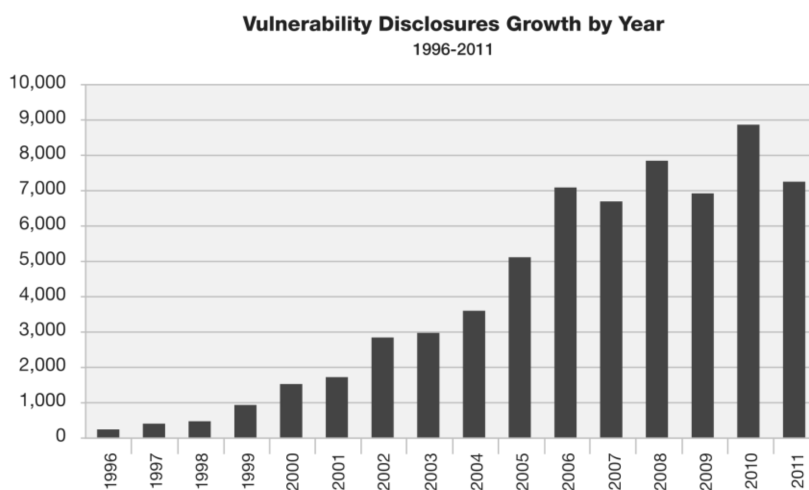

Figure 2-2. IBM X-Force Report – Vulnerability Disclosure Growth by Year

is no fix available to make a potential attack ineffective.

A report on security attacks and incidents in 2009 (Titterington 2010) described the following findings from major security research firms:

- The Security Intelligence Operations Team of Cisco Company (a global company in the business of network equipment) reported that security incidents rose 57% in 2009.
- RSA (the Security Division of EMC, a global technology company) reported a 50% increase in phishing attacks.
- The Computer Security Institute reported that during 2008 50% of organizations that responded to the CSI survey were infected by malware. There were similar increase rates in other types of attack: for example, in 2012, 19.5% institutions suffered financial fraud, compared to 12% in 2008. This suggests that there is an increase in the attacks and security concerns are increasingly pertinent.
- (Verizon 2012) reported that the number of data records breached in incidents it investigated in 2009 exceeded the total for the four previous years.

As shown above, security incidents have increased in recent years. As the number of successful attacks rises, it is necessary to develop and deploy better methods to augment the security of systems and counter these attacks. To this end, an important step is the development of means to measure the security level of systems, an important contribution of the security benchmark methodology proposed in this thesis.

2.2.5 Vulnerability risk assessment

Several initiatives have emerged with the purpose of assessing the risk of vulnerabilities. Microsoft defined a proprietary scoring system reflecting the difficulty of exploitation and the overall impact of vulnerabilities (Microsoft SecBulletin 2012). This scoring system consists in rating the vulnerability severity (Critical, Important, Moderate, and Low), and on the Microsoft Exploitability Index, which indicates the likelihood of a vulnerability to be exploited in the future. A similar approach is proposed by the SANS Institute (an organization that provides information security training and security certification) with the @RISK method (SANS @Risk 2012), which consists in ranking vulnerabilities by their criticality level (Critical, High, Moderate, Low). The problem of these approaches is that they lack a clear and detailed method on how the impact and exploitability of each discovered vulnerability is assigned.

The CVSS is an open framework aimed at standardizing the evaluation of vulnerability risk, mitigating the problem of having different impact scores for the same vulnerability (Mell, Scarfone, and Romanosky 2007). It is a vulnerability risk assessment approach that has been widely adopted by enterprises and that we use in the benchmark metric portion of our security benchmark methodology. CVSS is sponsored by the Forum of Incident Response and Security Teams (FIRST) and its popularity can easily be confirmed by browsing popular vulnerabilities databases. The importance of CVSS to our security benchmark methodology is that we use CVSS approach to estimate the risk of vulnerabilities in our security benchmark metric.

CVSS is composed of three metrics groups aimed at providing the definition and communication of the fundamental characteristics of vulnerabilities: base, temporal, and environmental. Each group of metrics (CVSS sub-equations) can vary from 0 (minimum) to 10 (maximum criticality of the reported vulnerability). A more specific group definition is as follows:

- **Base.** This refers to the vulnerability characteristics that are constant over time and across user environments. For example, the impact of a vulnerability to the security attributes of confidentiality, integrity and availability.
- **Temporal.** This refers to the vulnerability characteristics that change over time but not among user's environments - such as remediation level and report confidence.
- **Environmental.** This refers to the vulnerability characteristics that are relevant and unique to a particular user's environment. For example, the potential for loss of life or physical assets or the importance of the vulnerable component to the business.

CVSS framework has been improved over the time and there is a board responsible for receiving feedbacks from security community and adjusting and calibrating framework requirements, metric attributes and equations. From 2007 to 2015, CVSS version 2 was the official version and has been widely adopted by industry and academia. Most of the vulnerability scores provided in the US National Vulnerability Database, for example, have been reported in accordance with Version 2. However, in June 2015, CVSS Version 3 was announced, as a result of the work performed by the CVSS Special Group that started in 2012. This new version contains score adjustment, better description of framework criteria, updated vulnerability vector string, and etc. According to the authors, CVSS Version 3 explicitly states at which point of an attack the score should be

computed, reducing the variations in impact metrics between scorers. The fact that Version 3 is a very recent proposal, and also considering that the new version is in the process of being adopted by industry and academia, led us to keep our security measurements based on CVSS Version 2. We do recognize the improvements that were made on Version 3 and we do expect to provide in the future a new version of our security benchmark reflecting the changes that were recently proposed.

Within the base metric group of CVSS Version 2 there are 6 metrics covering two aspects: access and impact. The first includes the access vector (which indicates how the vulnerability is exploited), access complexity, and the authentication metrics that capture how the vulnerability is accessed and whether or not extra conditions are required to exploit it. The impact is measured by the three impact metrics (confidentiality impact, integrity impact and availability impact) measure how a vulnerability, if exploited, will directly affect the system. The impact is defined as the degree of loss of confidentiality, integrity, and availability independently from each other (e.g., a vulnerability exploit may cause a partial loss of integrity and availability, but no loss of confidentiality). These metrics and the equation to measure them are fully described in (Mell, Scarfone, and Romanosky 2007). The details on the attributes of the base metric group of CVSS are important as we use them to compute the benchmark metric of our security benchmark methodology. The attributes are:

- **Access Vector (AV).** This metric reflects how the vulnerability is exploited. The possible values for this metric are *Local*, meaning that the attacker needs either physical access to the vulnerable system or a local (shell) account, *Adjacent Network*, which means that the attacker needs access to either the broadcast or to the collision domain of the vulnerable software, and *Network*, meaning that the vulnerable software is bound to the network stack and the attacker does not require local network access or local access. Each one of these values has an associated CVSS score, defined as 0.395 (Local), 0.646 (Adjacent Network), and 1 (Network). The more remote an attacker can be from the target and still be able to attack it, the greater the vulnerability score: a vulnerability that is exploitable remotely (Network) will obtain the highest score in the access vector metric.
- **Access Complexity (AC).** This metric captures the complexity of the attack required to exploit the vulnerability once an attacker has gained access to the target system. The possible values for this metric are *High*: special conditions (such as a vulnerable configuration) are required but

they can hardly occur in practice, *Medium*: the required access conditions are somewhat specialized but are not commonly configured, (e.g., a non-default configuration), and *Low*: specialized access conditions do not exist, or if they exist they are ubiquitous (e.g., a default configuration). Each one of these values has an associated CVSS score, defined as 0.35 (High), 0.61 (Medium), and 0.71 (Low). The lower the required complexity, the higher the vulnerability score.

- **Authentication (Au).** This metric focus on the number of times an attacker must authenticate to a target in order to exploit a vulnerability. The possible values for this metric are *Multiple*, meaning that the attacker needs to authenticate two or more times, even if the same credentials are used each time), *Single*, meaning that only one instance of authentication is required to access and exploit the vulnerability), and *None*, which means that authentication is not required at all for the attacker to access and exploit the vulnerability). Each one of these values has an associated CVSS score, defined as 0.45 (Multiple), 0.56 (Single), and 0.704 (None). The fewer authentication instances that are required, the higher the vulnerability score.
- **Confidentiality Impact (C).** This metric reflects the impact on confidentiality of an exploited vulnerability. Confidentiality refers to limiting access and disclosure of information to authorized users, which means preventing access and disclosure to unauthorized users. The possible values for this metric are None: there is no impact on confidentiality, Partial: there is considerable information disclosure, and Complete: there is total information disclosure. The associated scores are 0 (None), 0.275 (Partial), and 0.66 (Complete) - the higher the confidentiality impact, the higher the vulnerability score.
- **Integrity Impact (I).** This metric focus on the impact an exploited vulnerability to integrity defined as the trustworthiness and guaranteed veracity of information. The possible values and score for this metric are None (0): there is no impact to the integrity of the system, Partial (0.275): it is possible to modify some information (e.g., files), but the attacker does not have control over what can be modified, or the scope of what the attacker can affect is limited, and Complete (0.66): there is a total compromise of system integrity - the attacker can modify any information on the target system). The higher the integrity impact, the higher the vulnerability score.
- **Availability Impact (A).** This metric captures the impact to availability

(accessibility of information resources) of an exploited vulnerability. The possible values and scores for this metric are None (0): there is no impact to the availability of the system, Partial (0.275): the availability of the system or its resources is reduced, but not completely, and Complete (0.66): there is a total unavailability of the affected resource - the attacker can render the resource completely unavailable). The higher the availability impact, the higher the vulnerability score.

The CVSS metrics can be used to assess the risk of software vulnerabilities. Because these metrics have clear and well-defined meanings and values, they can be helpful to obtain a method approaching a standard. For example, the IBM X-Force team has used CVSS to report the risk level of vulnerabilities, using the following scale: Critical (CVSS score is equal 10), High (CVSS score ranges from 7 to 9.9), Medium (4.0-6.9), and Low (0.0-3.9). Figure 2-3 presents the result of risk analysis for the vulnerabilities covered by X-Force Team during the year of 2011. It is worth noting that 2% of the disclosed vulnerabilities during 2011 were critical, while the major part had a medium risk (69%). This is a very important finding since it helps users and developers to concentrate their security efforts on the most critical vulnerabilities.

2.3 SYSTEM BENCHMARING

In general terms, the goal of benchmarking is to measure system attributes for

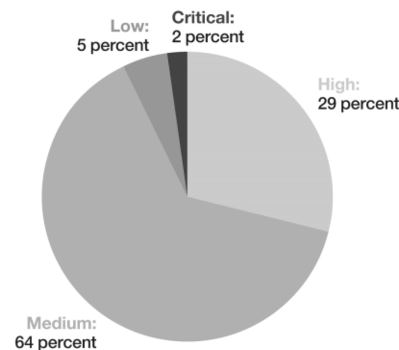

Figure 2-3. CVSS categorization of vulnerabilities captured by X-Force Team during the year of 2011

comparative purposes. Computer users adopt benchmarks to identify the *best system* to *select* considering a given system *attribute*. This means that users can take advantage of benchmarks either to select the fastest, the most robust, the most dependable, or the most secure among alternative systems.

Benchmarks have been proposed in the fields of performance, robustness, dependability, and, more recently, security. Regardless of the application domain, most of the existing benchmarks share common elements and have to comply with certain properties to be accepted by the community and users. In this section, we describe these elements, with a particular focus on dependability benchmarking, as those benchmarks have several relevant aspects in common with our methodology.

The benchmark metric is a key element of system benchmarks, indicating the level of a given attribute – which is collected and measured during the benchmark execution. Performance benchmark metrics, for example, indicate the level of speed that a system performs a set of tasks, while dependability benchmark metrics indicate the ability of a system to avoid services failures that are more frequent and more severe than acceptable. Benchmark users can take better decisions based on the measurements provided by benchmarks, such as increasing the capacity of the system to have better performance scores, or deploying more fault tolerant mechanisms to make the system more dependable.

There are certain characteristics that are inherent to system benchmarks, without which they cannot be considered a benchmark. These characteristics are important to our work, as we aim to propose a methodology that can be used to define benchmarks, as such, should comply with those characteristics. These characteristics are described as follows:

- **Standard nature.** A benchmark should be repeatable, which means that it should be defined as a standard procedure that will ensure the design, implementation, and execution of the benchmark in a uniform and standardized way (Koopman et al. 1997).
- **Open standard.** All rules and procedures to execute the benchmark should be independent from proprietary methods and tools. This is an important characteristic to facilitate the acceptance and popularity of a benchmark among users.
- **Comparison-oriented.** A benchmark must enable the comparison of functionally equivalent systems (Koopman and Madeira 1999), evaluating the relative strengths and weaknesses of systems

- **User-centric-view.** As a benchmark usually has different kinds of users, benchmark metrics should reflect the different needs of these users. For example, end-users of dependability benchmarks can be interested in system availability measurements, while developers may be interested in errors propagation measurements (Madeira and Koopman 2001).
- **Integrated nature.** The benchmark should provide a consolidated view of the attribute that is being benchmarked. In the security context, all relevant classes of vulnerabilities should be covered.
- **Representativeness.** The elements and components that integrate the benchmark must reflect what is represented in the real world. The benchmark must represent the operational profiles of a class of applications and/or a community of users.
- **Agreement.** A benchmark should be defined based on the agreement between the computer industry and user communities regarding the system under benchmark, the metrics, the procedures and rules to obtain these metrics and so on (DBench 2004).

A noteworthy question is how to provide a benchmark. Benchmarks can be provided in the form of executable programs (e.g., SPEC benchmarks (SPEC 1988)), or they can be provided through documents that specify what must be implemented (e.g., TPC benchmarks (TPC 1988)). The advantage of the first is that it is readily available to the user. The advantage of the second approach is that it is more transparent to the benchmark user and it does not require a specific support from the benchmark proposers.

This section provides examples of classical benchmarks in robustness, performance, dependability, and security fields. Note that dependability benchmarks are presented in more detail than robustness and performance benchmarks due to their relevance to the work presented in this thesis.

2.3.1 Performance Benchmarks

The goal of performance benchmarks is to measure and compare the performance of the targets using well-defined workload and measures. Organizations of different domains have been involved in benchmark consortiums with the intent of reaching an agreement towards measures, procedures, and rules, to benchmark the performance of systems in a standardized way (TPC 1988; SPEC 1988; Cybenko et al. 1990; Berry et al. 1989; Van Der Steen 1991). In this section, we focus on two relevant examples of these consortiums.

The Standard Performance Evaluation Corporation (SPEC) is a non-profit

corporation formed to establish, maintain, and endorse a standardized set of relevant benchmarks that can be applied to the newest generation of high-performance computers (SPEC 1988). SPEC benchmarks are organized in several groups, which are: CPU, Graphics and Workstation Performance, High Performance Computing, Java Client/Server, Mail Servers, Network File System, Power, Session Initiation Protocol (SIP), Virtualization, and Web Servers. One characteristic of these benchmarks is that they are provided in the form of tools or source codes. SPEC CPU2006 stresses system's process, memory, and subsystem and is provided as source code, which means that users need to compile the source code in order to have the executable binaries that will allow them to run the benchmark. SPECWeb evaluates web server performance using a workload that simulates the execution of an e-commerce system. Another important aspect is that each one of these SPEC benchmarks has its own set of benchmark measures. To benchmark the performance of web servers, SPEC Web, for example, measures the number of simultaneous confirming connections (SPEC metric), the number of operations per second (THR metric), and the average time in milliseconds that the operations requested by the client take to complete (RTM metric). The notion of workload, measures and tools is used in this thesis in the application of our benchmark methodology for web serving system.

The Transaction Processing Performance Council (TPC) defines transaction processing and database benchmarks and delivers trusted results to industry (TPC 1988). The benchmarks proposed by TPC are supported by large, competing computer companies such as Oracle, Sun Microsystems, IBM, and Microsoft (TPC members). The results of TPC are used worldwide, especially as it can indicate which product is the best for a given application environment. Examples of TPC benchmarks are:

- **TPC-C** is an on-line transaction processing benchmark that simulates users executing transactions against a database. The benchmark measure is the number of transactions per minute (tpmC).
- **TPC-E** simulates the on-line transaction processing workload of a brokerage firm. The TPC-E metric is given in transactions per second (tps). It refers to the number of Trade-Result transactions the server can sustain over a period of time.
- **TPC-H** is a decision support benchmark. This benchmark illustrates decision support systems that examine large volumes of data, run complex queries, and give answers to critical business questions. The TPC-H metric reflects multiple aspects of the capability of the system to process queries.

- **TPC-App** benchmarks the performance capabilities of application server systems and web services. The main TPC-App metric is the throughput of the application server measured in Web Services Interactions per Second (SIPS).
- **TPC-W** (which became obsolete in 2005) is a transactional web benchmark. The performance metric reported by TPC-W is the number of web interactions processed per second. Multiple web interactions are used to simulate a retail store activity, and each interaction is subject to a response time constraint.

SPEC and TPC performance benchmarks have key elements that we adapt to the benchmark methodology proposed in this thesis: a *workload* that consists in a set of tasks that stress the performance capabilities of targeted systems, a set of non-overlapping *metrics* collected during the execution of the workload that indicates the performance level of the system, and *procedure and rules* that guide users on how to execute the benchmark and ensure the repeatability of the benchmark execution.

2.3.2 Robustness Assessment and Benchmarks

The goal of robustness benchmarks is to characterize the behavior of the system under benchmarking in the presence of unexpected conditions (such as erroneous inputs at the system interface). Several works were proposed to evaluate the robustness of software systems such as Linux and Windows utilities (Koopman et al. 1997; Siewiorek et al. 1993). Additionally, robustness tools have been widely used both by academia and computer industry and have been applied in diverse classes of systems.

Ballista (Koopman et al. 1997) is a tool to test the robustness of software components through the combination of software testing and fault injection approaches. It uses diverse combinations of input values as parameters of system calls to assess and compare the robustness of different operating systems (Linux and Windows versions). The robustness of the target system is evaluated in accordance with five failure modes: catastrophic (the application causes a complete system crash that requires the reboot of the operating system), restart (the application hangs and needs to be restarted), abort (abnormal termination of a task or a process as the result of, for example, a segmentation fault), silent (no error code is returned, but one should have been returned), and hindering (error code returned is not correct).

MAFALDA (Fabre et al. 1999) is a tool that allows the characterization of the

behavior of microkernels using two forms of fault injection: (i) injection on the input parameters and (ii) injection on the state of the microkernel. This is a fault injection tool that tests the interface robustness of the system, allowing the assessment of the error handling behavior of the system, as well as the identification of microkernel deficiencies to be fixed.

One class of system that has been targeted in robustness testing research is web services. (M. Vieira, Laranjeiro, and Madeira 2007) proposed a robustness benchmarking for web services based on fault injection technique. The approach consists of injecting invalid web services call parameters aimed at disclosing both programming and design program. (Laranjeiro, Canelas, and Vieira 2008) also presented a user-friendly web-tool to test the robustness of web services in the presence of unexpected inputs. Their results have been widely accepted by the community and have been used in more recent works to evaluate the robustness of different services oriented applications such as Java Message Services systems (Laranjeiro, Vieira, and Madeira 2008).

2.3.3 Dependability Benchmarks

The Dependability Benchmarking Project (DBench) was established in 2001 by the European Commission as a consortium of several universities and organizations to the definition of a dependability-benchmarking framework targeting Off-the-Shelf components (OTS). The goal was to provide means to assess the dependability attributes of systems for comparative purposes. This section addresses dependability benchmarking following the approach found in classical dependability benchmarks proposed in DBench: benchmark dimensions, benchmark components, techniques, benchmark validation, and examples.

The main contribution of dependability benchmarks over performance benchmarks is that the former is focused on reliability aspects. The focus is not only on the system speed to perform a set of tasks, but also on the characterization of the system behavior in the presence of faults. Figure 2-4 shows the relationship between performance and dependability benchmark, where it is clear the presence of the faultload component, used to inject faults in the system under benchmark while dependability measurements are collected. This fault injection notion and its importance to dependability benchmarks are properly discussed in subsection 2.3.3.3.

2.3.3.1 Dependability Benchmark Dimensions

In the final report provided by the DBench project, we can find the three dimensions that are needed to define a dependability benchmark.

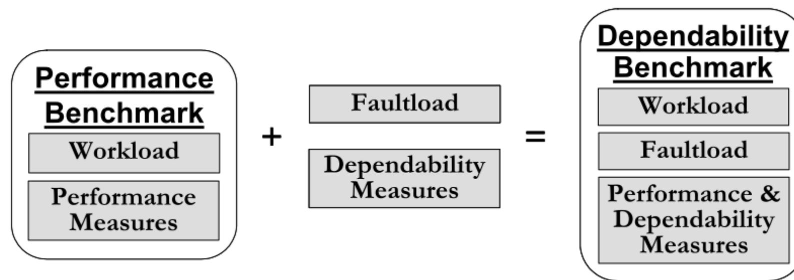

Figure 2-4. Main components of a dependability benchmark (Figure taken from (Marco Vieira 2005))

These dimensions were firstly discussed in (Madeira et al. 2002) and are described as follows:

- The **categorization** dimension allows us to organize the dependability benchmark space into well-defined categories, describing benchmark target and the benchmarking context (the application area of the benchmark).
- The **measure** dimension specifies the dependability benchmarking measures, defined in accordance with the benchmark target and the benchmark context.
- The **experimentation** dimension includes all aspects related to the experimentation of the benchmark target to obtain the measurements needed to compute the benchmark metric. The benchmark components related with experimentation are described in the next subsection and the elements needed to support the benchmark target and benchmark execution are described as follows:
  - *System under benchmark (SUB)*. This system hosts the components that interact with the Benchmark Target (BT) (e.g., operating system, hardware platform) and with the workload. Figure 2-5 shows a representation of the system under benchmark taken from (Durães, Vieira, and Madeira 2004).
  - *Benchmark Management System.* Refers to the set of components needed to orchestrate the benchmark run and to collect dependability measurements.

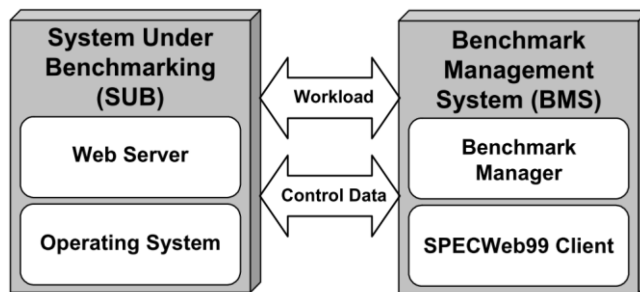

Figure 2-5. System Under Benchmark and Benchmark Management System Interaction (Durães, Vieira, and Madeira 2004).

2.3.3.2 Dependability Benchmark Components

The components of a dependability benchmark are described as follows:

- **Measures.** These characterize the performance and dependability of the system under benchmark during the execution of the workload and of the faultoad. The dependability measures are defined in accordance with the benchmark target and context and examples are provided later on. Once collected and properly measured, these are used to indicate the dependability level of the system and to help users to identify the most dependable among the benchmarked systems.
- **Workload.** The workload represents a set of tasks that are submitted to the target system during the benchmarking execution, representing the typical work that the system executes. In the DBench project, authors make a distinction between two categories of workloads: background (aimed at simulating a system activity profile) and foreground (aimed at analyzing the impact of fault on the service). An important aspect of workloads is that they should be representative. To be realistic, a workload must simulate the tasks that the system actually executes. If the system under benchmark is an OLTP system, this means that database transactions in a client-server environment is expected, as illustrated in (Marco Vieira 2005). If it is a web server workload, it should simulate multiple users requesting data to the web server and its hosted applications, as detailed in (Durães, Vieira, and Madeira 2004). In the dependability field, a common practice is the adoption of existing workloads from performance benchmarks such as TCP-C for OLTP dependability benchmarks and SPECWeb. This is done as performance benchmarks are well established, widely adopted and simulate in a realistic way the work executed by the system under benchmark.

- **Faultload.** This component is aimed at injecting faults in the target system, being an important tool to collect dependability measures (Durães, Vieira, and Madeira 2004; Vieira and Madeira 2003). This is the most critical component of a dependability benchmark, since the set of faults should be repeatable, portable and representative. It is worth pointing out that the deliberate activation of faults is essential in a dependability benchmark as it allows us to observe the tolerance of the system to survive to a faulty scenario. The injection of faults is necessary to speed up their activation in the system, as it would be very time-consuming to wait for the natural activation of faults in a system. The faultload thus represents a key component that injects faults that are activated by the execution of the workload component.
- **Procedures and rules**. These refers to the formal specification of all procedures and rules to conduct the benchmark, including the description of the means necessary to build and use the tools involved, ho to collect the measures.

2.3.3.3 Dependability Benchmarking Techniques

One of the most common techniques that has been used in the context of dependability benchmarking is fault injection. Fault injection is the deliberate insertion of faults into a software or hardware to determine its response (definition adapted from (J. A. Clark and Pradhan 1995)). In the dependability benchmarking context, fault injection is mostly applied to accelarate the occurrence of failures and is a key part of the faultload component. In this case, the benchmark management system can assess if the injection of faults compromises system availability or reliability.

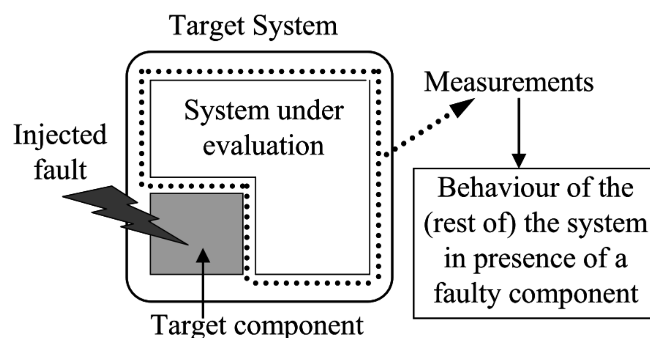

Figure 2-6. Software fault injection and system observation (Duraes and Madeira 2006)

As software faults are a common cause of computer failures (as investigated by (Gray 1990),(Sullivan and Chillarege 1992),(I. Lee and Iyer 1995), and (Kalyanakrishnam, Kalbarczyk, and Iyer 1999)) software fault injection is extremelly useful to study the behaviour of the system in the presence of faults, to evaluate the mechanisms of error handling, or to evaluate fault tolerance tools. A high-level representation of software fault injection in a dependability context is presented in Figure 2-6. In this case, while the faults are injected the benchmarkig management system collects the measurements that are needed to determine the effects of the fault injection to the target system. One important notion here in this representation is that the benchmark target should be different from the component under fault (Durães, Vieira, and Madeira 2004). This is justified by the fact that it is not realistic to inject software faults in the same component that is being evaluated as the faults would change the component and the conclusions would be unfair. What is realistic is to inject faults in a system/component closely related to the System Under Evaluation and observe the behavior of the component under observation when dealing with a faulty behavior of another system or component with which it interacts.

In the context of our work, software faul injection is used to inject faults in the source code (vulnerabilities) of software-based systems components that will lead to a security breach. This is needed to analyze the behaviour of the benchmark target system while attacks are exploiting the vulnerabilities injected in a component outside the benchmark target. In this sense, there are two ways of injecting software faults that are important to highlight in the context of our work: compile-time injection and runtime injection. In the former type, the program instructions must be modified before the program image is loaded and executed. In this case, faults are injected into the source code or assembly code of the target component to emulate the effect of hardware, software, and transient faults. In the later type, a mechanism (such as time-out, expection, or code insertion during runtime) is needed to trigger fault injection. One of the observations of (Hsueh, Tsai, and Iyer 1997) about software fault injection is that the software may disturb the workload running on the target system, which means that a careful design is needed when using this technique.

A technique for the injection of software faults that has been used in dependability benchmark works is the G-SWIFT (Generic Software Fault Injection Technique), which consists of finding key programming structures at the machine-code level where high-level software faults can be emulated (J. Durães and Madeira 2004). The benefit of this approach is the injection of the software fault even if the source-code of the target system is not available. This advantage

is important when the source code is not available, as is the case when using third-party components or simply off-the-shelf components.

One important step prior to the injection of software faults is the conduction of field studies to identify the most representative ones. This is important to assure the realism of the approach as the faults to be injected are the ones commonly found in real systems. (J. A. Durães and Madeira 2006) collected a large set of real software faults and analyzed the exact nature of these faults and their occurrence distribution. The authors used a three-step approach to classify and analyze 668 software faults from open-source programs. Since we take advantage of this approach in the experimental part of our security benchmark methodology (to inject software faults that lead to vulnerabilities in a system component), it is important to describe it in details, which is as follows:

1. Classification of the faults according to Orthogonal Defect Classification (Chillarege et al. 1992). In ODC, a software fault is characterized by the change in the code that is necessary to correct it (assignment, checking, interface, algorithm, and function).
2. Characterization of each software fault by one (or more) programming language constructs, which can be a statement, an expression, or a function. The three possible categories in this characterization are: *missing construct, wrong construct, and superfluous construct*.
3. Classsification of faults in specific construct-related fault types, for example: MVIV (Missing variable initialization using a value), WLEC (Wrong logical expression used as branch condition), EVAV (Extraneous variable assignment using another variable). In (J. A. Durães and Madeira 2006), the three most representative software fault types are MIFS (Missing if construct plus statements), MLAC (Missing *AND sub-expr* in expression used as branch condition), and MFC (Missing Function Call).

There are several works in literature that approach the topic of fault injection. Examples of fault injection techniques are presented in (Hsueh, Tsai, and Iyer 1997). Examples of fault injection tools are Ferrari (which uses software traps to inject CPU, memory, and bus faults) (Kanawati, Kanawati, and Abraham 1992), FTAPE (which injects faults into user-accessible registers in CPU modules, memory locations, and disk subsystem)(Tsai, Iyer, and Jewitt 1996), Doctor (which inject CPU, memory, and network communication faults)(Han, Shin, and Rosenberg 1995), and Xception (which uses a processor built-in hardware exception triggers to trigger fault injection)(Carreira, Madeira, and Silva 1998), and Jaca (which injects faults in Java byte code during runtime by corrupting

attribute values, methods parameters, or return values)(Martins, Rubira, and Leme 2002).

2.3.3.4 Dependability Benchmarking Validation

The validation of the benchmark is an important step when defining a new benchmark. A typical validation approach is experimental, during which the benchmark is applied in several case studies and the following properties are evaluated:

- **Representativeness.** The benchmark elements (measures, workload, etc.) should provide a representative contribution to the overall benchmark representativeness. This means that measures should be realistic, actually representing the benchmark context. The workload should correspond to the actual workload of the operational scenarios. The faultload should contain a set of the most common faults found in the system under benchmarking.
- **Repeatability and reproducibility.** To be repeatable, a benchmark should provide equivalent results when the benchmark is run in the same environment. Also, other users should be able to implement the benchmark following benchmark procedures and rules and to reach equivalent results. In other words, independent teams executing the same benchmark over the same systems should arrive at similar results.
- **Portability.** In the context of benchmarks, portability applies to all components of the benchmark. This means that all specifications (e.g., workload, measures, etc.) must be portable to any system within the application domain. This also means that for a benchmark based on documentation, at most, only the tools must be re-implemented (benchmarks that provide tools might not provide the necessary information to re-implement them for different platforms).
- **Non-intrusiveness.** Benchmarks should not introduce any changes in the behavior of the target system, and if they do, the changes should be minimal. In practice, the instrumentation related to the execution of the benchmarks means that the target will experience some change (usually performance). This intrusion should be kept at a minimum, and its effects must be taken into consideration when producing the benchmark results.
- **Scalability.** This concerns the ability of the benchmark to keep its ability to evaluate systems of increasingly larger sizes. A careful analysis must be conducted to understand the growth in complexity, effort (cost and

execution time) when compared to the growth of the target system. Case studies involving large target systems help validating the results of this analysis.

- **Feasibility.** This relates to the effort of deploying (executing) the benchmark. If the execution of the benchmark is too complex or requires a great effort (time, money, or operator effort), the benchmark will probably not be accepted as a standard. The feasibility property can be evaluated in a similar way as the scalability property.

2.3.3.5 Dependability Benchmark Examples

The book (Spainhower and Kanoun 2007) compiles most of the works on dependability benchmark in recent years, including dependability benchmarks for on-line transaction process systems, web-servers, and operating systems. In fact, this book includes several examples that were developed in the context of the Dependability Benchmark Project (DBench 2004).

One of the key contributions of DBench project was the dependability benchmarks for on-line transaction processing (OLTP) systems, web servers, and operating systems. These works were documented and published in the form of peer-reviewed papers. In (M. Vieira and Madeira 2003a), it is proposed a dependability benchmark for on-line transaction processing (OLTP) systems. This dependability benchmark uses the workload of the TPC-C performance benchmark and specifies the measures and all the steps required to evaluate both the performance and key dependability features of OLTP systems, with emphasis on availability.

A dependability benchmark for web-servers is presented in (Durães, Vieira, and Madeira 2004). SPECWeb99 benchmark, the faultload component and new measures to dependability are used. The measurements address both user and system-administrator viewpoints and target the key properties of the service expected from web servers.

In (Kalakech et al. 2004), it is described a dependability benchmark for operating systems. The goal of this benchmark is to characterize qualitatively and quantitatively the OS behavior in the presence of faults and to evaluate performance-related measures in the presence of faults.

Based on dependability benchmark approach, (Véras et al. 2010) proposed a systematic approach for benchmarking software requirements for space systems that adopt the European Cooperation for Space Standardization (ECSS) standards. The goal is to ensure that requirements specifications comply with the ECSS

standards, as well as they do not have any of the most frequent errors on this kind of document. In this benchmark approach, the workload is replaced by a checklist document aimed at obtaining measures that portray specific characteristics of the software requirements specification document. This checklist originated from an extensive field study that mapped the most frequent errors found in software requirements documents of space systems. A case study comparing the requirement of real space systems is also provided.

2.4 TOWARDS RISK-BASED SECURITY BENCHMARKS

The goal of security benchmarks is to measure security of systems for comparative purposes. Although to date there are no security benchmark standards proposed by academia or industry according to the notion of benchmark and its properties as stated previously, there are many security metrics, techniques and benchmark initiatives that have been proposed.

This section aims to describe relevant works in the field of security benchmarking. It starts by defining and presenting examples of security metrics. Then, examples of security assessment techniques and methods, focusing on vulnerability and attack injection are highlighted. Finally, we present approaches that have been used to compare the security of systems.

2.4.1 Security metrics

A security metric is an essential component of a security benchmark or of any useful security evaluation approach. It enables users to know the level of security of a system, to compare a system with others, and identify the most secure. The problem is that security is far more difficult to measure than other system attributes such as performance (where what matters is the *speed* of a system or component to execute a given task). A security metric should encompass different characteristics that affect the level of a system security such as the impact of system vulnerabilities, the probability of the occurrence of successful attacks, and the presence of security mechanisms. In other words, a security metric should capture quantitatively the intuitive notion of “the ability of the system to resist attack” (Brocklehurst et al. 1994).

(Wang et al. 2009) presents an approach to define software security metrics based on the representative weaknesses present in a system, where “representative” refers to vulnerabilities that can be exploited by attackers. The assumption here is that the number of vulnerabilities and the impact of these vulnerabilities (when successfully exploited by attacks) is an important security indicator. Another similar initiative that uses the notion of vulnerability impact to estimate the

security of software can be found at (Houmb, Franqueira, and Engum 2010). However, these initiatives rely on vulnerability information that is reported in the field, not considering the effects of hidden vulnerabilities to the security of the system.

Although it is possible to estimate system security using an integrated view (i.e., one single metric that indicates the security level of the whole system), in many cases security is expressed in the form of several complementary metrics. The Center for Internet Security (CIS 2012) has coordinated the Consensus Security Metrics (CSM) initiative to help companies make cost-effective security decisions. This initiative brought together a team of one hundred industry experts to investigate and define comprehensive security metrics and to define how to collect and analyze data on security process performance and outcomes. CMS proposed 28 metrics definitions organized in three categories:

- **Management metrics** provide information on the performance and business functions, and on the impact on the organization (e.g., Cost of Incidents, Percent of Systems with No Known Severe Vulnerabilities, Patch Policy Compliance, IT Security Spending as % of IT Budget)
- **Operational metrics** are used to understand and optimize the activities of business functions (e.g., Mean-Time to Incident Discovery, Mean-Time Between Security Incidents, Mean Cost to Mitigate Vulnerabilities, Mean Cost to Patch, IT Security Budget Allocation).
- **Technical metrics** provide technical details as well as foundations for other metrics (e.g., Number of Incidents, Number of Known Vulnerability Instances, Percentage of Critical Applications, Risk Assessment Coverage, Security Testing Coverage).

Security metrics related to financial aspects are also found in literature. In (Schechter 2002) was proposed the security of a system based on the estimation of the cost to exploit vulnerabilities to breach security (the lowest expected cost for anyone to discover and exploit a vulnerability in that system). A method to put a price on vulnerabilities is presented by the same authors in (Schechter 2004).

Metrics Center is a cloud-based service for designing, deriving, and delivering metrics (MetricsCenter 2012). In this initiative, metrics are unambiguously defined and mapped to business context in an on-line catalog. This catalog congregates data from different security metrics work, including those present in NIST standards, ISO/IEC 27002, and PCI DSS (Payment Card Industry Data Security Standard), among others. Each metric is characterized in terms of objective, unit of measure, frequency, source, and instructions to calculate each

metric are provided.

To be useful, security metrics should be defined and measured according to a set of requirements (named in the benchmarking field as *properties*) that, in our view, represents characteristics that should be taken in to account in the definition of security benchmark metrics. These requirements can be summarized in the following terms (Jaquith 2007):

- **Consistently measured.** Metrics that rely on subjective judgment are not metrics, but ratings. This means that different people should be able to apply the measurement to the same dataset and come up with equivalent answers. This refers to the important property of *repeatability*.
- **Cheap to gather.** Methods of gathering data should not be time-consuming and costly.
- **Expressed as a cardinal number or percentage.** Good metrics are given in cardinal number or percentage (counts how many of security there are) rather than ordinal number (denotes which position of security is in).
- **Expressed using at least one unit of measure.** Good metrics should contain at least one associated unit of measure that characterizes what is being counted (e.g., defects, security).

The above requirements are conceived to help security specialists to define useful and meaningful security metrics. From a benchmarking perspective, a security metric value is an indication of what happens in the real world. If the number indicates that the system is secure, then the system should tolerate attacks, otherwise users will not trust the metric and the benchmark will not be accepted.

2.4.2 Security assessment

This section presents examples of assessment techniques and methodologies that have been proposed, starting by initiatives that treated security as a process.

2.4.2.1 Security Assessment by Design

Security by design means that the security of a product (system, component) is evaluated since the beginning of the development cycle and are dealt in a systematic manner during the development. This approach is the answer to the realization that security needs to be built into the software from the very beginning and security activities need to take place throughout the software life-cycle (Ardi, Byers, and Shahmehri 2006). IBM Secure Engineering Framework (Buecker et al. 2010), Microsoft Trustworthy Computing Initiative (Gates 2002),

and Comprehensive, Lightweight Application Security Process (OWASP-CLASP 2012) are examples of initiatives that have considered security not as a product, but as a process. In (Futcher and von Solms 2008), a set of guidelines is provided for secure software development based on a number of internationally recognized standards and best practices.

Large software developers have realized the importance of focusing on security as a process and dealing with it in an integrated manner with the development process. Microsoft (Gates 2002) brought this issue to the attention of all its developers, which motivated a clear shift from focusing on features to spotlighting security and privacy in a large software company. One year later (2003), members of the Secure Windows Initiative and the Trustworthy Computing Security Team at Microsoft announced the book “Writing secure code” (M. Howard et al. 2003), with the best practices for writing secure code and stopping malicious hackers, also focusing on .NET platform. More specifically, this book offers practical insights into secure design, secure coding, and testing techniques, many of which are not documented in previous security works.

The Open Web Application Security Project (OWASP 2012) proposed a method to apply security to an organization's application development process: the CLASP ((Comprehensive, Lightweight Application Security Process)). CLASP (OWASP-CLASP 2012) consists of a set of processes that contains formalized best practices to build security into software development life cycles. CLASP security practices perform application assessments, capture security requirements, build vulnerability remediation procedures, define and monitor metrics, and so on.

2.4.2.2 Security Assessment Techniques

One technique that has been used in recent years to evaluate the security of system is the injection of vulnerability and attacks. Here “injection” refers to an intentional action of seeding and exploiting a vulnerability in a system. In fact, **vulnerability injection** is the deliberate insertion of vulnerabilities into software code to accelerate the occurrence of successful attacks. From a software code perspective, a vulnerability injection is as a fault that, once activated by an attack, will compromise the security attributes of a system. To inject the vulnerability, it is necessary to characterize the fault type that originates the vulnerability, named in (J. Fonseca, Vieira, and Madeira 2009) as vulnerability operator. The **vulnerability operator** is then a set of pairs of location patterns and vulnerability code change. The **location pattern** characterizes the place in the source code where the vulnerability is likely to be found. The **vulnerability code change** defines what has to be done to the piece of code targeted by the location pattern in

order to make it vulnerable, without affecting the functional behavior of the web application.

Fonseca proposed a vulnerability injection methodology for web application that is organized in three main steps (Jose Fonseca 2011):

1. **Static analysis of the source code of the web application**. This is done by analyzing the source code dependencies, input and output variables.
2. **Search for the location where a vulnerability may exist.** This is done by examining the code of the web application in order to identify all the points where each type of fault can be injected, resulting in a list of possible fault locations and their respective vulnerability types.
3. **Mutation of the code to inject a vulnerability.** This is done by applying, to the web application source code, the vulnerability code change defined by a vulnerability operator.

The importance of vulnerability injection to our work is that we take advantage of this approach to test the effect of hidden/unknown vulnerabilities in the system, by injecting vulnerabilities in one of the components of the system under benchmark and exploiting these vulnerabilities using attack injection.

Attack injection refers to the exploitation of vulnerabilities that are injected in the system. The goal of attack injection is to simulate real attacks to test how the target system behaves in the presence of attacks. This can be useful in several security assessment scenarios, for example, to test Intrusion Detection Systems. An example of attack injector aimed at discovering unknown vulnerabilities was proposed in (Neves et al. 2006). He proposed an attack injector tool (AJECT) aimed at discovering new vulnerabilities on network-connected servers. The AJECT tool uses a specification of the server's communication protocol to automatically generate a large number of attacks according to predefined test classes. While these attacks are performed, the tool monitors the behavior of the server looking for an incorrect system behavior. In this case, an incorrect behavior indicates a successful attack and the existence of a vulnerability.

A methodology that uses vulnerability injection to mount attacks against web applications was proposed in (J. Fonseca, Vieira, and Madeira 2009). The idea behind the methodology is that by injecting realistic vulnerabilities in a web application and attacking them automatically security mechanisms can be assessed.

The purpose of vulnerability assessment is to answer how vulnerable a system is. According to (Jaquith 2007), there are at least three potential ways that have been

applied to assess vulnerabilities in industry:

- ***Black-box measures.*** This technique seeks to identify or predict known, exploitable vulnerabilities and conduct security tests without knowledge about the code. In the context of web applications and web services, this is also termed as ***black box testing*** or ***penetration testing***. Examples of black-box measures are vulnerability scanners tools (e.g., Acunetix Web, Vulnerability Scanner (Acunetix 2012), IBM Rational AppScan (IBM Appscan 2012), HP WebInspect (HP WebInspect 2012), Foundstone WSDigger (WSDigger 2012) and wsfuzzer (neuroFuzz 2012), and (Nikto2 2015)).
- ***Code security measures.*** This technique seeks to identify design and implementation vulnerabilities in the software code and is termed as ***white box testing***. Examples of code analysis tools are (HP Fortify 2012), (Ounce Labs 2012) and (Pixy 2012).
- ***Qualitative process measures and indices.*** This technique seeks to create qualitative risk indices based on the business impact and criticality of vulnerabilities identified in security assessments. The Common Weakness Scoring System (CWSS 2011) is an example of method that provides a quantitative measurement of the unfixed weaknesses that are present within a software application.

One method that has been used to the prediction of the number of vulnerabilities in systems is the Vulnerability Discovery Model (VDM). VDMs are probabilistic methods for modeling the discovery of software vulnerabilities and are based on statistical methods. VDMs can be applied to evaluate the security risk of systems (Omar H. Alhazmi and Yashwant K. Malaiya 2006) and can be used to estimate characteristics of the vulnerability discovery process. In (Woo, Alhazmi, and Malaiya 2006), it is showed the applicability of VDMs models to predict the number of vulnerabilities that may potentially be present in a web server but may not yet have been found. A more detailed discussion on the effectiveness of VDM methods can be found at (Andy Ozment 2007), (Cavusoglu and Raghunathan 2007), and (Alhazmi and Malaiya 2006).

One important project in the field of security assessment is The Making Security Measurable, which is led by MITRE Corporation (MITRE Corp. 2012) and has brought together existing activities and initiatives related to security evaluation (R. A. Martin 2008). This project brings together dictionaries of vulnerabilities and attacks, assessment methods to evaluate the risk of vulnerabilities and configuration issues, and repositories of vulnerabilities, security checklists, and

security configuration. Vulnerability information one can find in on-line repositories such as the National Vulnerability Database (NVD 2014) and the Open Source Vulnerability Database (OSVDB 2014) are usually identified and categorized to the dictionaries maintained by MITRE (e.g., Common Vulnerability Enumeration, Common Weakness Enumeration) and have vulnerability risk scores estimated according to third-party organizations (e.g., Forum of Incident Response and Security Teams).

It is worth pointing out that the security techniques discussed above do not provide an integrated view of system security. In our view, these techniques could be used as supporting tools of a security measurement approach to speed up the detection of known vulnerabilities and to assess the effects of vulnerabilities in an automated way.

2.4.2.3 Security Assessment Methodologies

Security evaluation is also termed in literature as risk assessment analysis, since the goal is also to identify the sources of threats that would lead to a successful attack. Security risk assessment methods can be organized in two groups: quantitative (with the purpose of translating the security risk of an organization in a single set of metrics) and qualitative (estimate the potential impact of a security breach as high, medium, low). This subsection presents relevant examples of these approaches and then we describe their limitation from a security evaluation point of view. A detailed survey of information security risk analysis methods can be found at (Behnia, Rashid, and Chaudhry 2012).

The OCTAVE (Operationally Critical Threat, Asset, and Vulnerability Evaluation) is a qualitative risk assessment approach proposed by CERT-SEI (Alberts et al. 1999) to manage information security risks, helping organizations to map threats and protect organization assets. One characteristic that makes OCTAVE unique is that it is based on operational risk and security practices that are identified by the organization, not relying on outside requirements. OCTAVE is based on a three risk assessment phases: (a) building of a profile of threats, (b) identification of infrastructure vulnerabilities, and (c) development of a security strategy and plans considering the most critical assets to the organization. To support this methodology, an OCTAVE implementation guide is also provided, containing the tools and techniques that can assist users in the risk assessment conduction. Without any doubt, OCTAVE is a bold approach to help users to map the most critical assets to an organization and develop security plans to fix vulnerabilities and avoid the possible threats to the information technology environment. However, this approach lack of concrete details on how to measure

the security level of functionally equivalent systems and is not aimed at benchmarking the security of system components.

(Karabacak and Sogukpinar 2005) proposed the Information Security Risk Analysis Method (ISRAM) aimed at assessing the security risk of organizations with the participation of managers and staff in a quantitative manner. This consists in the conduction of a survey that is previously built based on the security needs of the target organization. This survey contains questions that help to estimate the probability of occurrence and consequences of security breaches, which are then used to estimate the security risk. As can been seen, ISRAM is a quantitative risk analysis approach with the advantage of translating the security needs of an organization in one single measure. However, since the security needs of different organization may vary, this approach cannot be used to compare the security risk of organizations using the risk measure.

The U.S. National Institute of Standards and Technologies (NIST) proposed the Risk Management Guide for Information Technology Systems (NIST SP 800-30) as a recommendation for US agencies and organizations (Stoneburner, Goguen, and Feringa 2002). NIST risk analysis starts by characterizing the organization systems and identifying system vulnerabilities and potential threats. Then, the likelihood and impact of security breaches are evaluated and a risk matrix is used to determine the security risk that the systems pose to the organization. NIST also has two additional steps aimed at eliminating the identified risks and documenting the results in the form of a risk assessment report to help upper management to make decisions.

The benefit of these risk assessment approaches to users is that they address security with a more holistic approach, not limited to a particular class of system or vulnerabilities. However, they do not provide a way to translate operational risks and security best practices in a single set of metrics applicable to a particular system that could be used to benchmark their security, which is the intent of our security benchmark methodology.

2.4.3 Security Benchmarking Initiatives

Several approaches aimed at assessing and comparing the security features of systems have been proposed in the past. The idea behind these approaches is to either classify a system according to a security level/class or provide a security score that will help end-users to select the most secure system - the highest the security level/class/score of a system, the more secure (the more protected against attacks) a system is. None of these approaches has targeted software-based systems as a whole according to the philosophy of classical performance and

dependability benchmarks. However, they provide a clear idea of the benefits of security benchmarks.

(Nibaldi 1979) proposed 6 security levels (and requirements for each level in the form of security policy, accountability, assurance, and documentation measures) to characterize the internal protection mechanisms of computer systems using aspects such as access control, protection policy, and design implementation.

The (Common Criteria (CC) 2009) is a security evaluation framework that is widely accepted in industry. CC evaluation focuses on the software development process rather than the software itself (Shapiro 2003). CC defines a set of IT requirements of known validity to help customers to establish security requirements to protect products and systems. CC also defines the Protection Profile which is a set of security requirements and objectives for a category of products or systems which meet similar consumer needs for IT security (examples of protection profiles are available at Common Criteria Portal (CC Protection Profiles 2012)). The Common Criteria Agreement provides a comprehensive methodology to help evaluators to apply Common Criteria audit: the Common Methodology for Information Technology Security Evaluation (Common Methodology (CEM) 2009). More specifically, CEM defines the minimum actions to be performed by an evaluator in order to conduct a CC evaluation, using the criteria and evaluation evidence defined in CC: Common Criteria is composed by seven assurance levels (EALs – Evaluation Assurance Levels) that cover many features of a given target system, such as documentation, security features, and development process. Every assurance family contributes to the assurance that the Target of Evaluation (TOE) meets its security claims. EALs provide a uniformly increasing scale which balances the level of assurance obtained with the cost and feasibility of acquiring that degree of assurance.

(M. Vieira and Madeira 2005) proposed a security characterization for Database Management Systems (DBMS) based on a set of security classes. That characterization analyzes DBMS security-related mechanisms (such as user authentication, user privileges, encryption, etc.) enabling users to select the DBMS best suited to his particular security requirements. In this approach, systems are classified using the following metrics:

- Security class (SCL): DBMS are categorized using a set of security classes varying from Class 0 to Class 5. For each class a set of security requirements is identified. A system is classified in a given class if it fulfills the requirements for that class.
- Security requirements fulfillment (SRF): to complement the security class

it is proposed the use of an additional metric that characterizes how well a given system fulfills the set of security requirements (in a scale from 0 to 100). This metric is useful to differentiate systems in the same security class.

(Neto and Vieira 2008) proposed a generic methodology to assess the security configuration of systems (i.e., servers in general). The merit of this approach is the proposal of steps ranging from the collection of security recommendations from different sources to the proposal of tests to assess and compare the security configuration of systems. An approach proposed by (Mendes et al. 2008) has applied and extended this methodology through a characterization of security practices according to the ISO 17799:2005 international standard for web servers. Additionally, (Mendes et al. 2008) was not only concerned with the configuration aspects of web servers, but also with the design of a secure network and the implementation of a strong security policy. All these aspects are important to reduce the possibility of successful attacks over web servers.

In 2010, (Neto and Vieira 2010) evolved (Neto and Vieira 2008) approach and proposed an alternative to assess security. Instead of assessing the security of a system by focusing on vulnerabilities and attacks, the idea is to assess the trust that administrators put on the system by implementing security practices. Their definition on security metric is as follows: "*the degree to which security goals are met in a given system allowing an administrator to make informed decisions*". By proposing a trust-based metric, the authors are interested in quantifying the trustworthiness relationship between an administrator and the system he manages. To compute this metric, a set of steps is defined in the context of database management system: 1) Database administrator analyzes all recommendations on the list; 2) For each recommendation, he evaluates if the configuration being assessed implements the recommendation or not; 3) The result of the evaluation is an answer of Yes or No for each security recommendation; 4) Use the equation available to weigh the recommendation in terms of threats that it exposes (in the case of not being implemented); 5) Compute the overall untrustworthiness value. This method is then applied to benchmark the untrustworthiness of real database management systems.

In (Antunes and Vieira 2010) it was proposed a methodology to benchmark web services security scanner tools (in previous research works the same authors found out that the most used vulnerability scanners have different vulnerability coverage, meaning that they uncover different sets of vulnerabilities (M. Vieira, Antunes, and Madeira 2009)). They proposed a method to analyze the flaws and limitations of web application scanners by using one secure version and one

insecure version of a custom-built web application. Another benchmarking approach comparing the security effectiveness of web application scanners is provided in (SecToolMarket 2012), where the authors assess not only their coverage features, but also audit and authentication features, among others. Inspired by these approaches, one could use our benchmark methodology to gather information in the presence and absence of security mechanisms to verify the security level improvement when the security mechanism is used or not.

The Center for Internet Security (CIS 2012) has proposed a set of security benchmarks for several classes of systems. CIS also provides auditing tools that compare the configuration of systems and reports conformance scores on a scale from 0 to 100. The security benchmark targeting Apache Web Servers, for example, has two configuration levels: the first level covers settings such as access control, authentication mechanisms, patches updating, and request limitation; the second level covers settings such as cryptography, logging, and blocking operating system commands. In addition to the fact that CIS benchmarks focus on configuration security issues, the tests performed are limited to a fail-pass approach and do not assess the effects of one insecure component to the whole system (e.g., the risk that a missing configuration poses to the whole system). Additionally, this cannot be formally considered a benchmark as CIS benchmarks are platform dependent and are not based on a well-defined specification and in conformity with a strict set of properties.

A model and framework to help in the definition and improvement of security benchmarks for e-business systems is proposed in (Pye and Warren 2007). The goal is to address the relevance of benchmark development over time and the changes in threat focus. For such purpose, a continuous improvement approach is described focusing on five broad areas: organizational security, infrastructure security, application security, network/system security and user management security. The framework consists of a four-stage process (initial security benchmark, online security assessment, current benchmark analysis, and continuous improvement analysis) to initially create a security benchmark and then continue to improve upon such benchmark developments. The minimum security requirements that a system should have, which belong to the first stage, are those specified in Australia and New Zealand information security standards (AS/NZS 4360 1999). Although the idea of continuous improvements of a security benchmark is quite useful, no details are provided about the metrics and the way to implement the benchmark in the field. For example, the security assessment just informs that it should be a “pass/fail” approach. Additionally, a standardized way to implement the benchmark is also lacking.

Initiatives to measure the security of systems in a more standardized way have been proposed in recent years. (Das, Sarkani, and Mazzuchi 2012) proposed a software product evaluation method based on a quantitative security risk model. This method consists of collecting vulnerability data from public databases, categorizing the collected vulnerabilities in topics, and calculates the probability distribution and a score for each vulnerability. This probability, score and number of vulnerabilities are used to estimate the security risk of a system based on vulnerabilities that are discovered. To identify these vulnerabilities, two approaches are used: a static analysis of the source code and the execution of vulnerability scanners. Although it represents an important attempt to use security risk as a benchmark metric, the assumptions to consolidate the metric is not totally clear and easy to understand (e.g., the topic modeling approach and the weights considered to compute the final metrics) affecting the repeatability and reproducibility of this approach. Additionally, this approach is source-code dependent and considers only known vulnerabilities to estimate the security risk score, leaving unaddressed the important threat of hidden vulnerabilities. Another important aspect is that the result of the approach is consolidated in three different metrics representing three different aspects of the system (system requirement risk, static analysis risk, and dynamic analysis risk), making it hard the task of identifying the most secure among the evaluated systems.

A more comprehensive framework to benchmark the security of systems is proposed in (Neto 2012). The purpose of this framework is to evaluate the tendency of a system to have unknown or hard to detect vulnerabilities or security problems (termed by the authors as trustworthiness benchmarking) and it is organized in two parts: the security qualification and the trustworthiness benchmarking. The security qualification is designed to target the vulnerabilities and security mechanisms that are obvious to exist in the system. This framework assigns a security level equal to zero to any system that has obvious vulnerabilities and disqualifies it (with a zero score) from the benchmarking process. For these authors, security deficiencies or publicly known flaws present in a system should never be used in a benchmarking approach since in a real situation the system will be patched when put into production. The trustworthiness benchmarking part consists of identifying threats that could result in a successful system attack (e.g., denial of service attacks, elevation of privileges, information disclosure) and then assessing the characteristics (in the form of security practices) that are in place to avoid these threats. A system that covers a large portion of these characteristics (with the implementation of security practices, for example) is a trustable system according to this benchmarking approach. Although this approach represents a very important contribution to the

field of benchmarking and security, the fact that it considers any obvious vulnerability with the same level of importance (security level is zero) is not realistic. Two functionally equivalent systems with vulnerabilities with different risks have obviously a different risk level that should be considered by a benchmark. Also, this approach does not consider the execution of real attacks to stress the security of systems. Another important point is that the identification of system threats and security characteristics is done by a time-consuming step of field study over documents from different sources, which makes the benchmark hard to reproduce and execute and makes the validation part, as recognized by the authors, a complex task.

The need of security benchmarks was also identified and reported by the Amber project (Assessing, Measuring and Benchmarking Resilience)(Bondavalli et al. 2009). This project, which was funded by European Commission under the FP7 program, brought together senior researches to define a research roadmap in the field of resilience and benchmarking. In the final roadmap report, authors suggest attackloads and injection tools to the development of security benchmarks. The reasoning behind this recommendation is that security benchmarks could adapt the techniques applied in the dependability field, where the faultload component is used to evaluate the tolerance of the target system against faults while performance and dependability measurements are collected. Applying this notion to the security field, an attackload could be built to stress the system with real attacks while observing the security behavior of the system. (Neto 2012) strongly disagrees with the use of attackload notion in the security benchmark field pointing out that the definition of an attackload is a complex problem and that it makes no sense to attribute a level to a system that is not able to resist attacks. Despite these objections, in this thesis we demonstrate that we were able to successfully build a security benchmark methodology incorporating attackload and vulnerability injection components to evaluate the security of systems in an experimental way. Although we agree that the identification of potential threats and the evaluation of security characteristics are important steps to increase the security of systems, we also believe that the only effective way to measure security must include subjecting systems to real attacks.

2.5 CONCLUSION

This chapter presented and discussed the state of the art on security benchmarking and. In order to help users to better understand the challenges in the design, development and deployment of security benchmarks, this chapter covered several foundational concepts ranging from vulnerabilities, attacks, and security assurance to security metrics and previous benchmarks initiatives.

We showed that benchmarking the security of computer-based systems is an emerging and pertinent research topic, as society is becoming increasingly more dependent on secure computer-based systems. We maintained that benchmarking the security of software-based systems is a research topic even more challenging than simply assessing security attributes as benchmarking presents specific issues such as representativeness and acceptance which are very hard to solve.

The key works that either guided or inspired the development of this PhD Thesis were also presented and are shortly described as follows:

- Database Management System and Web Servers Dependability Benchmarks developed in the context of the **Dependability Benchmark Project** (DBench 2004). These works used and applied the concept of workload and measures from classical performance benchmarks, and included new measures, a faultload element, and benchmark management systems, being undoubtedly the most important source of information to the definition and development of realist benchmarks to the security field.
- The **Vulnerability and Attack Injection** approach proposed in (Jose Fonseca 2011). This work helped us to adapt the faultload benchmark element to the field of security. Basically, in the experimental part of our methodology, vulnerabilities are injected, attacks targeting these vulnerabilities are executed, and the behavior of the systems are assessed to check in which degree the system security was impacted.
- The works developed in the context of **Making Security Measurable** led by MITRE Corporation (MITRE Corp. 2012). For example, the Common Weakness Enumeration (CWE 2012) and several NIST security standards helped us to properly understand the challenges involved when measuring security for comparative purposes.
- The **Common Vulnerability Scoring System** proposed by FIRST. The importance of this work is given by the fact that they provide an open framework and easy-to-use approach to estimate the individual risk of vulnerabilities. In fact, CVSS equation was included in our security tools to estimate our final security risk benchmark metric.
- The Vulnerabilities Repositories created and maintained by the **National Vulnerability Database** (NVD 2014) and the **Open Source Vulnerability Database** (OSVDB 2014). We collected vulnerability information from these platforms in the static (analytic) part of our security benchmark methodology.

As shown in this chapter, methods to evaluate the security of systems have been used mostly to identify typical security problems, and in recent years the focus has shifted to a better description of security metrics and to the assessment of the security characteristics. As these proposals do not follow a systematized, experimental approach to benchmark security (as seen in the field of performance and dependability), the development of security benchmarks remains an urgent need and a research topic of utmost importance. To contribute to this topic, we describe in the next chapter our security benchmark methodology, which can be applied to any class of software-based system.

We are absolutely convinced that the security methodology benchmark proposed in this thesis is a novel and important contribution to the security and benchmarking fields. We brought to the security field the elements used in dependability and performance benchmarks (metrics, workload, experimental setup, procedures and rules), complying with key benchmark properties (representativeness, repeatability and so on) that are important to the validation of our benchmark approach. We also use an analytical approach to identify known vulnerabilities and an experimental approach to stress the system with real attacks and observe the effect of unknown vulnerabilities. In addition to that, we apply the notion of risk to differentiate the impact and exploitability of vulnerabilities and consider these risk levels in the estimation of the benchmark metric. More importantly, our methodology enables users to identify the most secure among equivalent systems in an effective way, as we demonstrate in the case study presented in this thesis.

CHAPTER 3

3. BENCHMARKING THE SECURITY OF SOFTWARE-BASED SYSTEMS

This chapter describes our security benchmark methodology. The methodology is generic and suitable for any class of software-based system, and the description given in this chapter is independent from any specific software target. The goal of this methodology is to enable users to identify the most secure among functionally equivalent systems. This is achieved through the measurement of system security in a standardized way, using the notion of risk to estimate the security level of the evaluated systems.

Our benchmark methodology is structured in a similar way as a benchmark specification document (as opposed to simply providing tools ready to run), following the logic of established benchmarks from other fields to better allow developers to instantiate the methodology and implement security benchmarks to specific software system classes using any technology and tools available following the guidelines presented here. Note that an example of implementation of our security benchmark methodology for web-serving systems is provided in the Chapter 5.

The remainder of this chapter is as follows: Section 3.1 presents the benchmark concepts. Section 3.2 describes our strategy to benchmark the security of. Section 3.3 and 3.4 provide a detailed overview of the static and dynamic part of the benchmark. Section 3.5 introduces the benchmark components, whose implementation is specified in section 3.6. Section 3.7 concludes this chapter.

3.1 BASIC CONCEPTS

The characterization of the systems involved in a benchmark execution is a particularly relevant topic, as benchmark users need to properly distinguish the systems designed to support the benchmark from those under evaluation. In fact, our methodology is built around systems derived from classical works on performance and dependability benchmarks (SPEC 1988; TPC 1988; DBench 2004) and, henceforth, we adopted the same terminology used in these previous works. The definition of the main benchmark systems are provided next and their relationship in a benchmarking scenario is illustrated in Figure 3-1.

- **Benchmark Management System (BMS)** refers to the components that manage the benchmark experiments. These components are installed and deployed during the preparation of the analytical and experimental setup of the static and dynamic part and take part of the benchmark instrumentation. These components are properly described in the next sections.
- **Benchmark Target (BT)** refers to the system or component that is characterized during a benchmark run. Considering, as an example, the context of web serving systems, the benchmark target can be the web server, the web application, the database, or the operating system (or even subsets of these components).
- **System Under Benchmark (SUB)** refers to the system that provides the operational environment for the execution of the benchmark target. This may include components such as operating system, libraries, and interface components, to name a few examples. It also includes benchmark components directly related to the execution of the benchmark, such as the workload. As our security benchmark methodology is generic there are no specific restrictions concerning classes of software-based system for the system under benchmark role. As an example, in the methodology implementation we provide in Chapter 5, a web serving system is used as the system under benchmark, with the web server component as the benchmark target, and the remaining components of the SUB include the operating system, the database, and a web application.

The goal of measuring security for comparative purposes makes security the most important concept to be defined. According to (Avizienis et al. 2001), security is the concurrent existence of the attributes confidentiality, integrity, and availability. *Confidentiality* refers to the protection of functionality and data against unauthorized access (Bishop 2003). *Integrity* refers to the trustworthiness

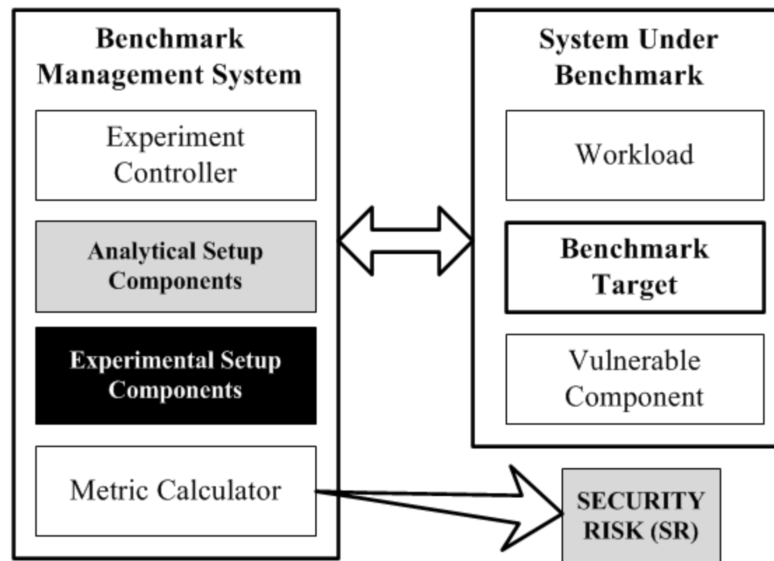

Figure 3-1. Relationship between Benchmark Management System (BMS) and System Under Benchmark (SUB) in a Security Benchmark Execution

of data or resources, assuring that the actions and data are correct (Bishop 2003). *Availability* refers to the readiness of the system to provide the expected service, i.e., to the ability to use the information or resource desired (Avizienis et al. 2001). It is important to make clear that a system is not secure if attackers are able to obtain restricted content (confidentiality), or to modify it (integrity), or make it unavailable (availability).

The security of a system is compromised when one or more *vulnerabilities* are successfully exploited by *attacks*. A software vulnerability - as already described in Chapter 2 - is an instance of a mistake in the specification, development, or configuration of software such that its execution can violate the explicit or implicit security policies (Krsul 1998). An *attack* (also termed in this thesis as *vulnerability exploitation*) is any action aimed at compromising the security of a system (adapted from (Stallings 1999)). A *successful attack* is the one that exploits a vulnerability and compromises, at least in part, one of the security attributes already described.

The need of quantifying the loss caused by attacks and taking into account the probability of their occurrence led us to use the notion of risk. The definition of *risk* used in this thesis was adapted from (Lowrance 1976) and refers to the measure of the impact and probability of adverse effects. This notion brings two important characteristics (Kirkpatrick, Walker, and Firth 1992): loss (a

vulnerability exploitation that has unwanted consequences or losses) and uncertainty (a vulnerability that may or may not be exploited). In the context of our work, the loss factor refers to the impact of an attack to the security attributes of a system. To measure the extent of such impact, we take advantage of the criteria defined by the Common Vulnerability Scoring System - CVSS (Mell, Scarfone, and Romanosky 2007). The impact assessment over system confidentiality is done by evaluating the system ability to keep confidential restricted areas, among other verifications. The verification if system responses are as expected is necessary to assess the impact over system integrity (i.e., attacks were unable to alter system content). The impact evaluation over system availability is performed by checking if the system becomes unresponsive for any period of time during the execution of attacks. The highest impact level is the one that cause a complete compromise of system confidentiality, integrity, and availability.

The probability of an attack occurrence is given by the level of easiness to exploit a vulnerability. This easiness is termed by the Common Vulnerability Scoring System 2.0 as *exploitability* and is given considering three independent factors: the network location from where the attack is executed (local network, adjacent network, or remote network); the need of system authentication (none, single, multiple); and the complexity to mount the attack over the target vulnerability (low, medium, high). In other words, an attack with the highest probability level is the one that can be executed remotely, with little skill needed to mount and execute, and with no need of authenticating into the system.

3.2 SECURITY BENCHMARKING STRATEGY

Our strategy to benchmark the security of software-based systems consists of estimating the risk of individual vulnerabilities identified in each system component following an experimental approach and a specific set of procedures and rules. This means that the cornerstone of our methodology is the benchmark metric along with the approach we follow to evaluate system security and produce comparable and repeatable benchmark results.

The metric of our security benchmark methodology is security risk, termed here as *SBench*. This metric is computed by the weighted sum of the security risk of each component of the system under benchmark. This Vulnerability Risk (VR) is estimated considering the product of the impact of vulnerability exploitation (I) with the probability of a successful vulnerability exploitation (P), as detailed in Equation (1).

$$VRi = Ii * Pi \quad (1)$$

The impact factor refers to the effects of a successful vulnerability exploitation on the security attributes. The impact can vary from no impact to complete impact, and the meaning of complete depends on the system attribute being observed (examples of complete impact are: to the availability attribute - the attack brings the system down for a long period of time; to confidentiality - the attack renders the sensitive information available; integrity – the attacker is able to modify the files). The impact is expressed in term of numerical values (no impact: 0, complete impact: 10) according to the approach proposed by CVSS Version 2.0. The probability factor refers to the easiness of exploiting a vulnerability, which is translated by CVSS as the exploitability factor and is given by vulnerability access vector (e.g., local access required vs. remote network), complexity/skill requirements (low, medium, high) and authentication requirement (none, single, multiple). We use the CVSS scoring system, and impact and exploitability score fall in the range of 0 to 10. The higher the values, the higher the risk of the assessed vulnerability. Because our methodology uses probabilities expressed in the range of 0-1, we transform the CVSS range from 0-10 into the range 0-1.

The vulnerability risks of our benchmark metric goes through a categorization process. This categorization is needed because low-risk vulnerabilities do not harm the system in the same way as high-risk vulnerabilities (when successfully exploited). For example, if an attacker is able to compromise several low-risk vulnerabilities in the same system, he or she may not succeed in causing a complete loss of confidentiality, integrity, or availability to the system. However, a successful exploitation of a single high-risk vulnerability will result in a total security compromise of one of the security attributes. In a security benchmarking perspective, this means that the most secure system is the one with the lowest level of high-risk vulnerabilities. To bring this key notion to the benchmark metric, our security benchmark methodology uses the same criterion applied by the National Vulnerability Database to define the risk ranges of each one of these risk categories: Low Risk (VRL - if the risk score ranges from 0 to 3.9 as illustrated in Equation 2), Medium Risk (VRM - if the score ranges from 4.0 to 6.9 - Equation 3), and High Risk (VHR - if the score ranges from 7 to 10 - Equation 4). These ranges are defined based on the Risk Score Categorization of the National Vulnerability Database (NVD 2014). Also, each one of these categories is reflected in the benchmark metric in the form of a weight. Naturally, we defined a much higher weight to the high-risk category, since vulnerabilities under this category are more prone to be attacked and, when successfully attacked, the security compromise is complete.

The vulnerability risks are also grouped according to the component to which they belong. This is done to enable users to associate weights to each component to reflect the relative importance of each component to the system (the impact scale of that component). In this step, vulnerabilities are still grouped into risk categories described previously, and this measure is estimated by adding the weighted vulnerability risk (VRL, VRM, VRH) from each component resulting into one CR-Low (CRL), one CR-Medium (CRM), and one CR-High (CRH) for each component.

Our benchmark methodology is organized in two major parts: one static part, and one dynamic part. The static part is aimed at measuring security risk posed by existing and already discovered vulnerabilities. These vulnerabilities, if still present in a given system (e.g., the administrator has not yet patched the system), can be exploited and pose a security risk. The dynamic part of the benchmark addresses the unknown vulnerabilities. It is focused on the analysis of the behavior of the system when facing realistic attacks that may exploit unknown vulnerabilities. A detailed description of each benchmark part is provided later in the remainder of this section. The security risk resulting from both the static and dynamic parts is the main output of a security benchmark run. This is expressed in a value and is what we believe will enable benchmark users to compare functionally equivalent components and systems according to security.

3.3 STATIC PART

The static part is aimed at measuring security risk posed by known vulnerabilities. The main element of the static part is the information obtained throughout its execution that will allow us to confirm which vulnerabilities are present in the system under benchmark. The strategy we follow in this part is to collect vulnerabilities reported in the field matching with the brands and versions of the system under benchmark and use impact and exploitability information to estimate their vulnerability risk. Two data sources are used for retrieving information on vulnerabilities: (1) public databases listing known vulnerabilities (e.g., The Open Source Vulnerability Database, the National Vulnerability Database), and (2) results from security-testing tools containing large sets of known vulnerabilities. This strategy is designed to collect known vulnerabilities to the most possible extent and provide to the security benchmark the information needed to detect the presence of previously discovered vulnerabilities and estimate their risk. These components are complementary: the vulnerability repository collects information from public databases that were reported by anyone interested in disclosing a vulnerability; the security test repository obtains information from more restricted sources (these tools usually need a higher

technical expertise to understand the security tests and to identify the known vulnerability associated to each test). For each one of these components, it is necessary to have a specification of the repository data model and of the rules needed to build them. These items are covered later on – in the specification of our benchmark methodology.

As we execute security tests in the static part, it is important to provide the definition we use and discuss about the tests that are allowed. A security test is the verification of a security practice or the exploit of a known vulnerability. An example is the following security practice for web servers (Mendes et al. 2008): "Disable direct file system access (directory browsing, directory traversal, etc.)". The security test of this practice refers to the verification of the web server configuration or contents to confirm if directory listing is enabled or not (this particular test is related to the CVE-2006-3835 directory listing vulnerability, which affects Apache Tomcat 5). A positive security test is the one that confirms the existence of a vulnerability.

The security tests allowed in the static part are the passive ones, meaning that these tests do not attempt to damage the system, nor execute any action that could change the security behavior of the target system. For example, checking if a given port is opened is a non-intrusive testing in the sense it does not execute any action that could alter system availability, integrity or confidentiality. The assessment of the effects of intrusive security tests (e.g., a Denial of Service attack, which can affect availability, and therefore is an intrusive test) and of unknown vulnerabilities is covered in the dynamic part.

Another important aspect is to clarify how we measure the risk of the vulnerabilities detected during the execution of the static part. Public platforms such as vulnerability databases (e.g., (OSVDB 2014; NVD 2014)) already contain the CVSS impact and exploitability scores that we use to calculate the vulnerability risk of known vulnerabilities (with the proper adjustments detailed in sub-section C). If these scores are unavailable, then CVSS criteria should be followed to obtain the risk of the detected vulnerability. The risk resulting from the collection and analysis of known vulnerabilities reported in public databases and from the execution of security tests are added to the accumulated security risk of the static part.

Equations 2 to 4 present the formula to estimate the System Security Risk of the Static Part (SSR) for each risk category (SSRL – Low Security Risk; SSRM – Medium Security Risk; SSRH – High Security Risk). This estimation consists in the weighted sum of the component security risk (CRs – Component security risk of the static part). The weight of each component (Wc) reflects the relevance of

that component to the system. As different benchmarks users may have different views on the relative weight of the different components, the final assignment weights to components is left to the benchmark users. The "*i*" index means that this estimation should be done for each vulnerability detected.

$$SSRL = \sum_{i=1}^{n} (CRLsi * Wci) \quad (2)$$

$$SSRM = \sum_{i=1}^{n} (CRMsi * Wci) \quad (3)$$

$$SSRH = \sum_{i=1}^{n} (CRHsi * Wci) \quad (4)$$

3.4 DYNAMIC PART

The dynamic part of our approach addresses the measurement of the risk related to unknown vulnerabilities. This is done by executing attacks against the system under benchmark while observing the impact of vulnerability exploitation in the system. In other words, we observe the capacity of the system to resist those attacks (and the impact when it does not). A two-fold attack approach is followed:

Attacks against the system interface. These attacks target the components that interact with the end-users (the system interface). We use these attacks to observe the behavior of the system when dealing with input data overflow and malicious manipulation of system input parameters. The interface-related attacks use relevant vulnerability field studies (published by trustable sources of security community, e.g., (IBM X-Force 2012; Symantec 2014)) to identify common interface security issues. Table 3-1 presents the top 10 vulnerabilities found unpatched on web-based systems present in the 2015 Internet Security Threat Report published by Symantec (Symantec 2015). Vulnerabilities related to Secure Sockets Layer (SSL) are one of the top vulnerabilities that could be exploited by interface-related attacks.

Once the interface target is identified, it is necessary to implement and execute each attack. To this end, the attack patterns provided in Common Attack Pattern Enumeration and Classification shall be used (CAPEC-100 2014; CAPEC-152 2014). The CAPEC-217, for example, details how to exploit Incorrectly Configured SSL security levels, with examples of attacks and skills and knowledge required to conduct the attack. To automate this step, our methodology allows the use of tools such as penetration testing and exploit kits (e.g.

(Metasploit 2015) and others listed in (SecTools 2014)). The rules to select these tools are provided later on in section 3.6.3.

Attacks against a component outside the benchmark target. The system under benchmark is formed by interconnected subsystems or components with different security issues (e.g., operating systems, web server, database management system, etc.). A security issue is a problem with a direct, negative impact on system security (loss of data, system outage, elevation of privileges, and so on). In this sense, it is important to observe the behavior of the benchmark target when attacks are conducted against other elements of the system (e.g., the web application) that are not part of the benchmark target but interacts with such system.

Our approach includes the injection of realistic vulnerabilities in the external component and launching attacks that exploit such vulnerabilities. Vulnerability injection is the artificial injection of software faults that, when activated (i.e., successful attack), compromise either partially or completely at least one of the security attributes of the system. The injection of vulnerabilities is justified by the need to accelerate the occurrence of vulnerability exploitation by attacks in a controlled environment: because the natural occurrence of attacks and vulnerability exploitation occurs at a slower rate than what needed for the benchmark experiments, we inject representative vulnerabilities and latter direct attacks to these vulnerabilities to observe the reaction of the system or specific components (not the one where vulnerabilities were injected). This idea is similar

Table 3-1. Top 10 Vulnerability Found unpatched in Web-based Systems (Symantec 2015).

Rank	Vulnerability Name
1	SSL/TLS Poodle Vulnerability
2	Cross-Site Scripting
3	SSL v2 support detected
4	SSL Weak Cipher Suites Supported
5	Invalid SSL certificate chain
6	Missing Secure Attribute in an Encrypted Session (SSL) Cookie
7	SSL and TLS protocols renegotiation vulnerability
8	PHP 'strrchr()' Function Information Disclosure vulnerability
9	http TRACE XSS attack
10	OpenSSL 'bn_wexpend()' Error Handling Unspecified Vulnerability

to the use of fault injection in dependability benchmarks, although adapted to the security field: inject root causes (fault → vulnerabilities) to accelerate the occurrence of problems (failures → security violations) when exercising the system (workload→ workload with attacks) to assess the impact to other components in the system and evaluate the how the system reacts.

It is expected that the component with injected vulnerabilities become compromised during the execution of attacks. That, however, does not (or should not) automatically translate into a compromised benchmark target (the whole system or another component), and that is what the benchmark evaluates in this step. Figure 3-2 illustrates the notion of injecting and exploiting the vulnerabilities of a component outside the benchmark target. In this picture, it is possible to see that while attacks are conducted against the vulnerable component, security measurements are collected from the benchmark target, which is not expected to have vulnerabilities.

The idea of conducting attacks in a component that is outside the benchmark target is inspired in classical works of dependability benchmarking. In (J. Duraes and Madeira 2004), the authors injected faults in one component (termed as fault injection target) to evaluate the impact on another component (the benchmark target) or in the overall system, benchmarking the tolerance of widely known web servers using the fault injection technique. The authors also emphasize that in a scenario of Component Off The Shelf system integrators may want to know the impact to a component when hidden faults are activated on another component. As can be seen, this is exactly the approach that we are adopting in our security benchmark methodology. However, we also take advantage of the attack injection technique proposed in (J. Fonseca, Vieira, and Madeira 2009) as a guideline to

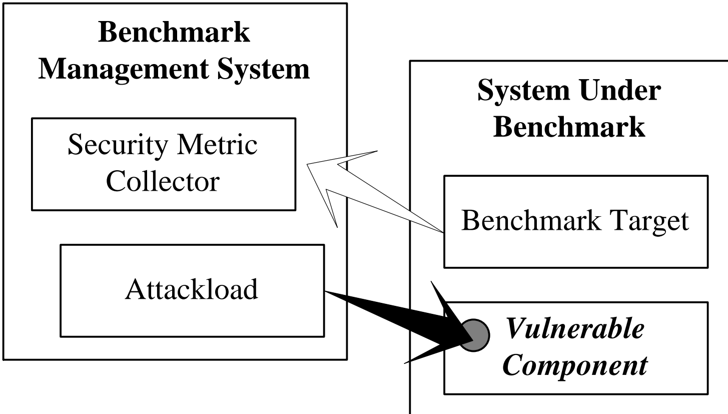

Figure 3-2. Attacks executed against a Vulnerable Component

mount the attacks of our security benchmark methodology. Attack injection consists in injecting vulnerabilities into a particular system component that are later exploited during the benchmark run.

One important aspect of the attack injection approach is the representativeness of the vulnerabilities that are injected. These vulnerabilities should reflect the common security issues that happen in the real world. It makes no sense to inject vulnerabilities that are rarely seen in real component, unless they represent a major security threat to the system (for example, a vulnerability classified with the highest risk score per the Common Vulnerability Scoring System). Additionally, this is particularly relevant to define where a vulnerability is located and what is the piece of code/configuration that corresponds to a vulnerability. In this sense, our methodology relies on vulnerability field studies to identify and characterize real security issues present in the system under benchmark. The benchmark implementer is allowed to take advantage of existing vulnerability field studies targeting the components of the system under benchmark. This is important to reduce the time needed to build a representative set of attacks, as the conduction of comprehensive filed studies is a very time consuming task, as it may require the analysis and comparison of vulnerable and patched code/configuration to exactly map the piece of software that needs to be injected. An example of field study that targets web application vulnerabilities is presented in (J. Fonseca, Vieira, and Madeira 2009), where the authors classified 655 cross-site scripting and SQL injection vulnerabilities present in different web applications.

The approach to measure the security risk is a key aspect of the dynamic part. We firstly run the system without executing attacks to observe its normal behavior, collecting measurements about the expected system response and response time. This execution is termed here as *baseline run*, as the purpose is to use these measurements as a comparison point to determine the extent of the impact of attacks. Then, we run the benchmark executing attacks against a component different from the benchmark target. For each vulnerability targeted by these attacks, we take into account the impact caused by the exploitation of vulnerabilities over the benchmark target and the easiness of exploiting them.

The attack impact over the benchmark target is evaluated by checking the system response to a set of tests. The purpose of these tests is to verify if there was any compromise of the security attributes (confidentiality, integrity, availability) during the execution of attacks. In fact, the implementation of these tests should comply with the criteria defined by the CVSS to the assessment of the impact of vulnerability exploitation. Here we provide some examples of these criteria,

which are detailed later on in the specification in Section 3.6.7. If the benchmark target becomes unresponsive for a limited period of time, then there is a partial impact on system availability. It is a complete availability impact if the benchmark target becomes totally unresponsive. The same approach is taken to test the ability of the system to keep restricted content protected (confidentiality) and to deliver the expected response during the attack run (integrity).

The probability of the occurrence of the attack is measured based on the easiness of attacking a vulnerability, following the criteria of CVSS as reference. In the dynamic part, the probability of vulnerability exploitation can be manually estimated prior to the execution attacks, while mounting the attacks to be executed against the target system. The highest probability (100%), and according to CVSS, should be assigned for attacks of low complexity, which require no authentication and can be executed remotely.

In the dynamic part of the benchmark, each benchmark run concerning one given system and one type of attack is executed at least three times to accommodate slight variations of the values measured and obtain a stable metric value. The resulting value is the average of the values. Equation 5 shows this computation (VRi is the risk measured during run i, and Nbr is the number of runs for that system and attack type).

$$\sum VRi \Big/ Nbr \tag{5}$$

Equations 6 to 8 present the formula to estimate the System Security Risk of the Dynamic Part (DSR) for each risk category (DSRL – Low Security Risk; DSRM – Medium Security Risk; DSRH – High Security Risk). This consolidation is very similar to the one already explained in the static part, summing the security risk obtained for each component (CRd – Component security risk of the dynamic part). It also has a weight for each component (*Wc*) to reflect its relevance to the whole system. The "*i*" index also means that the security risk shall consider each vulnerability detected during the dynamic part run.

$$DSRL = \sum_{i=1}^{n} (CRLdi * Wci) \tag{6}$$

$$DSRM = \sum_{i=1}^{n} (CRMdi * Wci) \tag{7}$$

$$DSRH = \sum_{i=1}^{n} (CRHdi * Wci) \quad (8)$$

3.5 BENCHMARK COMPONENTS

The security benchmark methodology is formed by components that together will measure the security risk of the system under benchmark. The components that are common to both benchmark parts (static and dynamic) are described next. The guidelines to build each one of these components are described in the next section.

Metric: characterizes the security of the system considering the risk posed by known vulnerabilities and the effects of unknown vulnerabilities. As described in Section 3.2, this metric (termed here as *SBench*) is estimated by the weighted sum of the security risk resulting from the static and the dynamic parts of the benchmark, considering four elements: vulnerability risk (VR), risk category, component risk (CR) per category, and weighted system risk (SR) per category (Low, medium, and high). This categorization is needed to enable the benchmark user a more complete understanding of the contributions to risk from vulnerabilities in each risk category (SRL, SRM and SRH, Equations 9 to 11). Finally, the benchmark metric value *SBench*, representing the *system security risk* is obtained by the weighted sum of the security risk of each risk category (Equation 11). This value represents the overall system security risk, including the contributions from vulnerabilities from all the risk categories, both from the static (SSR) and dynamic parts (DSR), and all the components of the system analyzed. The higher is the security risk of the system, the higher the benchmark metric is, with no upper limit. The weight of each risk category was defined to help remove ambiguity in the comparison of systems having similar overall benchmark results in scenarios of *one high-risk vulnerability versus many low-risk* vulnerabilities: the high-risk category weight is nearly three times larger than the medium category and fifteen times larger than the low category, increasing the influence of high security risks in the benchmark metric. It is worth noting that, in a future proposal of a security benchmark standard based on our methodology, the decision on the weights should be part of a technical agreement among industry and stakeholders. Technical agreements are quite common in performance benchmarks such as TPC and SPEC and our metric estimation approach should be seen as a contribution in this direction.

$$SRL = SSRL + DSRL \quad (9)$$

$$SRM = SSRM + DSRM \quad (10)$$

$$SRH = SSRH + DSRH \quad (11)$$

$$\boldsymbol{SBench = SRH * 0.70 + SRM * 0.25 + SRL * 0.05} \quad (12)$$

Procedures and Rules: guide users to run the benchmark and ensure that it is not twisted to favor a particular brand or vendor. Rules determine what is allowed and not allowed in the security benchmark. They are comprised in directives that show to users the conditions to collect security metric results, to build analytical and experimental setup, to enforce that the proper system characteristics will be disclosed to community, among others. Section 3.6 presents the guidelines to build the benchmark components (also termed here as component *specification*). In the next chapter, we approach the procedures and rules to execute the benchmark.

Instrumentation: include all the mechanisms and tools used in the context of the benchmark, from the benchmark execution to the estimation of benchmark metric. In the next Chapter, we present more details about our case study instrumentation that can be adapted for other domains.

The benchmark components of the static part are described below.

Vulnerability Information: refers to the whole set of data used to report and characterize a vulnerability including CVE-ID, vulnerability description, impact, affected systems and versions. These are collected from sources such as public vulnerability databases that are later used to identify known vulnerabilities in the system under benchmark. Public vulnerability databases (e.g., (US-CERT 2014; NVD 2014; OSVDB 2014)) usually contain most of these vulnerability characterization and, more importantly, the CVSS impact and exploitability information that we use to calculate the vulnerability risk of known vulnerabilities (with the proper adjustments detailed in section 3.3). The instrumentation components that work with the Vulnerability Repository:

Security Test Information: refers to the set of data to characterize security tests that can be used by a tool to confirm the existence of a given known vulnerability in the system under benchmark. These set of data include the following information: test description, affected system, affected component, affected versions, affected platform, CVE-ID, security practice, technical command, CVSS impact and CVSS exploitability. This information should be collected from security testing tools (e.g., vulnerability scanners) targeting the system under

benchmark. The purpose is to cover vulnerabilities that eventually were not reported in public vulnerability databases and, henceforth, are not present in the Vulnerability Repository. One justification to add information form security test is given in (Attrition 2008), where the author describes the difficulty in keeping vulnerability databases up to date due to the high amount of disclosures and staff restrictions.

The benchmark components that are used in the dynamic part are described next.

Workload: represents the load submitted to the system under benchmark and is composed by representative tasks for the class of that system. The workload is an essential component of benchmarks in general. In the context of pure performance benchmark, it is directly used to derive the metric values. In dependability benchmarks, it is required to exercise the system and to observe its behavior when specific areas of the system are activated (e.g., those that may contain faults). In our work, the need for a workload shares similarities with dependability benchmarks: we require that specific areas of the system be activated to expose (and make available to attacks) possible vulnerabilities existing in those areas.

Vulnerabilityload: refers to the set of vulnerabilities identified and coded to enable successful attacks during the benchmark execution. More specifically, represents changes in a component code/configuration to a vulnerable state for the same class of system components (for example, web applications written in PHP language). This is particularly useful to speed up the injection of vulnerabilities for components written using the same programming language. The approach to identify and inject these vulnerabilities are adapted from (Jose Fonseca 2011) and is comprised in the following steps: 1) select the field study containing the representative vulnerabilities to be injected; 2) identify the code changes that are necessary to make the code vulnerable (this may include comparing the vulnerable code with the patch applied to fix the vulnerability); 3) mount the set of pairs (termed in (Jose Fonseca 2011) as *vulnerability operator*) with the place in the source code where the vulnerability is likely to be found (*location pattern*) and the code change; 4) perform a static analysis in the source code of the component; 5) find the location where the vulnerability may exist; 6) mutate the code to inject the vulnerability 7) compile the code with the injected vulnerability and deploy it inside the system. In the implementation of the benchmark provided in Chapter 5, we present a concrete example to guide benchmark users on how to build a representative vulnerability injector. In the specification of the vulnerability injector provided later on, we detail each element of this component that should be taken into account. The implementation of the vulnerabilityload may vary according to the nature of the system domain.

For software written in C language, this refers to the change made in the source code to allow, for example, the execution of buffer overflows attacks. For software written in PHP language, the validation of input parameters should be weakened to allow the execution of Cross-site scripting and or SQL Injection attacks (we provide an example using this system domain in our benchmark implementation, Chapter 5). The same rationale applies to the configuration of a component. For software written in Java, the vulnerability injector can alter the structure of XMLs configuration files to force an unsecure state inside a given component.

Attackload: includes a representative set of attacks that are executed against the system under benchmark to measure their security risk in an experimental way. The attackload is built based on the vulnerabilities present in the vulnerabilityload meaning that the attacks are directed at those vulnerabilities. For example, if a vulnerability weakens the validation of input parameters of the target component, then the attacks must exploit this vulnerability, by sending data tailored to exploit that lack of validation and attempt to break some security property, depending on the context of the location of the vulnerability. In this sense, each attack represents an exploit of the vulnerability to be injected in the vulnerable component. The rules about the diversity and number of attacks are detailed later on. At this point in time, it is important to clarify that each attack implementation shall be done following the same approach of the attacks against the system interface. In other words, the attackload shall be built according to the attack patterns provided in the Common Attack Pattern Enumeration and Classification, which provide implementation examples and guidelines of a variety of attacks targeting different software classes.

3.6 BENCHMARK IMPLEMENTATION SPECIFICATION

This section presents the requirements to build our benchmark components. This is aimed at guiding benchmark users to properly implement our security benchmark methodology to any class of software systems.

One important aspect of this specification is the effort needed to implement the benchmark, especially to give to benchmark community an idea of the time and resources needed to define, develop, and deploy the benchmark. The implementation effort is independent from the benchmark execution and it is an one-time task. In the benchmark implementation we provide in Chapter 5, it was possible to build the components of the static part and dynamic part in 1 month, with one developer working full time. This topic will be better detailed later on and the purpose here is to point out our concern of defining requirements that

makes our methodology feasible from an implementation standpoint.

Another matter worth discussing is the effort regarding the benchmark execution. The time needed to execute the benchmark and get the results should be as short as possible, being crucial to the acceptance of the benchmark by the community. The background here is that, in general, benchmark users are not willing to wait for weeks to have benchmark results, especially in a scenario where they may have been pressured to make fast decisions to buy a secure software, to patch a system, to develop and deploy additional countermeasures, to increase the security team, among others. In the case study we present in Chapter 6, we discuss in more details the effort needed to execute the security benchmark we built targeting widely used web serving systems.

3.6.1 Metric Calculator

The component that measures the benchmark metric is a calculator present both in the static and dynamic parts. Figure 3-3 depicts the main elements of this calculator (vulnerability risk, component risk, and system risk calculators). The central point here is that security risk shall be estimated based on the risk of individual vulnerabilities detected during the benchmark run, considering the importance of each component and the three categories of risks. In Figure 3-3, vulnerabilities are represented by the blank circles present in the different components of the system under benchmark (left side). Then, the benchmark risk calculator, and based on the vulnerabilities detected during the benchmark run, estimates the risk of vulnerabilities of each component and then provide an overall estimation of system security risk, which is in fact the benchmark metric.

The general requirements to build the Benchmark Metric calculator is as follows:

- It shall have three components: the Vulnerability, the Component, and the Security risk Calculator.
- The equation to estimate the benchmark metric is the one provided in section 3.5.
- There is no restriction regarding the time needed to execute the metric calculator. However, it is expected to have this completed in a short time so that this does not impact the overall execution of the benchmark run. In the case study we provide in Chapter 6, this step, and for each benchmark run, was completed in 10 seconds.

The requirements for each one of the Benchmark Metric Calculator components are as follows:

- The component weight provided by the benchmark users shall be taken into account. If the benchmark user provides no weight information, all

components shall be considered as having the same level of importance.

- The component risk estimation shall reflect the categorization provided by the Vulnerability Risk Calculator. In Figure 3-3, components are represented by the CR boxes and provide the input for the system risk estimation of each benchmark part.
- The Security Risk Calculator shall estimate the security risk of the system by adding the security risk of the static and dynamic parts. This sum shall be done separately for each category of risk.
- The Security Risk Calculator shall assign a different weight factor for each category prior to the final security risk sum. The highest factor should be given for the high-risk category, followed by the medium and low-risk categories. The values we recommend were already provided in Section 3.5 and are respectively 0.70, 0.25, and 0.05.
- The Security Risk Calculator shall calculate the system security risk as a numeric value, with no upper limit. Given that this refers to security risk, the higher the numeric value, the most insecure the system is.

3.6.2 Vulnerability Repository

The Vulnerability Repository stores and maintains information about known vulnerabilities targeting software-based systems. This repository shall be built in the form of a relational database. This is justified by the fact that several vulnerability databases are already built using this format and relational databases allows the execution of SQL queries, widely used by the developer's community across the world.

The data required to support the vulnerability repository involves the following

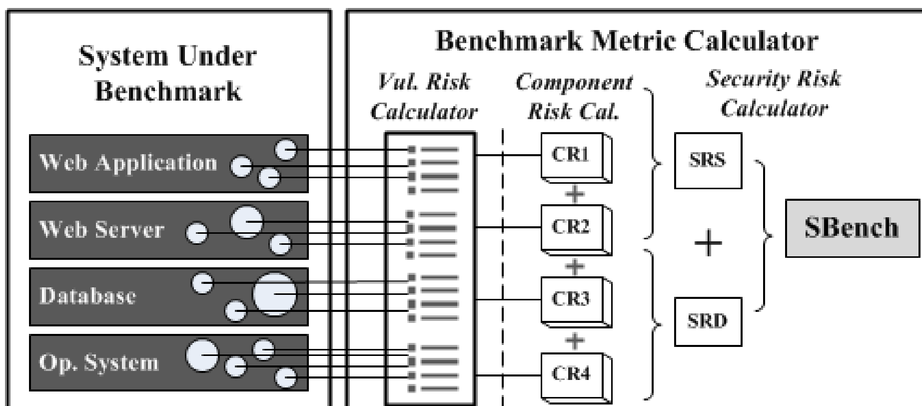

Figure 3-3. Security Risk Calculator

entities: Vulnerability, Vulnerability Risk, Category, System, Version. The Vulnerability entity represents the characterization of each vulnerability, which is described by the following properties: vulnerability ID, description, solution, and dates (e.g., disclosure, discovery, exploit, and solution date). The Vulnerability Risk represents information about the impact and exploitability of each reported vulnerability. The Category entity represents the classifications of each vulnerability. The System and Versions entities represent respectively the system and version where the vulnerability has been identified. Each occurrence of Vulnerability is related to one Vulnerability Risk, one Vulnerability Category, one System and one Version.

The resulting relational schema of the vulnerability repository is depicted in Figure 3-4. This model is a simplified version of the one developed by the Open Source Vulnerability Database, which currently hosts more than 120 thousands vulnerabilities (data collected in September, 2015). A brief description of each one of these entities is as follows:

- The "Vulnerability" table is the main entity in this data model and is mandatory, hosting vulnerability ID, description, solution, and dates (e.g., disclosure, discovery, exploit, and solution date).
- The "CVSS_Vulnerability_Risk" table stores information about the impact and exploitability of each reported vulnerability, following the criteria defined in the Common Vulnerability Score System.
- The "Category" table describes the classifications considered inside the database model. As not all vulnerabilities are classified by vulnerability databases, this is an important entity.
- The "Vulnerability_System" table links system, version and vulnerability together.
- The "System" table hosts product names targeted by the vulnerability report (e.g., Windows, Exchange, Apache, and MySQL).
- The "System Version" table stores the version names of the products where the vulnerability was discovered (e.g., 1.0, 2.0, 0.1, XP, 2000, and 95).

The requirements to build the benchmark Vulnerability Repository are as follows:

- The Vulnerability Repository shall be created preferably using a database management system. Using an external RDBMS allow better interoperability with other data repositories, and alleviates the effort

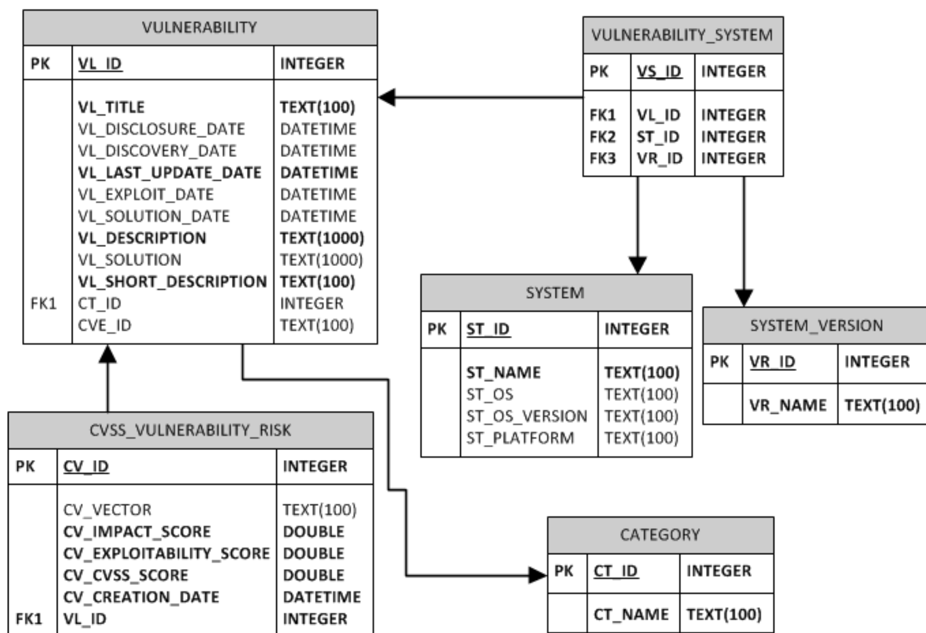

Figure 3-4. Vulnerability Repository Data Model

needed to implement tools (in fact, it would make no sense to implement a custom data manager because common available RDBMS provide an optimized and standardized data management. The selection of the DBMS is left to the benchmark user.

- The main entity of this database is the one that hosts information about each vulnerability reported in the field (vulnerability table).
- The minimum information (for each vulnerability) required to build the vulnerability repository is as follows: id (e.g., CVE ID), release date, description, affected system, affected components, affected versions, affected platforms, CVSS impact, CVSS exploitability.
- The vulnerability data shall be stored in a database repository using the Entity-Relationship data model (Chen 1976), where Vulnerability represents the main entity and each vulnerability ID represents a unique record. A comprehensive example of vulnerability data model was proposed by (OSVDB 2014). We used that data model in the instantiation of this methodology for web serving systems, which is described in Chapter 5.
- Once the vulnerability repository is implemented, we estimate the

maximum time needed to the automated collection of vulnerabilities from a typical web serving system is about 2 hours. This time was defined based on the case study we describe in Chapter 6, already considering additional time for the case of more complex target systems or for the systems to which more vulnerabilities are available in the field. In fact, with the automation tools we built in our methodology implementation (described in Chapter 5), this was done in less than 15 minutes targeting 304 vulnerabilities across 6 benchmarked systems.

The following are important characteristics that should be taken into account during the definition and development of a vulnerability repository:

- Each record in the vulnerability entity corresponds to a vulnerability report.
- Each vulnerability report describes one and only one vulnerability.
- A vulnerability report containing multiple vulnerability occurrences should be either separated (one vulnerability per report) or ignored in the benchmark run.
- The impact and exploitability of each known vulnerability shall be obtained from public vulnerability databases (CVE 2014; NVD 2014; OSVDB 2014). This is important to estimate the individual risk of discovered vulnerabilities.

3.6.3 Security Test Repository

The Security Test Repository contains – in the form of technical commands ready to be executed – the security tests that shall be run by a testing tool to confirm the presence of known vulnerabilities in the system under benchmark. In Chapter 5, we provide examples of tests conducted against web servers consisting in a HTTP command that seeks for a given file in a given web server directory.

The data required to support the security test repository involves the following entities: Security Test, Vulnerability, Vulnerability Risk, System, Version. The Security Test entity represents the characterization of each test to be executed and is described by the following properties security test ID, programming language, command, creation date, operating system, operating system version. The Vulnerability and Vulnerability Risk entities are similar to the ones described in the previous section. The System and Versions entities represent respectively the system and version where the vulnerability has been identified. Each occurrence of a Security Test is related to one Vulnerability, one Vulnerability Risk, one System, and one Version.

The resulting relational schema of the Security Test Repository is depicted in Figure 3-5 and has some similarities with the Vulnerability Repository data model in what regard the vulnerability entity. This is needed to associate each security test command with a vulnerability ID, which has the values of impact and exploitability that will be used to estimate the vulnerability risk for each positive test (i.e., the vulnerability was found in the target system). The entities of this data model are described as follows:

- The "Security Test" table is the main entity in this data model, hosting security test ID, description, commands and links to test installer (if applicable), or binaries (if needed).
- The "Vulnerability" table stores the vulnerability ID, description, solution, and dates (e.g., disclosure, discovery, exploit, and solution date). This is needed in the data model as each security ID has to be an associated vulnerability ID.
- The "CVSS_Vulnerability_Risk" table has the same specification already described in the previous section, holding data about the impact and exploitability of each reported vulnerability, according to the Common Vulnerability Score System.
- The "Test_System" table links test, system, and version.
- The "System" and "System Version" tables have the same specification already presented in the Vulnerability Repository.

The requirements to build the benchmark Security Test Repository are described as follows:

- It shall be created using a database management system and the selection of the DBMS is also left to the benchmark user.
- The main entity of this database is the one that hosts information about each security tests and associated vulnerabilities.
- The minimum information required to build a security test repository is as follows: test ID, description, affected system, affected component, affected versions, affected platform, CVE-ID, security practice, technical command, CVSS impact and CVSS exploitability. Please note that the repository shall contain not only the description, but also the commands that test the presence or absence of a known vulnerability. This is needed to speed up the implementation of these tests.

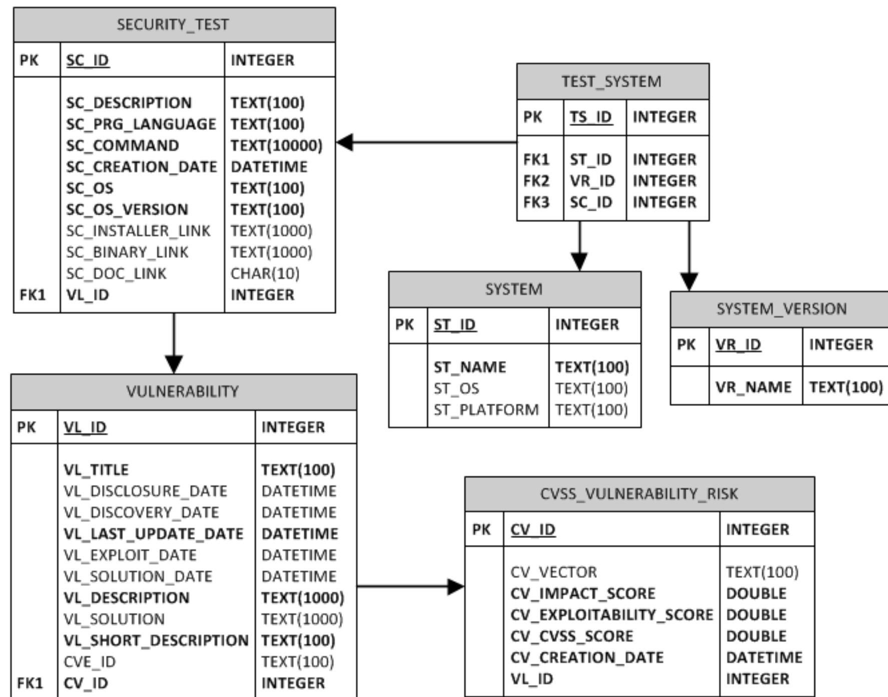

Figure 3-5. Security Repository Data Model

- It shall contain security tests covering the systems and components under benchmark.
- It shall contain tests that do not change the behavior of the target systems (termed here as non-intrusive security tests). For example, if for a certain system a known vulnerability is given by the presence of a certain file, then the security test consists in simply identifying the presence of this file.
- If the vulnerability risk of a given security testing is available in a public vulnerability database, the impact and exploitability factors shall be taken from these databases. Otherwise, CVSS criteria shall be followed to estimate the impact and exploitability of the vulnerability associated to the implemented security test.

Our methodology allows the use of existing tools as an implementation of the security test repository. The rules to select a representative security tool is as follows:

- It shall have a comprehensive set of tests to identify known vulnerabilities - additional tests from other tools can be incorporated in order to reach a

representative range of tests and cover a large amount of known vulnerabilities;

- It shall execute non-intrusive tests against the system. The goal is just to check if a given vulnerability exist, and not to execute attacks that are covered in the dynamic part
- It shall allow the development of new features to automate the estimation of the benchmark metrics described in section 3.5.
- The maximum time recommended to the automated verification of vulnerabilities by the security testing tool is 2 hours[1] per target system systems. In the case study we present in Chapter 6, this stage was done in less than 30 minutes considering 4 systems under benchmark and 6456 security tests.

3.6.4 Workload

The workload represents the typical work that the system under benchmark executes. This means that a workload should be considered based on its abilities to simulate in a real way the set of tasks that are submitted and executed by the target system.

The development of a workload from the scratch is a complex task. If the system to be benchmarked is a database management system, this means that benchmark implementers should code a program that simulates the typical DBMS operations. This is strictly required only if there is no representative workload available in the field for the software-based system that one wants to benchmark. In fact, if there is already a benchmark for the class of system to benchmark (e.g., a performance benchmark) having a workload already accepted by the industry, then it is desirable to adapt and use that workload (if possible and if it is representative) as it avoids effort and gains the acceptance already given to that workload. From the perspective of our security benchmark methodology, and given the myriad of workload available that could be adapted for different benchmarking purposes, the most important aspect resides in the rules to select a representative workload. These rules, that are inspired in classical performance benchmarks such as TPC and SPEC, are as follows:

- It shall simulate a realistic execution of the system under benchmark (including user interaction);

[1] This time of 2 hours is obviously postulated. However, based on the experiments done for the present work this time is enough to accommodate typical systems. At the same time, 2 hours represents an acceptable effort for most benchmarking scenarios.

- If the system runs in a client-server environment, it shall simulate client request using loads of different sizes and using different frequencies.
- It shall allow the development of new features (to collect benchmark measurements and to incorporate the attackload components);
- It shall be executed in a short period of time, having a maximum time to complete its operations (execution time), with upper limit of 30 minutes per benchmark run. The logic behind this decision is the result obtained in the case study we conducted, where each benchmark run took less than 5 minutes to complete, allowing us to execute 25000 attacks on 6 systems in 24 hours. Without this execution time restriction, benchmark usability will be compromised, since end-users want to get fast results when benchmarking systems.
- It shall have a ramp-up and ramp-down time. The maximum time recommended for each one of the ramp-up and ramp-down phases is 5 minutes, also based on the experience we had with our case study. This is the time needed to the execution of the benchmark without the conduction of attacks and collection of security measurements. Without this time defined, the system under benchmark will not be working in a stabilized way (e.g., stable memory and process consumption) and the benchmark results will be either biased or compromised. In other words, benchmark measurements shall be only collected once the system reach a stable state, avoiding the performance peaks during its startup and shutdown.

3.6.5 Vulnerability Injector

The vulnerability injector of our security benchmark methodology is based on the vulnerability injection methodology proposed in (J. Fonseca, Vieira, and Madeira 2009). The prerequisite here is that the *source code* and *configuration files* of the target component shall be available. In case source code is unavailable, the benchmark remains useful, as the attackload will target only vulnerabilities present in the system interface, with no need of injecting malicious code inside the system component. These are the rules to build the vulnerability injector:

- It shall provide the following functionality: (1) get the source code and or configuration files of the system component where vulnerabilities will be injected, (2) change the code and or the configuration of the component to an unsecure state (this is when vulnerability injection happens), and (3) turn the vulnerable component back to the system.
- It shall inject only representative vulnerabilities for the system under

benchmark. To obtain a representative set of vulnerabilities two approaches can be adopted:

Field study on the vulnerability field. Benchmark implementers shall take advantage of studies that already collected, analyzed, and characterized real vulnerability codes from several versions and brands of the targeted components. An extensive field study mapping software faults to security vulnerabilities is described in (J. Fonseca and Vieira 2008), analyzing 655 security patches of widely used web applications.

Analysis of known vulnerabilities. Vulnerability information collected from vulnerabilities databases can be used to identify the most representative vulnerabilities reported in a given system component. The challenge here is to obtain access to the system source code when it is unavailable in the vulnerability report. What one can do is to take note of the affected version and configuration, download and analyze the source code. Obviously, this is just possible when benchmark users have full access to the source code of the component or systems under benchmark.

- To work in accordance with the expected functionality, it shall be formed by the following components (adapted from (J. Fonseca, Vieira, and Madeira 2009)):

  Vulnerability Operator. This component shall store the location pattern and vulnerability code/configuration change. The location pattern characterizes the place in the source code where the vulnerability is likely to be found. The vulnerability code change defines what has to be done to the piece of code targeted by the location pattern in order to make it vulnerable, without affecting the functional behavior of targeted component. Both the location pattern and the vulnerability code are identified as a result of the field study conduction described earlier.

  Content Collector. This component shall collect the configuration and code files of the system component to be targeted by vulnerabilities (vulnerable content) and deploy it into the *vulnerability injector workspace* (a place in the benchmark management system to the manipulation of the component contents). Examples of contents are the source code of web applications (Java file, PHP files, etc.) and configuration files of application servers (e.g., .xml, .properties files).

  Injection point locator. This component shall localize where configuration and software fault vulnerabilities can be injected in the collected contents based on the information provided by the Vulnerability

Operator. Examples of injections points are described below.

Content mutator. This component shall change a configuration or a piece of code to a vulnerable state defined by the vulnerability operator. For example, an input variable that is correctly escaped could be mutated to allow Cross-site script or SQL injection attacks. An important aspect is that these code changes should not prevent the component from running properly. In other words, even after injecting the vulnerability, the end user shall be able to execute all the component features.

- The injection shall be done in a component different from the benchmark target (for the reasons described earlier).
- There is no restriction regarding the time needed to the injection of vulnerabilities since it is expected to be conducted prior to the benchmark execution.

Table 3-2. Missing Function Call Extended (Jose Fonseca 2011) presents one of the vulnerability operators described in (Jose Fonseca 2011). This vulnerability operator can be used to locate Cross-site scripting vulnerabilities in web application code written in PHP language. This vulnerability operator was proposed after a field study mapping software faults and security vulnerabilities (J. Fonseca and Vieira 2008). An extension of this study to Java-based web application is presented in (Seixas et al. 2009). In these fields studies, the authors analyzed the software faults reported as a vulnerability (by comparing the vulnerable code with the fix pack) following the procedures and fault categorization proposed by (J. A. Duraes and Madeira 2006). The list of software fault categorization used in these works is presented in Table 3-3. This list is not a compilation of faults that will necessarily lead to vulnerabilities. The intent here is to present the faults types that are representative of open-source software faults and that could be injected if they make the system behaves in an insecure way.

3.6.6 Attackload

The attackload is used during the workload execution by attacking the system while it performs a set of common operations. This is done by emulating malicious users attempting to exploit vulnerabilities present in the *system interface* and in a component outside the *benchmark target* (those injected by the vulnerability injector). While attacks are conducted, the security checker (which is described in the next section) observes any alteration in the behavior of the system (system response correctness, system availability, restricted areas access) and provides the input needed to the estimation of the security risk of the

benchmarked system.

The attackload applied in the context of security benchmarking is one of the key contributions of our methodology. This allows the assessment of security in an experimental way by stressing the systems with real attacks. There are two important aspects that are worth describing prior to getting into the attackload requirements: the representativeness of the attacks and their implementation. The

Table 3-2. Missing Function Call Extended (Jose Fonseca 2011)

Vulnerability Operator Attribute	**Attribute restrictions and actions**
Location Code Pattern	Locate a function with the following characteristics: - The function must be the (int) type cast or it is the intval PHP function. - The argument of the function is directly or indirectly related to an input value from the outside: POST, GET, the return of a SQL query. - The output of the function is going to be displayed on the screen or is going to be used in a POST, a GET variable or is going to be used in a SQL query string. - The function can be an argument of another function or have another function as the argument. - In the argument of the function, the vulnerable variable may also be present inside a $_GET, $HTTP_GET_VARS, $_POST, $HTTP_POST_VARS PHP variable arrays.
Code Change	If the function is used in an assignment as the only line of code and the variable is not inside $_GET, $HTTP_GET_VARS, $_POST or $HTTP_POST_VARS PHP variable arrays the whole line of code is removed. For example, remove the line: $vuln_var = intval($vuln_var); - If the function is used in an assignment as the only line of code and the variable is inside $_GET, $HTTP_GET_VARS, $_POST or $HTTP_POST_VARS PHP variable arrays only the function is removed from the code, leaving the argument intact. For example, replace: *$vuln_var = intval($_GET['vuln_var']);* with *$vuln_var = $_GET['vuln_var'];* - In the other cases only the function is removed leaving in the code only the variable, or the *$_GET, $HTTP_GET_VARS, $_POST, $HTTP_POST_VARS* PHP variable array if the variable is inside. For example, replace: *... "'str1'.intval($vuln_var).'str2'";* with *... "'str1'.$vuln_var.'str2'";*

rules for executing the attacks are described in the next chapter; and the assessment of attack impact that are used as input for the security risk estimation is presented in the next section.

To provide useful results, the attackload should focus on attacks that are commonly found in the real world. To build a set of representative attacks, benchmark implementers should look at the representative vulnerabilities targeting the systems under benchmark. The logic behind this reasoning is that a successful attack necessarily exploits a vulnerability that exists on the system. By mapping the common vulnerability first, it becomes easier to decide which attacks should be addressed by the attackload. Taking web applications written in PHP as example, the improper validation of input parameters is the top vulnerability found in the field (J. Fonseca and Vieira 2008), having the SQL Injection and Cross-site Scripting as representative attacks.

The implementation of the attacks is a real challenge. Not all benchmark implementers may have the technical skills to code and deploy the attacks needed to stress system security. Although we recognize the need of having programming expertise to complete this part of the attackload process, we do believe that there is sufficient information in the field to guide developers on how to code a successful attack. The first source of information, as already mentioned before, is the Common Attack Pattern Enumeration and Classification, which contains the attack execution flow, exploit examples, and methods for a myriad of attacks. For example, if the vulnerability (CWE-79 2014) is one of the most representative for a web-based system, then (CAPEC-18 2014)(embedding scripts in non-scripts

Table 3-3. Software faults types

Type	Description
MFC	Missing Function Call
MFC Extended	Missing Function Call returning the same data type as argument
MVIV	Missing Variable Initialization Using a Value
MIA	Missing If Construct Around Statements
MIFS	Missing If Construct Plus Statements
MIEB	Missing If Construct Plus Statements Plus Else Before Statements
MLPA	Missing small and localized part of the algorithm
WPFV	Wrong variable used in parameter in function call
WLEC	Wrong logical expression used as branch condition
EFC	Extraneous function call

elements) shall be used as a guideline for implementing the attacks. The second source of information is the exploits already available in the field. There are several open-source tools that could be used as source of information to build the needed attacks (Metasploit is one of the exploit tools widely used in the field (Metasploit 2015)). It is also possible to adapt penetration testing tools to fit in our benchmark process and take advantage of already developed tools to stress the security of our systems. In the case study we present in Chapter 6, we coded our own attacks, but it is worth remarking that our methodology has a high degree of flexibility, allowing implementers to reuse attacks originated from other tools.

The rules to define the attackload are described as follows:

- It shall be organized in a set of vulnerability-attack pairs. The pairs vulnerabilities-attack shall be chosen in accordance with a vulnerability field study (e.g., for web applications there is a field study available in (J. Fonseca and Vieira 2008) on the most representative vulnerabilities) as already described in the previous section.
- It shall be built with the components exemplified in Figure 3-6. These components were adapted from (J. Fonseca, Vieira, and Madeira 2009) and here they are used with the purpose of stressing the security of the system and collect metrics that will allow security comparison:

  Vulnerability Injector. This corresponds to the vulnerability injector described early. This component injects real vulnerabilities in a component outside of the benchmark target.

  Attackload Generator. This component generates the exploit (malicious

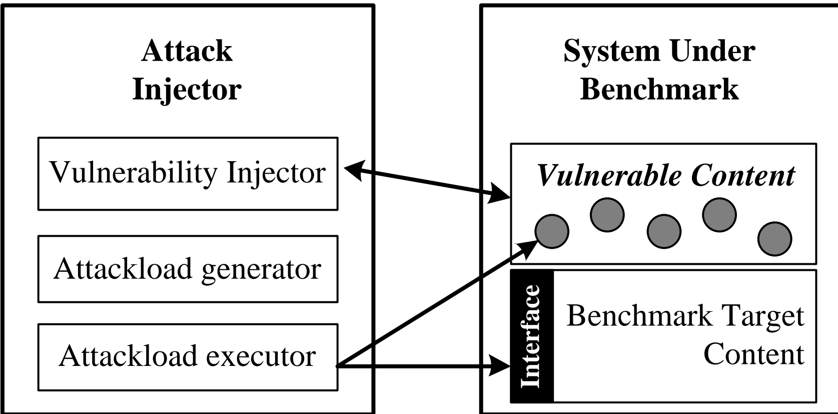

Figure 3-6. Attack injector components

interaction patterns and code) for each vulnerability injected.

Attack Executor. This component runs the exploit against the Vulnerable Component during the benchmark run.

- The attacks built by the Attackload Generator shall be implemented in accordance with the guidelines of the Common Attack Pattern Enumeration and Classification.
- It shall conduct attack against the component outside the benchmark target. As already described earlier in this section, the goal is to observe the behavior of the benchmark attacks when attacks are conducted in another component.
- It shall exploit only one vulnerability per each benchmark run. If more than one vulnerability is exploited, it will not be possible to determine which one compromised the security of the system and the vulnerability risk estimation will not be possible to compute.
- It shall execute multiple and concurrent attacks (targeting the same vulnerability). The purpose here is to verify how the system behaves when one vulnerability is exploited simultaneously by different attackers (simultaneous attacks may stress the system more than just one attack at a time, and the observation of this scenario may be relevant). At least 20 attackers shall be simulated, simultaneously executing 10 attacks per benchmark run. This number was defined based on the case study we conducted with a workload execution duration of 3 minutes. This is the base measurement that shall be considered. If the workload measurement interval is 10 minutes, for instance, then at least 40 attackers shall be simulated and so on.
- It shall support all attacks types identified during the vulnerability field study. A diversity of attacks shall be conducted in order to properly check the behavior of the system.
- The number of attacks for a given attack type shall be the same across all benchmark runs and target systems (note that the attackload is used during the dynamic part and it is not directly related to the number of discovered vulnerabilities for each system, which will be diverse).
- The attackload shall be conducted during the measurement interval of the workload, which means that it shall have a maximum time of 30 minutes

of execution per benchmark run[2], without considering the ramp-up and ramp down time (5 minutes each). This is done to observe the behavior of the system while attacks are executed and to set up a system state similar to those of real attacks in the operational scenario

3.6.7 Security checker

The Security Checker is aimed at verifying if the benchmark attacks were successful and at assessing the extent of the attack impact to the whole system. This is basically done by observing the behavior of the system considering the security attributes of confidentiality, integrity, and availability.

To properly evaluate the impact of attacks, the Security Checker compares system information collected during two independent benchmark execution stages. In the first stage, the benchmark is executed *without* attacks (termed here as *baseline* execution). From a benchmark standpoint, this means that the Benchmark Management System will start the system along with the benchmark workload (with no attack execution). This is a crucial step, as it enables to collect system metrics and output within a normal circumstance (when there is no actions to stress its security). More specifically, and for this scenario, the security check is executed to collect the following: the expected response of the system for each request made by the workload; the expected response time for each request and the average response time during the workload execution; the expected results when accessing restricted contents; the expected response when accessing restricted contents and so on.

The second stage is the execution of the system and of the benchmark by activating the attackload component. This means that the security checker collects information while attacks are being conducted the vulnerable target. Once each piece of information is extracted, the security checker performs a comparison with the information gathered in the first scenario, in order to measure the extent of the impact of attacks on the security attributed we already described. If there were limited to certain areas or data of the system, then it shall be considered a partial impact for a given security attribute; otherwise; it shall be considered a complete impact.

The impact assessment on system confidentiality shall be performed considering the capability of the system to keep information undisclosed (when and where needed) during the execution of attacks. One example of checking here is to

[2] Execution time recommended based on the on the case study presented in Chapter 6. It may vary depending on the characteristics of the system under benchmark.

monitor the access to restricted areas. If for some reason the system allows the access to restricted areas, then there was an impact on confidentiality. Another important example is to monitor the content of restricted areas. If there is more information than the one provided without the execution of attacks, then this means that there was an impact on confidentiality. This shall be done comparing the response to access restricted areas during the baseline execution (no attacks) with the attack execution.

The impact assessment on system integrity shall be performed considering the capability of the system to keep information protected against unauthorized modification (when and where needed). During the baseline execution of the system, the security checker shall get all response and output provided by the benchmark target. Then, this output shall be compared with the one generated during the execution of attacks. If there was any change to a limited number of information, then the impact on integrity was partial; otherwise, there was a complete impact.

The impact assessment on system availability takes into account the response time of the system by comparing the results of the baseline execution against the execution with attacks. The goal here is to monitor not only the impact on response time, but also the capability of the system and the benchmark target to handle the requests sent by the workload client. If the response time was not affected, but the benchmark target was unable to process the expected number of requests when compared with the baseline execution, there was a performance degradation from a client perspective in the sense that he or she did not receive the expected the response within the time expected. If the availability was impacted in a limited number of requests, then the impact was partial; otherwise, it was a complete impact.

The rules to build the Security Checker is described as follows:

- It shall have the following components (as depicted in Figure 3-7): the confidentiality checker, the integrity checker, the availability checker, the exploitability checker, and the data collector. These components are described later on in section 3.6.7.
- Each one of these components shall assess the impact extent of a vulnerability exploitation for each one of the security attributes according to the criteria provided in the Common Vulnerability Scoring System, already explained in Chapter 2 and specified in (Mell, Scarfone, and Romanosky 2007).
- The Confidentiality Checker verifies if there was partial or complete

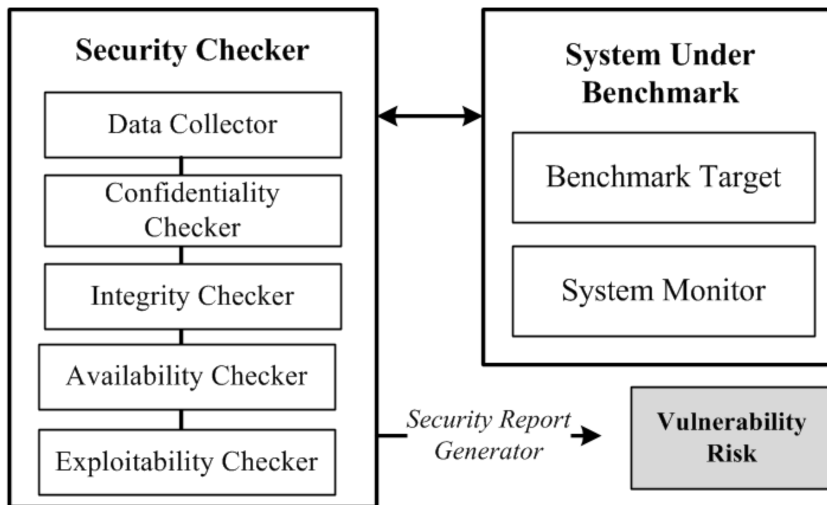

Figure 3-7. Security checker components

impact to system confidentiality. A partial confidentiality impact occurs if there is considerable information disclosure (partial loss of confidentiality). A complete confidentiality impact occurs if there is total information disclosure (complete loss of confidentiality). Table 3-4 presents the rules to assess the impact on confidentiality attribute according to the criteria of Common Vulnerability Scoring System. If at least one of these rules are meet (and this verification shall be implemented by the security checker), then there was either a partial or complete confidentiality compromise.

- The Integrity Checker verifies if there was partial or total impact on system data integrity. A partial integrity impact occurs if users are allowed to partially modify system component. A complete integrity impact occurs if users are allowed to completely modify system content. Table 3-5 presents the specific rules that shall be implemented in the security checker to verify if the integrity was compromised.
- The Availability Checker verifies if there was partial or total availability compromise during attack execution. Table 3-6 specifies the rules to verify if there was a partial or complete availability compromise, also following the CVSS specification.
- The Data Collector shall monitor and collect the following metrics from the benchmark target: response time, response correctness, access to restricted areas, and modification of restricted contents. These are the

Table 3-4. Confidentiality Impact Assessment Rules

Category	**Partial Impact**	**Complete Impact**
Disclosure	Access to some restricted contents is possible	All restricted and confidential contents were revealed after the vulnerability exploitation
System Data	The attacker is able to read a portion of system's data	The attacker is able to read all of the system´s data (memory, files, database records, etc.).
Restricted content	An unauthenticated user is able to access restricted content on the Benchmark Target	Any user is able to access restricted content on the Benchmark Target
Privilege elevation	An unauthorized user is able to execute authorized actions in the Benchmark Target	Any user is able to execute authorized actions in the Benchmark Target

metrics that will allow the security checker to observe the impact on the security attributes.

- The Report Generator shall provide the result of an attack execution in terms of vulnerability risk. If the attack was successful, it is necessary to check the level of impact and exploitability to the exploited vulnerability.
- Benchmark users are free to add other tests to check the impact on security attributes, with the condition of executing again the benchmark against the system under benchmark, and not changing the security tests across the entire benchmark run. This is needed to make sure that the tests will be the same and the results of the benchmark will not be biased.
- The time needed to execute the security checker, considering that this is fully automated, is the same of the attackload execution, which means that it will be executed in a maximum time of 30 minutes. The rationale of this number was already mentioned and is based on our experience with the case study presented in Chapter 6.

3.6.8 Exploitability checker

The Exploitability Checker assesses the easiness of exploiting a vulnerability considering the complexity of the attacks (low, high, medium), the layers of authentication needed (none, single, multiple), and the location from where the attack is executed (local, remote). The result of this analysis is used by the Metric Calculator to estimate the probability factor of the vulnerability risk equation. This calculator also translates the CVSS metric values that shall be used for each exploitability category.

The Exploitability Checker shall allow the manual input of exploitability factors

Table 3-5. Integrity Impact Assessment Rules

Category	Partial Impact	Complete Impact
Data modification	The attacker does not have control over what can be modified, or the scope of what the attacker can affect is limited.	The attacker is able to modify any files on the target system
Content overwritten	Content may be overwritten or modified in a limited way.	Any content may be overwritten or modified
Expected Response	The BT target is not providing the expected response for some requests.	The benchmark target is not providing the expected response for the system clients.

by benchmark implementers. For example, if all representative attacks are executed from a remote network, with no need of authentication, and with a low level of complexity, then these parameters can be manually set by the benchmark implementer and the exploitability checker has no need to perform any verification.

The exploitability requirements were described in Chapter 2 and are summarized as follows:

- The access vector shall be checked considering the network source of an attack (local, adjacent network, network). If a vulnerability can be exploited from a remote network, then it is easier to exploit than a vulnerability that can only be exploited with a local access into the system. If the vulnerability is easier to be exploited, then this means that the probability of exploitation is higher, which in turn will result in a higher probability value in the risk equation of the benchmark metric.
- The authentication shall be assessed considering the number of layers of authentication needed to conduct an attack (none, single, and multiple).

Table 3-6. Availability Impact Assessment Rules

Category	Partial Impact	Complete Impact
Response	The system or any resource provided by the system to the client became unavailable for a short period of time.	The system or any resource provided by the system became unavailable longer than the workload timeout. If this happened, then there was a complete loss of availability.
Trustable Host	A trustable host had its connection denied by the benchmark target.	All trustable hosts had their connection denied with the Benchmark Target.

The more authentication layers are needed, the more difficult is to exploit a vulnerability.

- The access complexity shall assess the difficulty that an attacker has to exploit a vulnerability, according to the CVSS categories that are exemplified as follows:
    - *Low complexity*: The attack can be performed with no automation, and little skill is needed to deploy the attack. The vulnerable configuration is default.
    - Medium complexity: Attacker shall gather some information before conducting a successful attack (e.g., social engineering may be needed). The vulnerable configuration is non-default.
    - High complexity: To exploit the vulnerability, high level of privileges on the vulnerable component is required. The vulnerable configuration is very rare.

3.7 CONCLUSION

This chapter presented our risk-based methodology to benchmark the security of any class of software-based systems. This general methodology is based on classical benchmark proposals in the field of performance and dependability, using components such as metrics, workload, procedures and rules, and experimental setup. This methodology uses the notion of risk in a quantifiable way and allows the comparison of functionally equivalent systems (or different configurations of the same system) to enable users and system integrators to identify and select the most secure one. Our benchmark metric is *SBench*, which is estimated independently for each benchmark part and is calculated by the weighted sum of the security risk of these parts.

Our security methodology takes into consideration both known vulnerabilities (i.e., those that were previously discovered for the target system and are known to the public) and unknown vulnerabilities (i.e., those that are not yet discovered). The risk posed by known vulnerabilities is estimated using a different process than the risk of unknown vulnerabilities. Consequently, our benchmark methodology is organized in two parts: one static part, and one dynamic part.

The static part measures the security risk based on the knowledge of known vulnerabilities. These vulnerabilities are identified considering two different sources. The first source refers to the vulnerabilities that are reported in the field (e.g. vulnerabilities databases). The second source refers to the execution of security tests aimed at confirming the presence of known vulnerabilities.

The dynamic part measures the security risk based on the behavior of the system under benchmark when facing realistic attacks. This is done by injecting vulnerabilities in a component different from the benchmark target (the vulnerable component) and attacking them, while collecting security measurements. The purpose of this vulnerability injection is to anticipate security breaches that may occur in another component and, from a security comparison perspective, evaluate the robustness of the benchmark target when attacks are directed against other components of the system.

The specification of benchmark components was also provided. The objective was to provide the rules to be followed when building each component of the security benchmark. The vulnerability repository and the security test repository of the static part should be built in accordance with the data model we described. The vulnerabilityload and the attackload should target the most representative security issues that happen in the real world. This is particularly important to make the results of the benchmark useful to the community. Furthermore, the purpose of this specification is to ensure that different teams following the same set of criteria will reach to a similar implementation. The effort involved in the building of the benchmark components will be subject of discussion in the case study we present in Chapter 6.

The security benchmark metric, the combination of the static and dynamic parts, and the specification we provided form the basis of our security benchmark methodology (the procedure on how to execute the benchmark are provided in the next chapter). To the best of our knowledge, this is a completely novel contribution to the security field, as our methodology proposes a risk-based metric to quantify the security level of systems (also allowing the breakdown of this metric for more detailed analysis) and has a complementary approach to cover known and unknown vulnerabilities (the static and the dynamic part) in a analytical and experimental way.

CHAPTER 4

4. BENCHMARK PROCEDURES AND RULES

This chapter presents the procedures and rules to run our security benchmark methodology. The purpose is to present the procedures to be used by benchmark implementers to prepare and execute the benchmark components described in the previous chapter. These procedures are essential to ensure that different users will use the security benchmark in a consistent and uniform way and produce results that are repeatable and comparable.

The reminder of this chapter is as follows. Section 4.1 provides a general description of the execution rules of the benchmark. Section 4.2 describes the instrumentation components of our security benchmark. Section 4.3 details the deployment procedure to install and configure the target system and the benchmark. Section 4.4 presents the procedures of the static part. Section 4.5 describes the procedures of the dynamic part. Section 4.6 details the procedures related to the benchmark metric. Section 4.7 presents the rules for the disclosure of the benchmark report. Section 4.8 concludes this chapter.

4.1 BENCHMARK GENERAL PROCEDURE

The general steps that are needed to build and execute our benchmark (execution profile) are described as follows:

1. **Benchmark implementation.** This consists in the implementation of the benchmark components, including the workload, the vulnerability repository, the security tester, the vulnerability injector, and the attackload that are described later on. The particularities to instantiate the

benchmark components to target the system of our case study are discussed in the next chapter.

2. **Setting the analytical and experimental environment.** This consists in the installation and configuration of software and network that will enable the benchmark run of the static and dynamic parts.
3. **Execution of the static and dynamic parts**. This consists of performing tasks that include the analysis of information from the target system, the extraction of vulnerability information and the execution of attacks. The final step of the benchmark run is the estimation of the benchmark metric. In Chapter 5, we show that most of these tasks can be automatized.

The execution of our security benchmark methodology is guided by the following rules:

- A benchmark run is organized in two parts: the static part and the dynamic part.
- The benchmark executer is free to decide which benchmark part will be executed first.
- The configuration of the SUB must be exactly the same in each phase of the static and dynamic part. The use of a configuration optimized for security in any one of the phases is not allowed.
- At the end of the execution of both parts, the Benchmark Management System (BMS) shall[3] calculate the final security risk based on the preliminary security risks of the static and dynamic part.
- The Static and dynamic parts shall be executed in an automated way. However, human interventions that do not affect the execution and results of the benchmark are accepted. An example of human intervention is the identification of the brands, versions, and configuration of the components running in the system under benchmarking. Another example is the manual injection of vulnerabilities in a component outside the benchmark target as the automated injections of vulnerabilities rely on tools that are usually hard to find and to develop.

It is worth noting that the duration of the static and dynamic parts depend on the assessment of the individual risk of the known and unknown vulnerabilities (those that were found by the execution of attacks) present in each System Under

[3] When stating the rules, we adopted a typical construction (shall) used in standard requirements of Software Engineering

Benchmark (SUB) component.

The rules specific to each benchmark part are described in the next sections.

4.2 BENCHMARK INSTRUMENTATION RULES

This section details the rules to implement the instrumentation that support the benchmark components described in Chapter 3. The benchmark and instrumentation components of our security benchmark methodology are depicted in Figure 4-1. The instrumentation components are described as follows:

Experiment Controller: This component is responsible for orchestrating the benchmark execution and shall be implemented according to the following rules:

- It shall manage the execution of the entire benchmark, starting and stopping the target system and benchmarked components in the proper time and order.
- It shall have the proper permissions to handle the system processes and benchmark components.
- It shall ensure that no process or services from the previous benchmark execution are being executed prior to the initiation of a new benchmark run.

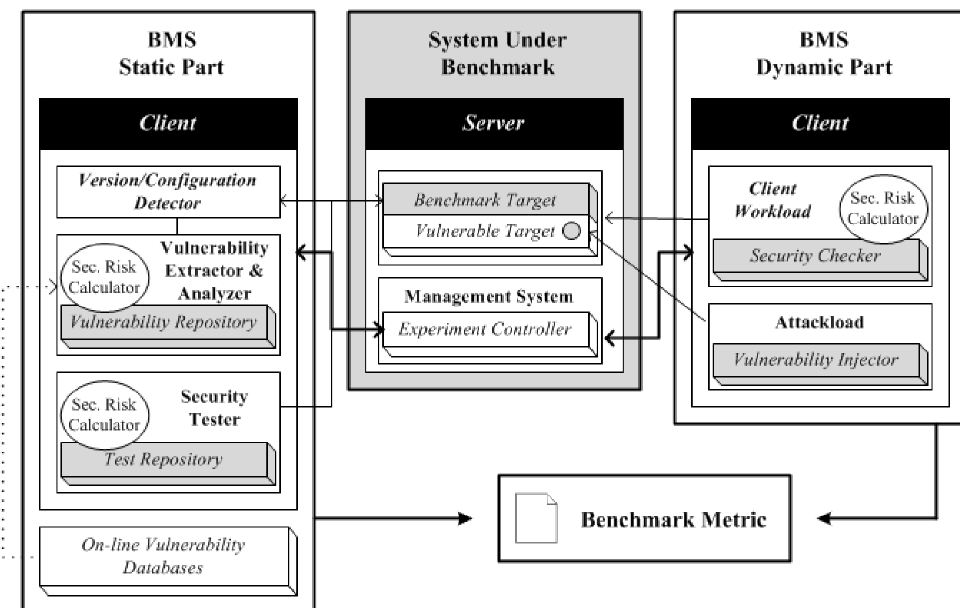

Figure 4-1. Security Benchmark and Instrumentation Components

- It shall run the two benchmark parts in different times.
- It shall collect and keep information that will be used later to estimate the benchmark metric.

Vulnerability Extractor and Analyzer: This instrumentation component conducts the load of known vulnerabilities into the repository and shall comply with the following rules:

- It shall extract vulnerability information from vulnerability databases, transform the data when needed, and load it into the repository.
- It shall categorize vulnerabilities either by using information that are available on vulnerability databases or by parsing information available in the vulnerability description.
- It shall keep the extracted information locally to avoid unnecessary queries to external databases.
- It shall be executed in a regular basis to keep the repository up to date (at least every week), ensuring that the most recent known vulnerabilities are taken into account during the benchmark execution.
- It shall consolidate benchmark results in an open format (e.g., CSV) to make risk estimation easier.

Version and Configuration Detector: This instrumentation component is responsible for gathering the brand name and versions of each software to be benchmarked, with the following rules:

- It shall detect the version and brand of the components of the system under benchmark.
- It shall keep this information in an open format (e.g., CSV).
- It shall provide the version and brand parameters to the experiment controller, which will be used to query the vulnerability repository and filter only the known vulnerabilities affecting the system under benchmark.

Test Collector. This instrumentation component is responsible for collecting security tests from representative security tools (e.g., widely used security scanners) and shall comply with the following rules.

- It shall collect tests from representative tools and load them into the Security Test Repository. The rules to select a representative tool were already described in Chapter 3.

- For each vulnerability associated to each test, it shall estimate the impact and probability of vulnerability exploitation following the criteria defined by the Common Vulnerability Scoring framework described in Chapter 2. The benchmark allows the manual estimation of these risk factors prior to the execution of the benchmark.

Test Executor. This is the tool responsible for executing the security tests against the target system and shall be implemented according to the following rules:

- It shall execute the tests available in the Security Test Repository, but only those applicable to the system under benchmark.
- It shall provide to the metric calculator the test results to the estimation of the vulnerability risk (in case a test is positive).

Metric Calculator: This component estimates the risk of the vulnerabilities identified during the benchmark run and shall be instantiated within the two benchmark parts and also should comply with the following rules:

- It shall have a Static Risk Calculator instance to estimate the risk of known vulnerabilities, consolidate them with the results of the security tests and compute the security risk of the static part.
- It shall have a Dynamic Risk Calculator that will measure risk to the benchmark target of successfully exploited vulnerabilities, also consolidating the results to form the security risk of the dynamic part.
- It shall have a Metric calculator to add up the security risk of the static and dynamic parts and compute the benchmark metric (*SBench*), following the criteria provided in Chapter 3.

4.3 BENCHMARK DEPLOYMENT PROCEDURE

The deployment of the benchmark consists in preparing the analytical and experimental setup that will enable the benchmark execution. This preparation addresses the installation and configuration of the hardware and systems aimed at supporting the benchmark execution, as well as the deployment of the benchmark supporting tools. We assume here that these tools are already developed, tested and ready for deployment. The deployment procedure is organized in five steps that shall be executed only once. These steps are described as follows:

- **Step 1.** Installation and configuration of the hardware needed to the execution of the system and benchmark. This starts with the installation of the computer equipment and network that will form the experiment infrastructure.

- **Step 2.** Installation and configuration of the software that runs on the top of the hardware infrastructure, the operating system.
- **Step 3.** Installation and configuration of the system to be benchmarked, including all component of the system (e.g., the benchmark target, and the vulnerable component).
- **Step 4.** Installation and configuration of the instrumentation components, ranging from the experiment controller to the metric calculator.
- **Step 3.** Installation and configuration of the benchmark components of the static part, including the Vulnerability Repository, the Version and Configuration Detector, the Vulnerability Extractor and Analyzer, Security Test Tool, and Static Risk calculator.
- **Step 4.** Installation and configuration of the benchmark components of the dynamic part, including the Workload, the Attackload, Security Checker, and the Dynamic Risk Calculator.
- **Step 5.** At the end of each phase, the installation and configuration must be verified. This basically consists in checking if all systems were properly installed, configured and will run as expected.

The execution of this preparation procedure must comply with one single rule: The hardware and software configuration shall remain the same across all benchmark runs. The only exception to this rule refers to the Benchmark Target, which is the component in which security metrics are collected for comparative purposes. For example, in our case study we benchmarked the security of functionally equivalent web servers. Different web servers are tested, but the remaining components and hardware configuration remain the same

4.4 STATIC PART PROCEDURE

The static part shall start with the detection of the version and current configuration of the components of the targeted system. Then, the benchmark searches in the *vulnerability repository* component for known vulnerabilities matching the version and configuration provided as input. Once this step is concluded, the benchmark runs the security test component to identify additional known vulnerabilities. Here the user is allowed to run vulnerability scanners tools to make sure that a large extent of known vulnerabilities are addressed. This simply consists in executing tests to verify whether given known vulnerability are present in the system under benchmark. At the end of this part, and considering the vulnerabilities identified (i.e., present), the benchmark calculates their risk and measures the security risk of the static part following the rules detailed in section

3.3.

The static part shall be run in four stages in the following order: Preparation, Vulnerability Extraction and Analysis, Security Test Execution, and Measurement Stage.

4.4.1 Preparation Stage

The purpose of this stage is to select the tools that will be used during the static part execution. This is applicable to the execution of security tests, as the extraction of known vulnerabilities may require the development of customized tools.

The rules to select a representative security tool that will run security tests to confirm the presence of known vulnerabilities were already discussed in Chapter 3. The purpose here is to remark that this shall be done prior to the execution of the remaining stages.

4.4.2 Vulnerability Extraction and Analysis Stage

This stage addresses the measurement of the security risk based on the knowledge of individual vulnerabilities collected from the field. This must be done in three steps:

- **Step 1.** Refers to the detection of the brand, version, and configuration of the SUB components.
- **Step 2.** Consists in selecting from the Vulnerabilities Repository known vulnerabilities matching the criteria provided in step 1, i.e., matching the specific SUB.
- **Step 3.** Includes the measurement of vulnerability risk as the last step of this stage. The criterion to compute this vulnerability risk is the one presented in Chapter 3.

4.4.3 Security Test Stage

This stage consists in the execution of the security testing tool to discover known vulnerabilities present in SUB and should include two steps:

- **Step 1.** Run the security test tool against each component of the SUB, or under the component under benchmark. The security tests must be executed one at a time. The results of each security test should be made available to benchmark users in an open format (e.g., (Shafranovich 2005)).

- **Step 2.** For each positive test (a test is positive if it reveals that a known vulnerability is present), the impact and the exploitability of the discovered vulnerability are computed to obtain the vulnerability risk. The criteria to estimate the impact and the exploitability risk are those defined in the Common Vulnerability Scoring System (Version 2), as seen in Chapter 3.

4.4.4 Measurement Stage

This stage consists of the measurement of the security risk of the static part. The procedure is as follows:

- **Step 1.** The security risk of each component must consider the relative importance of each component to the whole system. This relative importance shall be provided by the benchmark user.
- **Step 2.** At the end of the extraction and analysis of known vulnerabilities and of the execution of security tests, the security risk of the static part is computed. This security risk takes into account the risk of individual vulnerabilities detected in each component, as well as the component weight provided in step 1.
- **Step 3.** The security risk of the static part is computed by adding the results of Step 2. In fact, the security risk of the static part is given by the sum of the security risk of the Vulnerability and Analysis and Security Test Stage.

The execution of this stage shall comply with the following rules:

- This stage shall only be executed after the successful completion of all previous stages of the static part.
- The security risk of each component shall be computed based on the individual risk of vulnerabilities that were discovered in the Benchmark Target (BT) after the vulnerability extraction and test execution.
- The security risk of the static part shall be computed based on the security risk of each component.
- The security risk of each component shall consider the relative importance of each component to the whole system.
- The formula that should be used to calculate component and static part security risk were described in Chapter 3.

4.5 DYNAMIC PART PROCEDURE

The execution of the dynamic part is aimed at assessing the effects of unknown vulnerabilities to the benchmark target. The idea here is to execute attacks against system components while observing the security behavior of the benchmark target. This means that an experimental approach must be used to achieve this goal, and there is no certainty that the effects of all unknown vulnerabilities will be assessed.

The dynamic part begins by running the system under benchmark in a normal operation with the *workload* component. The purpose is to observe the system baseline behavior and collect measurements that will be used later to help evaluating whether the system security was compromised. Once the baseline execution is completed, attacks are executed against the system interface and against the system components through the *attackload* component (vulnerabilities are injected in system components by the *vulnerability injector,* prior to the benchmark execution). While attacks are being executed, the *security checker* keeps verifying if any of the security attributes of the benchmark target has been partially or completely compromised. This security checking is done by monitoring system responses. By the end of attack execution, and considering the risk of each successfully exploited vulnerability, the benchmark estimate the security risk of the dynamic part that, together with the security risk of the static part, will form the benchmark metric. Figure 4-2 shows the components of the benchmark dynamic part and illustrates how they work together to produce the security metric of the dynamic part.

Attacks are organized in execution slots. One attack execution slot corresponds to a measurement interval during which the benchmark workload is run and one or more attacks from the attackload are executed in order to evaluate the system behavior. This definition is adapted from the fault injection slot definition present in DBench-OLTP Clause 2.3.1 (Marco Vieira 2005).

Note that each run corresponds to the execution of attacks against only one vulnerability. This is necessary to determine which part of the system was actually compromised due to the exploitation of a particular vulnerability and to identify the value for the exploitability to use in the risk computation. This means that the number of benchmark runs corresponds to the number of vulnerabilities injected in the vulnerable component and also the number of vulnerabilities that are exploited in the system interface. If the attackload is set up to exploit 100 vulnerabilities, then the total number of benchmark runs will also be 100.

Another important aspect is the time needed to start and run the benchmark and

the system under benchmark. The maximum time to the execution of each benchmark run relies on the characteristics of the System Under benchmark. Considering the case study we conducted in Chapter 6 with web server systems, we recommend 40 minutes, with 30 minutes of measurement interval and 10 minutes to start/stop system components. This 30-minute value was defined considering two premises:

- The whole benchmark execution must not take a long time to complete as this would affect the benchmark acceptance. Ideally, results should be ready within 1 week, but 2 weeks would be acceptable if the system is complex and the volume of attacks is large. This execution time rule was defined based on the experience of previous dependability benchmark initiatives and on the experience we acquired with our case study (see chapter 6). In DBench project, the maximum time acceptable to benchmark a system is one week (DBench 2004). In (Kalakech et al. 2004), the benchmark execution time for measuring the dependability of each operating system was 46 hours – 6 days were necessary to benchmark 3 different operating systems. In (M. Vieira and Madeira 2003b), the authors took 12 days to benchmark the dependability of each OLTP system. In (Durães, Vieira, and Madeira 2004), the authors were able to compare the dependability of two web servers in 1,3 day. The time

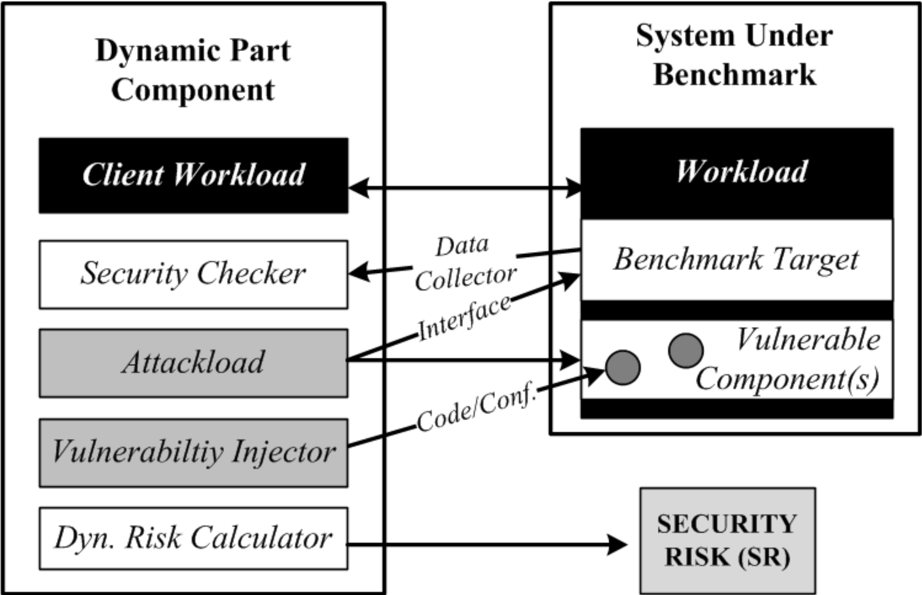

Figure 4-2. Benchmark Dynamic Part Components

needed to execute the security benchmark we built for our case study was 24 hours covering 6 different web servers.

- A small system with few applications and a small set of users and transactions will take few minutes to complete. In the case study presented in Chapter 6, we were able to execute thousands of attacks simulating multiple users and the measurement interval (dynamic part run) was 3 minutes. By recommending 30 minutes as the upper limit of workload execution, we believe that our security benchmark methodology is prepared to cover large systems, without putting the benchmark usability in jeopardy.

The dynamic part shall include the following stages: Preparation, Vulnerability Injection, Attackload Generation, Baseline & Attack Execution, Security Checking and Measurement Stage. These stages are described in the remainder of this section.

4.5.1 Preparation Stage

The preparation stage of the dynamic part is aimed at selecting representative components, vulnerabilities and attacks. This should be done during the benchmark deployment procedure.

To select a representative Vulnerable Component (the one that will be subject to vulnerability injection), the following rules shall be taken into account:

- The Vulnerable Component shall be different from the Benchmark Target. The reason for that is that it is not fair to inject vulnerabilities in the same component where security measurements are performed, as the results would be biased. For this reason, our specification remarks the importance of injecting the vulnerabilities in another elements of the SUB (Vulnerable Component) in order to emulate the impact of attacks launched by exploiting unknown vulnerabilities.
- The Vulnerability Component shall be in a reachable location from an attacker standpoint, allowing the successful execution of attacks. It makes no sense to select a component that will be out of attack range, making the attackload always unsuccessful.
- The Vulnerable Component shall interact with the Benchmark Target or at least with other system components that interact with the Benchmark Target. This is aimed at observing the impact of attacks over the benchmark target. If the Vulnerable Component does not send or receive any command or data that will reach the Benchmark Target, it is clear that

attacks will have no effect to the purpose of our benchmark.

The vulnerabilities and attacks types to be injected must be representative. This means existing field studies on vulnerabilities must be used to support this part of the injection process. The important aspect here is to choose vulnerabilities that are representative in the field. These rules were detailed in Chapter 3 and the purpose here is to clarify that this shall be executed prior to the vulnerability injection stage.

4.5.2 Vulnerability Injection Stage

This stage injects vulnerabilities into the Vulnerable Component using the Vulnerability Injector component. This stage must be executed in the following phases:

- **Step 1.** In this step, the identification of the vulnerability injection points is performed. These points are located in the source code and configuration files of the Vulnerable Component. For example, a web site crawler could be used to collect information about a web application (Vulnerable Component) and map the input and output variables in an automated way (J. Fonseca, Vieira, and Madeira 2009). Also, communications with the database could also be intercepted by installing a probe mechanism between the web application and the back-end database (J. Fonseca, Vieira, and Madeira 2009).
- **Step 2.** This consists in changing the source code or configuration to a vulnerable state. This should be done for each injection point located in the previous step.
- **Step 3.** If changes are made in the source code, it is necessary to compile the changed code and generate the binary to be deployed in the system.
- **Step 4.** The vulnerability component is deployed along with the system, to be executed during the benchmark run.

The execution of this stage should comply with the following rules:

- The Vulnerable Component where vulnerabilities will be injected must interact with the benchmark workload. Otherwise, attacks will not be successful since they will not reach the vulnerable component and the effects of these attacks to the BT will be impossible to determine.
- Vulnerabilities are injected prior to the benchmark run.
- Vulnerabilities shall be injected in places of the code and/or configuration

of the Vulnerability Component that will allow the attackload to conduct a successful vulnerability exploitation. Although this certainly limits the number of vulnerabilities injected, this does not reduce the effectiveness of the approach. The main purpose of injecting vulnerabilities will be achieved.

- Vulnerability injection procedure must be repeated until all vulnerabilities points were covered. In case of software fault vulnerabilities, the result will be a set of files containing one or more vulnerabilities.

4.5.3 Attackload Generation Stage

The goal of this stage is to generate the malicious codes that will exploit the vulnerabilities that were injected in the previous stage. The phases to execute this stage are as follows:

- **Step 1.** Based on the analysis of the location, code pattern, and data type of the vulnerabilities that were injected, it is necessary to generate possible (malicious) values and exploits that could be used to exploit each vulnerability.
- **Step 2.** Creation of the actual exploits with malicious values attached into it. For example, in the case of a web application, this corresponds to the construction of an HTTP request with the malicious content in a GET or POST function.

The execution of this stage must comply with the following rules:

- For each vulnerability injected in the previous stage, a set of malicious interactions (attacks) and their expected outcome (result of the attack) shall be generated. This will be used later one to identify if the attack was successful.
- Each attack shall not have only one instance. For example, a Cross-site scripting vulnerability should be exploited using different values and attack patterns. The guidelines to mount these attacks were provided in Chapter 3.

4.5.4 Baseline and Attack Execution Stage

This stage collects security measurements when no attacks are executed during the benchmark run (Phase 1) and when attacks are executed (Phase 2). As described in Chapter 3, the maximum time recommended to the execution of each benchmark run is 40 minutes.

The steps to execute this stage are as follows and are illustrated in Figure 4-3:

- **Step 1.** The workload is submitted without the presence of attacks. This corresponds to the workload measurement interval and are used to collect baseline security measures through security checks that represent the security of the system with normal optimization settings.
- **Step 2.** Execution of the workload in the presence of attacks, which are organized in execution slots. Each slot corresponds to a measurement interval during which the workload is run and one or more attacks are executed to evaluate the system behavior. As mentioned in Chapter 3, this notion is adapted from the fault injection slot definition present in DBench-OLTP Clause 2.3.1 (Marco Vieira 2005).

  The evaluation of the system behavior involves security checks, which are conducted in two steps:

  - Collection of the measurements of the System monitor and Data Collector after the execution of each attack.
  - Based on the impact and exploitability rules defined in Chapter 3, it determines the level of security impact and exploitability of a successfully exploited vulnerability.

The execution of the first phase of this stage must comply with the following rules:

- Benchmark tools and the workload shall be run without the execution of attacks (baseline execution stage).
- Baseline execution shall be organized in three periods:

  Ramp-up time. This is the period that workload applications are starting and performing the first transactions. During this time, no SUB responses and security measurements should be collected. The duration of the ramp-up time should not last more than the start-up time. We acknowledge that this duration can vary from system to system, but a small system with few applications and transactions will not take more than a few seconds of ramp-up time (e.g., 30 seconds), while a large system could take minutes of ramp-up time (e.g., 5 minutes). The same notion is applied to the ramp-down time described later on. The maximum recommended time for ramp-up is 5 minutes.

  Measurement time. This period immediately follows the ramp-up. At this time, all applications of the workload and BMS tools were fully started

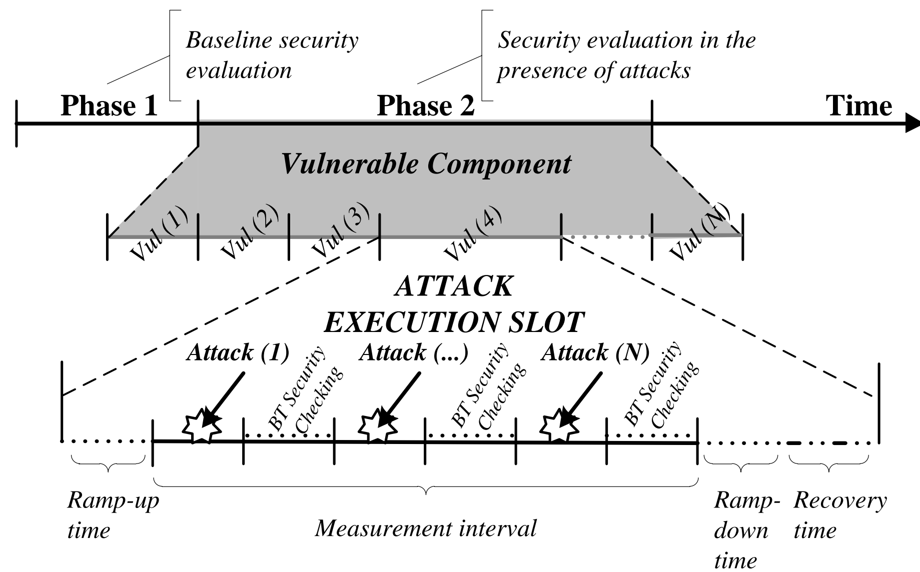

Figure 4-3. Inspection and Attack Execution Stage

and are responding to workload client requests, and measurements can commence. Security measurements from the BT should be collected during this period. The duration of the measurement time should not last more than the time needed to completely exercise the different parts of the systems, with requests of different formats and sizes and with different users. This should last a few minutes for a small system (e.g., 5 minutes) and several minutes for a large one. The maximum recommended time for the measurement interval is 30 minutes.

Ramp-down time. This period corresponds to the closing of the operations of the system related to the workload (i.e., the workload has ended). Depending on the system, it may require explicit commands from the BMS. No security measurements should be collected during this period. The maximum recommended time for ramp-down time is 5 minutes.

- Baseline measurements (e.g., response time) and data to be used by the security checker shall be collected during this stage.

The execution of the second phase of this stage should comply with the following rules:

- Attacks should target the vulnerabilities injected into the Vulnerable Component (source code, configuration, interface, etc.).

- The attack execution should be organized in four periods, which are very similar to the baseline execution. Here we focus on the items that are particular to the attack execution, which are:

  Ramp-up time. During this time, no attacks should be executed. The ramp-up duration is the same one set for the baseline phase, with an upper limit of 5 minutes.

  Measurement time. Attacks should last until the measurement time of the workload reach the timeout period, with an upper limit of 30 minutes.

  Ramp-down time. No attacks should be executed during this period. The ramp-up duration is the same one set for the baseline phase, with an upper limit of 5 minutes.

  Recovery time. This is the time needed to recover the vulnerable component to the state prior the attack execution. This is aimed at avoid that the impact of an attack can affect the result of the next attack, with an execution limit of 5 minutes.

- Each attack execution slot should exploit exactly one vulnerability. This is required to be able to assign the impact to each vulnerability.
- Attacks should be executed within the maximum measurement time.

The execution of security checks should comply with the following rules:

- The Security Checker component must monitor and assess any impact in the security attributes of the Benchmark Target during and after the execution of each attack.
- The Security Checker must also consider the exploitability level of each attack following the rules specified by CVSS.
- At the end of attack execution, the security checker must determine how many attacks were successful.
- As the vulnerability exploited will be always the same during an attack injection slot, just one impact and exploitability metric must be produced.
- The attackload component must be configured and prepared to exploit each vulnerability at a time during the workload execution.
- The Security Checker component must be prepared to analyze the expected result of the attack of the vulnerable BT and verify if the typical response of non-vulnerable BT was changed (and in which degree was changed) during the attack execution.

- At the end of each successful attack execution, the individual risk of discovered vulnerabilities must be measured. The criteria defined in the Common Vulnerability Scoring System to compute individual vulnerability risk must be used.

4.6 BENCHMARK METRIC COMPOSITION PROCEDURE

The final stage of the benchmark execution is the estimation of the benchmark metric based on the results provided by the static and dynamic parts. The execution of this stage should comply with a single rule: The formula that shall be used to compute the vulnerability, component, and system security risk is the one described in the section 3.5. This shall be done in four steps, which are:

- **Step 1.** This consists in collecting the vulnerability estimation reports produced during the execution of the static and dynamic parts. These reports already have the list of identified vulnerabilities (static part) and successfully exploited vulnerabilities (dynamic part).
- **Step 2.** It is necessary to estimate the vulnerability risk considering the impact of vulnerability exploitation and its probability. Once this is concluded, these risk are added up by category and compose the component security risk.
- **Step 3.** In this step, it is necessary to estimate the security risk of each benchmark part by multiplying the component security risk and the component weight assigned by the benchmark user[4] (which represents the importance of the component to the whole system), also considering the categories of each risk.
- **Step 4.** This refers to the estimation of the benchmark metric (SBench), by adding up the security risk of each benchmark part and taking into account the weights of each risk category.

4.7 BENCHMARK DISCLOSURE REPORT

A Disclosure Report is required for the results to be considered compliant with our security benchmark methodology. The objective of this report is to allow benchmark implementers to reproduce the experiments in functionally equivalent systems. Here are the requirements to build such report:

- The benchmark metrics should be included in the Disclosure Report. In Chapter 6, we provide the metrics (*SBench*) and benchmark results that

[4] Benchmark user's decisions shall be equally applied across all the systems under benchmark.

should be taken into account to meet this requirement.

- All information collected for the estimation of the security benchmark metric must be disclosed in the Disclosure Report. This shall include at least the list of identified or attacked vulnerabilities during the benchmark run with the respective vulnerability risk score.
- All the information considering the benchmark components specification have to be disclosed. This basically consists in the specification we provided in Chapter 3. More specifically, the details we provided in regarding the rules supporting each component.
- All implementation details must be disclosed, including the technical characteristics of the components that form the BMS. This refers to a summarized version of the implementation details we provide in the next chapter.

4.8 CONCLUSION

This chapter presented the procedures and rules to deploy and execute our security benchmark methodology, following the traditional prescriptive (rule or clause oriented) approach that is used in most of the benchmarks, especially performance and dependability benchmarks. We first described the benchmark general procedures, specifying that the benchmark user is free to choose which benchmark part will be executed first and that the benchmark duration depends on the assessment of system's vulnerabilities. We then described the rules to build the instrumentation components and the procedure that shall be followed to install and configure the benchmark components and the system under benchmark.

The procedures of the benchmark static part were presented and organized in four execution stages: preparation, vulnerability extraction and analysis, security test execution, measurement stage. The purpose of the first stage (preparation) is to build the analytical setup of the benchmark. The second stage seeks to extract and analysis known vulnerabilities from the benchmark vulnerability repository. The third stage is aimed at executing security tests against the system under benchmark to confirm the existence of a given vulnerability, executing only one test at a time and consolidating the results in an open format (CSV). Finally, the measurement stage consists in computing the security risk of the static part.

The benchmark dynamic part procedures were also described. These procedures were organized in five execution stages: preparation, vulnerability injection, attackload generation, baseline & attack execution, security checking, and measurement stage. The preparation stage is aimed at building the experimental

setup of the benchmark. The attackload generation is aimed at building an attack for each vulnerability injected in the vulnerability injection stage. The baseline and attack stage starts by executing the benchmark with no attack execution (to observe the behavior of the system in a normal scenario, without attacks) and then it executes attacks against the vulnerable target (exploiting the vulnerabilities that were injected). The security checking verifies the system response after each attack execution and assesses the attack impact to the whole system, collecting security measurements from the benchmark target. The security risk of the successful attacks of the dynamic part is estimated in the measurement stage.

It is worth noting that the purpose of the procedures and rules presented in this chapter is to ensure that our methodology will be implemented in a similar way by different teams. We do believe that the guidelines we provided will allow benchmark users to implement and execute a security benchmark in a standardized fashion, producing repeatable and comparable results. This is especially important to make the benchmark useful and to facilitate its acceptance by the security community.

CHAPTER 5

5. BENCHMARK IMPLEMENTATION

This chapter presents the implementation of the security benchmark methodology for a widely used system class: the web serving systems. These systems form the basis of many important private and public-related services, such as e-commerce and banking systems, and are, in fact, essential to the current society way of life. Web serving systems are typically built by several heterogeneous components, resulting in complex systems. This complexity potentiates the existence of internal vulnerabilities that might be exploited by attackers. Because these systems are naturally connected to the Internet, and thus exposed to many users and attackers, any internal vulnerability becomes a real threat to security (e.g., (OWASP 2013; B. Martin et al. 2010)). Therefore, the use of a web-serving scenario as case study of our security benchmark is both useful and pertinent.

Our security benchmark methodology implementation serves two purposes. The first purpose is to support the case study of benchmarking the security of real web serving systems (Chapter 6) and show that it is a feasible methodology to evaluate and compare security of software systems. The second purpose is to provide to benchmark implementers an example (and a guideline) on how to overcome the technical difficulties to implement a security benchmark. It is not our goal to provide a final and standard benchmark implementation, especially because a benchmark implementer could find a different technical solution that would better fit on his or her environment. The idea here is mostly to demonstrate the practicality of our benchmark definition and to provide the key tools that could sensibly speed up the implementation of a security benchmark targeting web serving systems, with a particular focus on the web server component.

The reason to target the web server over the remaining components is due to its

central role handling the user requests and connecting the outside environment to the internal resources of the web serving system, making it a key component regarding security aspects. We provide a set of tools to benchmark the security of web servers following the static and dynamic approach of the security benchmark methodology described in Chapter 3, which include tools to discover known vulnerabilities on web servers, and tools to attack web servers and their contents (in our case, web applications) to assess the effects of unknown vulnerabilities.

The following tools were used in the static part: the Vulnerability Extractor Analyzer (VEXA) and the Nikto Security Testing Tool (Nikto2 2015). VEXA collects and analyzes known vulnerabilities from vulnerabilities reported in the on-line public databases such as Open Source Vulnerability Database and the National Vulnerability Database. Nikto is a widely used security tool with a large set of tests aimed at checking the presence of vulnerabilities in a wide range of web server's brands.

It is worth noting that the tools developed to enable and support our security benchmark are built for Microsoft Windows Platform. This means that benchmark users interested in building a security benchmark for other platforms (e.g., Linux, Solaris, AIX) shall adapt our tools accordingly - or develop new ones - based on the procedures and rules defined in Chapter 3.

The remainder of this section is described as follows. Section 5.1 provides the definition of the target system. Section 5.2 presents the benchmark rules specific to web serving systems. Section 5.3 describes the implementation of the static part of our security benchmark methodology. Section 5.4 presents the implementation of the dynamic part. Section 5.5 shows how to consolidate static and dynamic part results to obtain the benchmark metric. Section 5.6 concludes this chapter.

5.1 BENCHMARK TARGET DEFINITION

Web Serving Systems are the system under benchmark of our case study, providing the operational environment for the execution of our benchmark target. These systems are composed by a set of components that when put together provide web-based services ranged from simple static information repositories to web-applications such as e-banking. The typical main components of a web serving system are the operating system, the web server and its hosted web applications, including a database engine. The use of this particular type of system in our study is relevant as web-based services are currently critical to many society aspects, and because its components can be obtained from several alternatives sources, using security as selection criteria is pertinent.

Web servers, an important component of any web serving system, are our benchmark target. The remainder of this section describes the main characteristics of web servers and provides the definition of web applications, a key component that run on the top of web servers. In fact, the web application is the component chosen to inject vulnerabilities that will be later exploited by attacks during our experimental setup.

5.1.1 Web Servers

The web server is a component that serves data (e.g., web pages, files, etc.) to web clients (e.g., (Pettit 2001; Vass et al. 1998)) using a communication protocol such as HTTP protocol (Fielding et al. 1999). Web server is responsible for accepting HTTP requests and sending the requested data to the user's browser through a markup language (HTML). Examples of widely used web servers are the Apache Web Server (Apache HTTPD 2015) and the Internet Information Services web server (Microsoft IIS 2015).

5.1.2 Web Application

A web application is a computer program that executes in a client-server environment. The typical web application architecture contains three components: web browser, web server, and application server (Hassan and Holt 2002).

Figure 5-1 depicts an example of a web application architecture including the web browser component that runs in the user machine and the web server and the application server components that reside in the server machine. The application server component is composed of sub-components (e.g., database, web services, and multimedia objects).

Web application units (components and sub-components) are typically components off-the-shelf (COTS), which typically means that the components were not developed by the same software company. This means that different components may have been written in different languages and according to

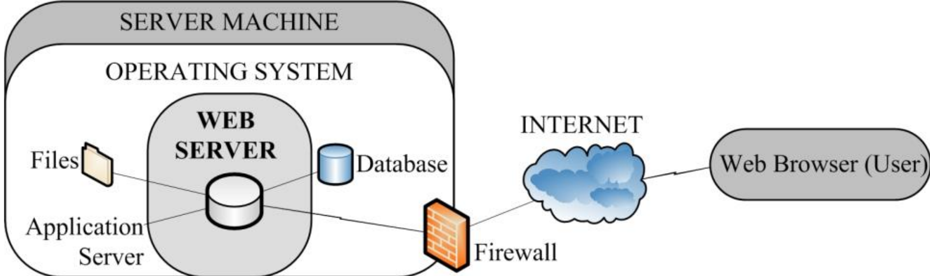

Figure 5-1. Web Application Architecture Example

different development methodologies, resulting in a higher risk of mismatches at the interface level. The web application complexity, the technological mismatches across its components, and the back-box nature of COTS components contribute to the high risk of vulnerabilities in these applications. For that reason, requirements, policies and mechanisms of a web application are fundamental to guarantee the availability, integrity and confidentiality for the whole web application architecture, including our target component (web servers).

5.2 INSTANTIATION OF THE BENCHMARK RULES FOR WEB SERVING SYSTEMS

In Chapter 3 and 4, we presented the rules and procedures of our security benchmark methodology that are applicable to any software-based system. In this section, we discuss the concrete application of the rules to the case study and provide examples of technical decisions related to the implementation of the benchmark.

5.2.1 Vulnerability Repository

The vulnerability repository shall be formed by vulnerability information targeting web serving systems components. This means that this repository should contain known vulnerabilities from web application, web servers, operating systems, and databases, covering the most extent possible of vulnerabilities from different system brands and versions. Obviously, it is important to extract information from representative vulnerability databases available in the field.

We use the Open Source Vulnerability Database, as it already collects information from other databases and has an on-line platform that allow users to report known vulnerabilities. We also use the National Vulnerability Database to complement vulnerability information (e.g., vulnerability categorization, CVSS metrics), as this database uses the Common Vulnerability Scoring System to score the risk of each vulnerability.

There is no restriction regarding the number of vulnerabilities to be stored in the repository. The more information we have about known vulnerabilities, the more precise the benchmark metric estimation will be for the static part.

5.2.2 Security Test Repository

The Security Test Repository shall be built with information from realistic tests targeting web serving systems components. As in our case study we focus on the web server component, this means that the security tests shall be addressing the weaknesses of different brands and versions of web servers.

As the methodology allows the use of existing security testing tools, we use the Nikto Security Testing tool (Nikto2 2015). Nikto totally complies with the requirements present in our benchmark specification, for example:

- It has a database of security tests with the entities and attributes needed by the Security Test Repository.
- It contains non-intrusive security tests. These tests essentially check the presence or absence of a file, or check of the web server response has unexpected values.
- It allows the development of new, additional features. The fact that Nikto is open source allowed us to incorporate our own benchmark features to estimate the vulnerability risk of each of the tests executed by Nikto.

There is also no restriction regarding the number of security tests to be hosted by the security test repository and execute against the target system. However, during the benchmark execution, it is important to make sure that the number of tests will not impact the total time allowed for the benchmark execution. If that is the case, the security tests targeting high-risk vulnerabilities shall be prioritized.

5.2.3 Workload

The workload shall simulate a web serving system environment. This means that benchmark implementers shall built an experimental setup with an operating system running a web server, a web application, and a database. The workload runs in a client-server environment (as the one illustrated in Figure 5-1), with emulated users sending and receiving requests to the web server and web application from a remote network.

We use existing workloads (e.g., such as the ones specified in TPC) from the performance field to simulate a web serving system environment. More specifically, we use the TPC-W workload, which simulates the activities of an ecommerce web site, with emulated users browsing and ordering products.

There is no restriction regarding the number of requests per emulated users that can be done during the execution of the workload, which should be implemented in accordance with TPC-W specification (TPC 1988). However, there is a very important distinction to be made. In the TPC-W benchmark, the client workload requests stop when the benchmark execution timeout is reached. In our methodology, the client workload will stop sending requests when the total number of attacks are reached. There is no timeout associated to the execution of the client workload as we need to have the same amount of attacks executed across all benchmark runs. This is better detailed in section 5.2.5.

5.2.4 *Vulnerabilityload*

The vulnerabilityload shall be formed by representative vulnerabilities targeting the web serving system component chosen to be subject of vulnerability injection. To this end, a field study shall be conducted to find the common vulnerabilities targeting the Vulnerable Component, which is necessarily different from the benchmark target. In our case, as we use the web application as the vulnerable component, we adopted the results of the field study published by (J. Fonseca and Vieira 2008), which identified Cross-Site Scripting and SQL Injection vulnerabilities as the most representatives for this system class.

There is no restriction regarding the number of vulnerabilities to be injected, except if the exploitation of these vulnerabilities will last longer than the maximum duration allowed. If the vulnerable component has a large set of injection points, then the number of vulnerabilities should be limited to the ones that poses a major risk to the vulnerable component, which can be verified through the vulnerability risk score of the vulnerability to be injected.

5.2.5 *Attackload*

The attackload structure is conditioned to the type of vulnerabilities that are injected by the vulnerabilityload. However, only attacks targeting web serving systems components shall be implemented. For vulnerabilities that weaken the input parameters of a web application (improper input parameter sanitization), Cross-site and SQL injection attacks shall be mounted, for example.

The number of attacks to exploit each vulnerability shall not impact the maximum duration allowed per benchmark run (30 minutes according to our methodology). Although there is no specific rule to limit the amount of attacks against each vulnerability, the same number of attacks shall be executed in each benchmark run. This is to ensure a fair comparison among functionally equivalent systems, as a more attacked system could have a lower security measure than the ones with a reduced number of vulnerability exploitation. We adopted the following attack parameters to have the dynamic part completed in 24 hours (considering the 5 vulnerabilities types covered in the case study described in Chapter 6):

- Each vulnerability is exploited by 1 attacker executing 20 attacks per benchmark run. This is intended to observe the system security behavior without stressing its capacity.
- Each vulnerability is exploited by 20 attackers, each one of them executing concurrently 10 attacks per benchmark run (a total of 200 attacks). The goal here is to stress the capacity of the system to deal with

concurrent attacks.

The values and parameters implemented are also chosen in accordance with the vulnerability to be exploited. For Cross-site script attacks, it is necessary to send malicious characters using Java script code. The characters to be implemented shall be taken from the attack pattern recommended by the Common Attack Pattern Enumeration and Classification (CAPEC) and shall vary for each attack implemented, with the purpose of having different attacks of the same class inside the attackload.

5.3 STATIC PART IMPLEMENTATION

This section presents our implementation for the components of the static part that seek to identify known vulnerabilities on web servers and estimate their security risk. The components of the static part covered by our implementation are depicted in Figure 5-2 and described in the remainder of this section.

5.3.1 Experiment Controller

The Experiment Controller orchestrates the execution of all components of the static part. This component is implemented through two batch scripts. The first script is written in Microsoft DOS batch and starts the VEXA tool, which collects and analyzes known vulnerabilities for the components listed in its configuration file. The second script is a Microsoft DOS batch file that runs the Nikto Security Testing tools. It starts by stopping all web servers' services. Then it runs one web server at a time and then executes a Perl program (Perl 2013) to trigger Nikto tool.

5.3.2 Vulnerability Repository Implementation

The Vulnerability Repository is a database to keep vulnerability information that are used to identify known vulnerabilities in the system benchmark. The first requirement presented in Chapter 3 to build the Vulnerability Repository is to use an external DBMS. In our implementation, the repository is formed by a local instance of two popular vulnerability DBMS: the Open Source Vulnerability Database (*osvdb_repository*) and the National Vulnerability Database (*nvd_repository*). These instances are installed in the Benchmark Management System server. These databases, once installed, were deployed using MySQL Database commands according to the instructions available at (MySQL 2012). MySQL database was selected because OSVDB is also provided in the form of a MySQL dump file.

As specified in our benchmark methodology, and for each vulnerability, the minimum required information is the following: id (e.g., CVE ID), release date,

description, affected system, affected components, affected versions, affected platforms, CVSS impact, CVSS exploitability. The Open Source Vulnerability database is formed by a set of entities that contains more information than those specified in the requirements. The most central entities are vulnerabilities, cvss_metrics, and objects. The vulnerabilities entity stores information about each vulnerability reported in the database, hosting the dates, description, solution, and etc. The cvss_metrics entity hosts information about the impact and exploitability of each vulnerability. The objects entity relates each vulnerability to its name, version, and vendors.

The local instance of the National Vulnerability Database (*nvd_repository)* is formed by only one table, the nvd_local_repository. This table stores data that complements the vulnerability information stored in the *osvdb_repository* database (e.g., vulnerability categorization, CVSS metrics). The Vulnerability Extractor collects information from NVD vulnerability reports and then loads them into nvd_local_repository table. An example of vulnerability report from NVD can be found at (CVE-2013-2205 2013). This is a web page that contains the information that we collect and store in the *nvd_repository* database.

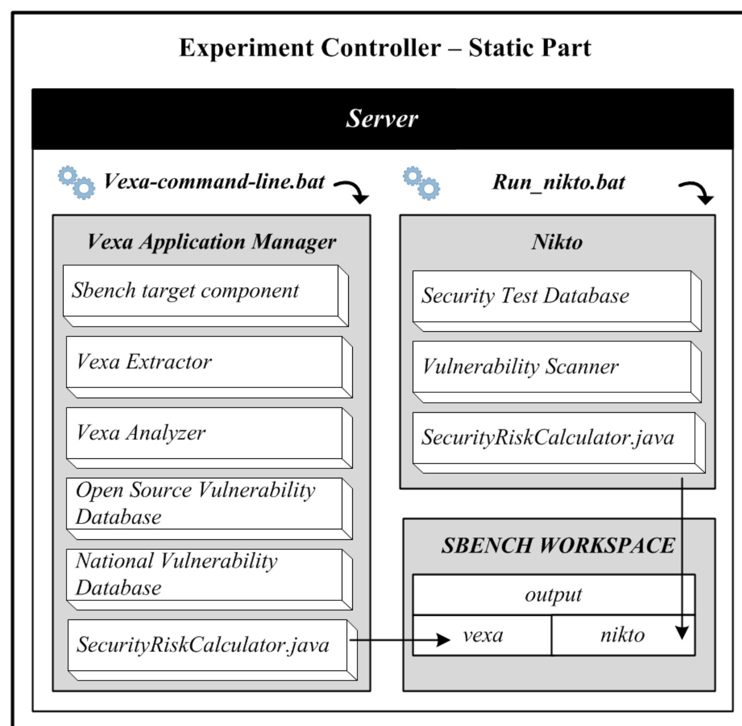

Figure 5-2. Experiment Controller of the Static Part

5.3.3 *Vulnerability Extractor and Analyzer (VEXA)*

The Vulnerability Extractor and Analyzer collects vulnerability information from public vulnerability databases and loads them into the Vulnerability Repository. The components of the vulnerability extractor and analyzer are: the Vulnerability Extractor, the Vulnerability Parser, the Vulnerability Analyzer, the Vulnerability Report Generator, and the Database Synchronizer. These components are implemented in a single tool (the VEXA tool) that extracts vulnerability information and provides a set of reports in the form of spreadsheet files that we use to estimate the security risk of each web serving system component.

VEXA is available in a web-based and in a command-line format. The web-based format collects and extracts vulnerability information and provides the results in a CSV format that can be downloaded by VEXA users (as illustrated in Figure 5-3). The command-line format is aimed at skipping the downloading phase and provides summarized results in a CSV output. This output contains the static part vulnerability risk estimated based on extracted vulnerability information from the component under analysis.

The technology supporting VEXA tool is as follows: the programming environment is Java Enterprise Edition; the architectural pattern is the model-view-controller (MVC) implemented using Spring; the database used is MySQL 5 Community Server; and the web server is the Apache Tomcat 6. The advantage of using MVC design pattern is to separate the information representation from the view layer.

5.3.3.1 VEXA Database Synchronizer

Another relevant aspect is to keep the local instances of the vulnerability databases up to date, with the most recent vulnerabilities. We implemented a database synchronizer that downloads a dump file from OSVDB database and imports the data in our local database instances.

The database synchronizer is executed in a regular basis in the Benchmark Management System server. This execution is configured as an event of the Microsoft Windows Batch Scheduler, which runs a batch file that calls the DumpLoader Java class. At the end of the execution, the DumpLoader sends an e-mail to the benchmark user informing the synchronization status.

5.3.3.2 VEXA Vulnerability Parser

The idea of the Vulnerability Parser is to put in the Vulnerability Repository additional vulnerability information using as source the National Vulnerability database. This is done by collecting information from vulnerability reports

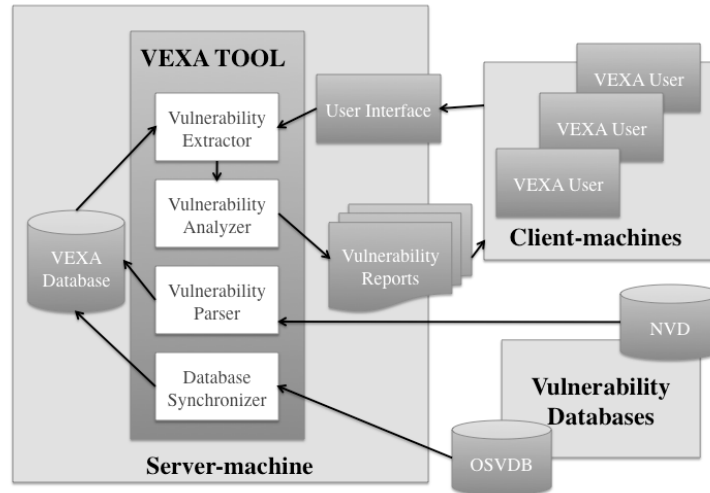

Figure 5-3. VEXA Information flow

available at NVD web site. More specifically, it connects to a NVD vulnerability report web page, using the CVE-ID as a URL filter (on NVD, each vulnerability report is identified uniquely by a CVE-ID). After the connection is established, the parser downloads the entire page to a local file. Then, it searches specific information (e.g., vulnerability type, vulnerability impact, CVSS score) in the downloaded web page using Regular Expression. At the end, the local repository is updated with the collected information.

5.3.3.3 VEXA Vulnerability Extractor

The Vulnerability Extractor executes queries against the Vulnerability Repository (OSVDB and NVD local database instances) and brings the list of known vulnerabilities of a given component. After obtaining the system brand and version, VEXA creates a temporary repository table and loads it with the known vulnerabilities collected from the OSVDB database. Then, it completes vulnerability information with data collected by the Vulnerability Parser. This is done by querying NVD local instance and storing the result into a temporary table. To avoid any duplicity of temporary tables, each one of them is created with a different identification. At the end of the vulnerability analysis and report generation, these temporary tables are removed.

5.3.3.4 VEXA Vulnerability Analyzer

The Vulnerability Analyzer was implemented to remove unnecessary information from the repository, classify vulnerabilities that were not classified in public

databases (using an internal dictionary based on the vulnerability description), and estimate the vulnerability risk using the CVSS impact and exploitability scores collected from the public databases. It avoids unnecessary queries to the external NVD web page, by checking if the vulnerability information that will be enriched was already collected in previous executions. More specifically, each information collected from NVD web page is kept into the *nvd_local_repository* and the web page is only parsed in the first time a given vulnerability is queried.

The categorization of vulnerabilities that were not categorized by OSVDB or NVD is done by analyzing the keywords typically found in a vulnerability description and assigns a vulnerability category following NVD classification criteria (NVD-CWE 2013).

5.3.3.5 VEXA Report Generator

The Report Generator was implemented to consolidate the information of known vulnerabilities in a set of files that will make easier the estimation of the security risk of the static part. Reports are available both from the web and from the command-line interface. These reports are described as follows.

- Raw Data. This report lists all known vulnerabilities collected for a given component and is used as data source by the remaining Excel files.
- CVSS Risk Results. This presents the graphs and summarizes data about the risk of known vulnerabilities.
- Exploitability Results. This presents the graphs and summarizes data about the exploitability of known vulnerabilities (the probability factor of our vulnerability risk equation).
- Impact Results. This presents the graphs and summarizes data about the impact of known vulnerabilities.
- Vulnerability Frequency Results. This presents the graphs and summarizes data about the frequency of known vulnerabilities.

5.3.4 Version and Configuration Detector Implementation

The purpose of the Version and Configuration detector is to identify the brand and version of the system under benchmark when using the command-line version of the tool. The Version and Configuration detector is formed by two components: the Brand and Version Checker and the Configuration Checker. However, no integrated tool to check component brand and version was implemented. We simply run existing component commands or access control panel tools to get the

information we need.

5.3.5 Security Tester Implementation

One of the key components of the static part is the Security Test Repository, a database that contains tests that can be used by a tool to confirm the existence of known vulnerabilities in the system under benchmark. To implement this component, we decided to adapt a widely used security testing tools that checks the existence of known vulnerabilities on web servers. Nikto is developed in PERL and is simple to execute. The user needs to specify the IP address and port of the web server target using Perl commands.

Figure 5-4 shows the components of the modified version of Nikto, which are described as follows:

- **Test Database.** This represents the implementation of the Security Test Repository of our benchmark methodology and contains the tests to be run against the web server. In the security test entity of this database, we included an extra field containing the CVSS2 scores (impact and exploitability factors) that is used to estimate the security risk of a positive test.
- **Nikto Controller.** It is responsible for executing the tool, loading configuration files and calling additional plugins.
- **Nikto Plugins (Test Executor).** It is composed by a set of plugins that executes many security tests against the web server under benchmark.
- **SBENCH Plugin.** This is the component we developed to assess the vulnerability risk of each positive security test.
- **Benchmark Reports.** It is the component responsible for keeping the tool execution log and providing results in an open format (CSV, HTML, XML). We also added in this report the security risk after a test execution (when successful).

The remainder of this section provides more details about the components that we have-implemented to enable the execution of security tests.

5.3.5.1 Security Test Database

The Security Test Database is the implementation of the Security Test Repository of our benchmark methodology, currently hosting 6495 security tests for different brands and versions of web servers.

The name of this database in Nikto is db_tests. Figure 5-5 shows the last lines of

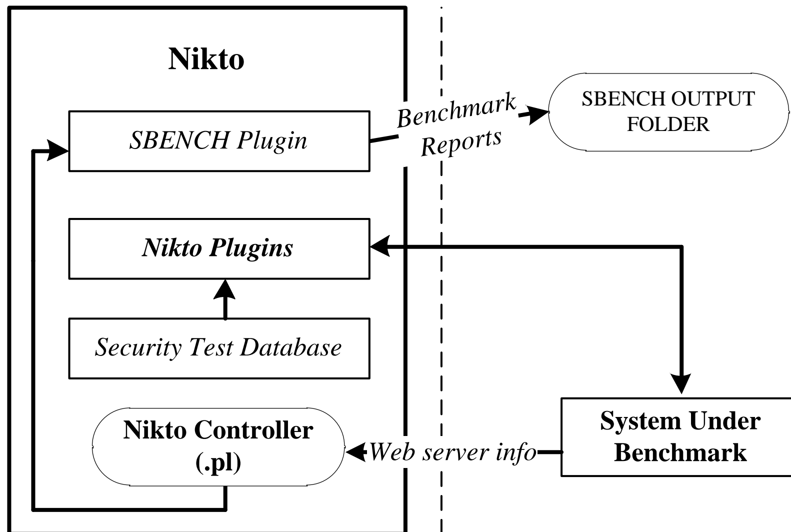

Figure 5-4. Nikto web server scanner components

the db_tests database. The fields of this db_test file are described as follows:

- **TEST ID.** This is the numerical identification of each test run by Nikto.
- **OSVDB ID.** This is the vulnerability entry reported on the OSVDB on-line database. If no OSVDB ID is reported, this refers to a security test that targets a vulnerability that is probably not reported in public vulnerability databases.
- **Server type.** This refers to the security test type. This field can have ~~has~~ one of the following values: 0 (File Upload), 1 (Interesting File / Seen in logs), 2 (Misconfiguration / Default File), 3 (Information Disclosure), 4 (Injection (XSS/Script/HTML)), 5 (Remote File Retrieval (Inside Web Root), 6 (Denial of Service), 7 (Remote File Retrieval (Server Wide)), 8 (Command Execution / Remote Shell), 9 (SQL Injection), a (Authentication Bypass), b (Software Identification), c (Remote source inclusion).
- **URI.** This field presents the URL that will point to the target of the tests.
- **HTTP Method**. This is the HTTP method that is used during the security test.
- **Match 1.** String or code used to match for a positive test.
- **Summary.** Summary message to report if a vulnerability was discovered.

TESTID	OSVDB-ID	SERVER TYPE	URI	HTTP METHOD	Match 1	Summary	HTTP Data	Headers	CVSS Base Vector
6486	59001	7	**/axis2/services/Ve**	GET	root:	Apache Axis2 contains a direc			**AV:N/AC:L/Au:N/C:C/I:N/A:N**
6487	0	b	**/Util/login.aspx**	GET	EPiServer	EPiServer admin login page fc			**AV:N/AC:L/Au:N/C:P/I:N/A:N**
6488	3092	b	**@PHPMYADMINse**	GET	index\.ph	phpMyAdmin is for managing			**AV:N/AC:L/Au:N/C:P/I:N/A:N**
6489	0	1	**/jsp/**	GET	[iI]ndex [c	JSP directory has indexing en			**AV:N/AC:L/Au:N/C:P/I:N/A:N**
6490	0	b	**/portal/console/**	GET	function\s	Vignette Server admin consol			**AV:N/AC:L/Au:N/C:P/I:N/A:N**
6491	0	2ab	**/network/cgi/netw**	GET	Network\	IndigoVision web console acc			**AV:N/AC:L/Au:N/C:P/I:P/A:P**
6492	0	b	**/sitefinity/Login.as**	GET	checkForl	Telerik Sitefinity CMS login fc			**AV:N/AC:L/Au:N/C:P/I:N/A:N**
6493	3092	1	**/cms/**	GET	200	This might be interesting...			**AV:N/AC:L/Au:N/C:P/I:N/A:N**
6494	3092	1	**/helpdesk/**	GET	200	This might be interesting...			**AV:N/AC:L/Au:N/C:P/I:N/A:N**
6495	0	1b	**@PHPMYADMIN**	GET	200	phpMyAdmin directory found			**AV:N/AC:L/Au:N/C:P/I:N/A:N**

Figure 5-5. Security tester – Example of Nikto security tests

- **HTTP Data.** HTTP Data to be sent during POSTs tests.
- **Headers.** Additional headers to send during tests.
- **CVSS Base Vector.** Show the exploitability and impact values to be considered by the SBENCH Plugin for each positive test. Each Base Vector was manually assigned following the CVSS criteria described in Chapter 2.

5.3.5.2 Test Executor

The Test Executor is the component that actually runs security tests against the benchmark target (web servers), and it is implemented by the Nikto Plugin Component. This is a Nikto plugin that is called by the Benchmark Controller of the static part. This Nikto plugin gets one test at a time from Nikto test database and executes it against the web server under benchmark. The test is done by executing a HTTP request with the URL page defined as a test database. Then, it calls the Data Collector and Analyzer to check if the test was successful or not. If a known vulnerability was found (i.e., the test was successful), it calls the component that measures the vulnerability risk based on CVSS values we assigned. Examples of plugins available in Nikto are the CGI plugin (Enumerates possible CGI directories), the Cookies plugin (Looks for internal IP addresses in cookies returned from an HTTP request), and Tests plugin (uses standard Nikto tests).

5.3.5.3 Data Collector and Analyzer

An important aspect to verify if a known vulnerability actually exists is the evaluation of each security test executed. Nikto already has a plugin to collect the response of the web server and to compare the result with the expected value registered in the test database. More specifically, there is a variable that stores the response of the web server that is latter used to check if the test was successful (a

known vulnerability was found) or not. The analysis of the web server response is organized in two parts: test verification and the estimation of the vulnerability risk.

To verify if a test was successful, the web server response is compared with the expected output (match1) of each test. If the result matches, this means that the test was successful and that a known vulnerability is present in the targeted web server.

To estimate the vulnerability risk, we developed a Security Benchmark Plugin (SBENCH Plugin) and added it into the Nikto tool. For each positive test, it estimates the vulnerability risk (in decimal format) based on the impact and exploitability factors.

5.3.5.4 Report Generator

The purpose of the Report Generator is to provide the output needed to confirm the existence of a known vulnerability and the input needed to estimate the security risk of the static part. The Report Generator is implemented in Nikto by the SBENCH and Core plugins. The SBENCH plugin generates a summarized version of Nikto results in CSV format. The core plugin provides the standard Nikto output. Both of them are described as follows:

- *CSV output.* This report was added in Nikto to provide a consolidated view of the final results. This function is called at the end of tests execution, gets the data sent through input parameters and writes the data into a text file. The fields covered in this report are as follows: date, web server brand, total tests, total errors, total vulnerabilities, number of failed test, percentage of failed tests, and security risk.
- *Standard output.* This consists of a log file registering each step of a test execution and also providing debugging information that could be used for getting more details about the system target and test execution.

5.3.6 Static Risk Calculator

The risk calculator is a very important element in the implementation of the static part as it consolidates the results from the Vulnerability Extractor and Analyzer and from the Nikto Testing tool. This component is implemented as a Java class that, once executed, gets the information from CVSS reports and estimate the vulnerability risk, which are added up to form the security risk of the static part.

5.4 DYNAMIC PART IMPLEMENTATION

This section presents our implementation for the components of the dynamic part, aimed at assessing the effects of unknown vulnerabilities in web applications over web servers. The components of the dynamic part of our security benchmark methodology are the Experiment Controller, the Workload, the Vulnerability Injector, the Attackload, the Security Checker, and the Dynamic Risk Calculator. This section describes the implementation of each one of these components.

5.4.1 Experiment controller

The experiment controller orchestrates the execution of all modules necessary to run the dynamic part of the security benchmark. This controller is based on Powershell scripts (Powershell 2013) and command-line scripts that can be executed either by the benchmark user or by the startup script of the operating system. Also, it is deployed inside the system under benchmark and keeps the benchmark output and configuration files in the security benchmark workspace.

The Powershell modules that are triggered by the experiment controller are illustrated in Figure 5-6 and described next. These modules start and stop web serving system components and collect information that is used to compute the security benchmark metric.

- The Precondition module makes sure that all web servers, database and resource monitors are properly stopped before each benchmark run. This is aimed at avoiding any influence of previous benchmark runs into the current run.
- The Diagnose module starts the resource monitors to collect information about CPU, Memory, Disk Usage during the benchmark run. This information are collected by Microsoft Windows Perfmon counters. Each web server has its own collectors defined in a counter configuration file. This counter file is used by the Logman Windows Monitoring application (Logman 2013), which is started by the diagnose module on server side.
- The database module starts and stops (when needed) the database used by the benchmark target application.
- The web server module starts and stops in the proper time the Windows service related with the web servers under benchmark.
- The Client application module starts in the client machine (using psexec command (PsExec 2013)) the workload and attackload client applications according to the parameters defined in the configuration file. These

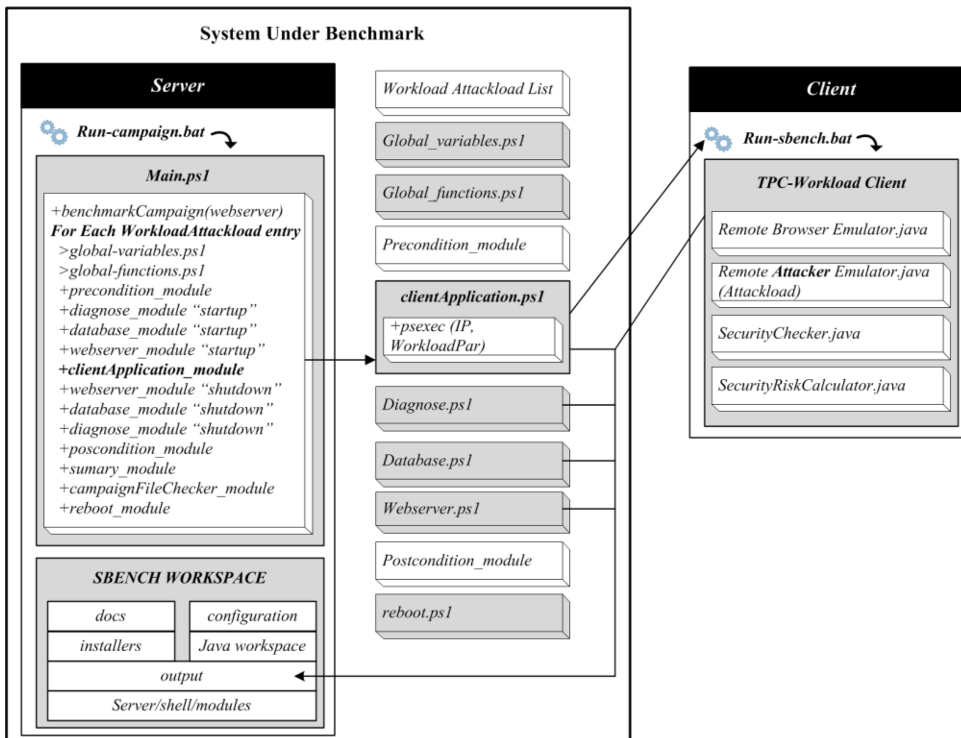

Figure 5-6. Experiment Controller Dynamic Part

applications are started only when the web server component is up and running.

- When the total number of attacks per benchmark run is reached (this number is also defined in the configuration file), the Main.ps1 script calls the stop functions of web server, database, and diagnose module.
- Once all applications are stopped, the Postcondition module is run to collect and consolidate benchmark reports in CSV format and store summarized reports at the benchmark output folder.
- The reboot module is a feature we developed to restart the operating system after a benchmark run. However, we decided to keep this feature disabled, since we used the precondition module to make sure that all processes started during the benchmark were killed and that resource monitors were stopped.

The experiment controller is also supported by a configuration file where it is necessary to set the number of benchmark runs and the parameters of the

workload and attackload application for each web server under benchmark. This configuration file is used by the workload and attackload components during the benchmark run. It contains the identification of the benchmark run, the benchmark execution type (baseline run, or attackload run), the number of emulated users, and the number of emulated attacks.

5.4.2 Workload Implementation

The workload of our security benchmark simulates the activities of a content management application and is formed by the Benchmark Target, the Web Application, the Vulnerable Component, and the Client Workload. The Benchmark Target corresponds to the web servers we benchmarked in the case study described in Chapter 6 (e.g., Apache 1.3, Apache 2.2, Lighttpd 1.4). The Web Application was implemented using the Wordpress Content Management Software (Wordpress 2015b). The Vulnerable Component is a vulnerable instance of the Wordpress web application. The implementation of the Client Workload was done by adapting a Java implementation of the TPC-W Benchmark developed at the University of Wisconsin in 1999 (Java TPC-W 2013).

5.4.2.1 Web Application Implementation

We used an existing application to implement this part of the workload: the Wordpress web application, a full content management system with millions of users and thousands of plugins, widgets and themes. The reason to choose Wordpress is supported by two factors:

- There are more than 70 millions Wordpress Sites in the World. In fact, Wordpress is one of the most popular blogging web applications, with over 409 million people visualizing more than 19.6 billion Wordpress pages on the Internet per month. These statistics was extracted directly from Wordpress Official web page (Wordpress 2015a) and demonstrate the huge popularity of this web application.
- A severe vulnerability in production Wordpress pages will necessarily affect millions of pages. Examples of recent Cross-site scripting and others vulnerabilities reported in (NVD 2014) are CVE-2013-2205, CVE-2013-3253, CVE-2013-0305, CVE-2013-2201, CVE-2013-0236.

After installing the Wordpress application, we configured each one of the web servers under benchmark to point to this directory. Once properly configured, Wordpress application allows users to add, remove, update, and search for posts and comments. There is also a setting web page where the administrator can change Wordpress template, approve comments, and manage the entire page.

The database management system used by Wordpress is the MySQL Database. This database is automatically configured by an automated script incorporated in the Wordpress Application. In fact, when we access Wordpress site for the first time, the application guides us through the database configuration. The database's tables include comments, posts and user data.

5.4.2.2 Vulnerable Component Implementation

The Vulnerable Component is an instance of the Web Application (Wordpress) with vulnerabilities that we injected. The way we implemented these vulnerabilities is described in Section 5.4.3.

We also created an independent database instance for Wordpress application subject to vulnerability injection. This means that the attacks conducted against the hacked application does not affect the database instance of the Benchmark Target.

Note that both our Web Application and the Vulnerable Component are hosted in the same web server, responding simultaneously to users requests. Both applications remain the same across all benchmark run. The client workload makes requests to the Benchmark Target that forwards it to the Web Application, while the attackload makes malicious requests to the Vulnerable Component. As specified in our security benchmark methodology, security measurements that are used in the security benchmark are only collected from the Benchmark Target responses.

5.4.2.3 Client Workload Implementation

Our client workload is an adaptation of the Remote Business Emulator (RBE) of the TPC-W Benchmark Implementation developed at the Department of Electrical and Computer Engineering and Computer Sciences Department of the University of Wisconsin – Madison (Java TPC-W 2013). RBE is the main component of the TPC-W Benchmark and is a specification for a set of Emulated Browsers (EBs), which simulate multiple concurrent web browsing users, each of them making autonomous requests to a web server for web pages and images. During its execution, an EB works in the same way a user navigates a web page, clicking one hypertext link after another in a web browser.

The RBE implementation we use is available at (Java TPC-W 2013). It is coded in Java programming Language and its architecture is described in (Cain et al. 2001). This implementation is conceived to make requests to the BookStore Application specified in TPC-W Benchmark. The implementation of this web application is also provided in (Java TPC-W 2013) in the form of Java Servlets.

However, we changed RBE Java code to make requests to our benchmark target - the Wordpress Content Management application.

At this point, it is clear that our benchmark workload does not follow the TPC-W specification in what regards the web application implementation (e.g., we do not simulate the activities of a retail store web site, we do not take into account performance metrics such as the number of web interactions per second). However, we believe this does not represent an issue at all, since TPC-W benchmark is designed to benchmark performance, and not security, and the purpose here is to use the elements of TPC-W that would be useful to emulate our client workload. Even so, the requirements of TPC-W for RBE that were implemented in (Java TPC-W 2013) are also covered in our work (for example, ramp up and down periods, communication using TPC/IP protocol, and so on).

5.4.3 Vulnerability Injector Implementation

The components of the Vulnerability Injector are: Content Collector, Injection point locator, Content Mutator, Vulnerable Content Generator, Content Deployer. Due to the complexity of developing an automated solution to inject vulnerabilities in Wordpress code and automatically mounting the respective attack, we decided to inject the vulnerabilities manually, as the effort of developing the tool would be much higher than injecting the vulnerabilities manually for the purpose of the research work of this thesis. This means that our vulnerability injector is not implemented in the form of a tool. We selected the injection points and injected the most representative vulnerabilities targeting PHP web applications. For this reason, it was not necessary to implement the Vulnerability Operator component, which stores the set of pairs of location pattern and vulnerability code change.

5.4.3.1 Content Collector Implementation

The content collector is represented by the actions we took to get Wordpress source file. Since this application is written in PHP language, and given that this language does not require compilation of the source code, what we had to do was to download the Wordpress application from the Official Web Application Web Site (Wordpress 2015b). Source code files were downloaded and put in the web server folder that hosts web applications. Then, we started looking for injection points.

5.4.3.2 Injection Point Locator Implementation

The technique we used to find injection points in Wordpress application was to

search for input parameters that could be manipulated by attacker and verify if the code could be changed to a vulnerable state. For example, Wordpress Search page (Search.php) allows users to query specific information in the entire site by providing a search entry. We noticed that once we clicked on the Search button the search entry appeared in the browser URL filed as a value of the ‘s’ input parameter:

```
http://192.168.56.101/wordpress-hacked/?s=hello
```

After observing this input parameter, we went to the Search page code and verified that there was a ‘s’ input parameter that was being handled internally and could be mutated to a vulnerable state. Following this approach, we identified 10 injection points in five different web pages (Header.php, Search.php, Archive.php, Category.php, and Wp-login.php).

5.4.3.3 Content Mutator Implementation

The purpose of the content mutator was to inject into the Wordpress PHP code Cross-site scripting and SQL Injection vulnerabilities. These are the most representative vulnerabilities for the PHP application class (J. Fonseca and Vieira 2008). We did not require mutation operator for the web interface because these attacks exploit vulnerabilities existing in the http protocol. In the Attackload Implementation section 5.4.4, we provide more details about these attacks.

We changed the code of Wordpress to remove sanitation of input parameters in order to inject cross-site scripting vulnerabilities (note that we are changing the vulnerable component, not the system directly under evaluation). By default, Wordpress has several functions to test if the input parameters contain characters that could be used as attacks. The code change consisted in calling the input parameter without these functions. More specifically, we simply print a message in the web page containing each parameter, to make sure that this will appear on client front-end. For example:

```
<?php echo "Missing Function Call: ".$_GET['s']; ?>
```

As ‘s’ parameter are printed without any validation, this portion of the PHP code is vulnerable to Cross-site scripting attacks.

To inject SQL injection vulnerabilities, we realized that no action was necessary, since the input parameters were already being used in SQL queries without the proper sanitization. An example of query for the search parameter is as follows:

```
    <?php $var2 = $_GET['s_id'];
$results = $wpdb->get_col("SELECT link_url FROM wp_links WHERE
```

```
link_id = $var2");?>
```

5.4.3.4 Content Deployer Implementation

The implementation of the content deployer was simple. We moved the vulnerable application to the place where the web servers were hosting the web applications. The goal was to make sure that the vulnerable component would run along with the web application during the execution of the workload.

5.4.4 Attackload Implementation

The components of the attackload are the Vulnerablity Injector, the Attackload Generator, and the Attackload executor. Given that the vulnerability injector implementation was already described, this section describes the attackload generator and executor. These components are also implemented with Java technology.

5.4.4.1 Attackload Generator Implementation

Five attack types are implemented in the attackload. With the exception of denial of services attacks, we emulated 1 attacker executing 20 attacks per benchmark run.

Two attack types were implemented to exploit the vulnerabilities we injected and are described as follows:

- **Cross-site Scripting Attack.** This attack is generated by using a set of malicious characters (“<script>alert(/XSS 001/)</script>”) against the vulnerable input parameters. 15 attacks of this type (three for each input parameter) were implemented.
- **SQL Injection Attack.** This attack was implemented by sending malicious SQL content to get more data from the database (parameter=’1 OR 1=1’) than expected. 15 attacks of this type (three for each input parameter) were implemented.

Three attacks types were implemented against the web server interface through the HTTP protocol: Code Injection Attack, Buffer Overflow, and Denial of Service Attack. These attacks were chosen since the exploited vulnerabilities are among the most representative considering the web server interface.

- **Code Injection Attack.** This attack sends malicious code to HTTP code fields. For each field, the attack strings presented in Table 5-1 were implemented. These strings were chosen based on the Path Traversal cases we found in known vulnerabilities reports of web servers. Another

important aspect is that, for each of the five HTTP fields, only one HTTP method was attacked: GET. This means that we implemented 15 code injection attacks (5 fields x 3 strings x 1 method).

- **Buffer Overflow Attack.** This attack sends large amount of HTTP data to the web server under benchmark. We attacked the same HTTP fields listed in the Code Injection description using the same set of attack string. The difference, however, was that each attack string was increased 10 times in order to test the behavior of the web server in the presence of large amount of data in HTTP protocol. Attacks using GET and POST HTTP methods were implemented, which means that another 15 attacks constitute the Buffer Overflow attackload.
- **Denial of service attack.** This emulates 20 simultaneous attackers, sending 200 Code Injection or Buffer Overflow attacks to web server interface. The purpose here is to flood web servers with dozens of malicious requests. This is done by setting the RBE parameter responsible for determining the number of users that are emulated during a benchmark run.

5.4.4.2 Attackload Executor

The attackload executor is implemented using an instance of the Remote Browser Emulator of the Java Implementation of TPC-W. The difference is that the RBE of our attackload sends malicious content to the vulnerable component and the web server interface.

The Benchmark Manager of the Client Workload also contains a feature to call the RBE with attacks (attackload) to execute the client workload and the attackload at the same time. However, the attackload starts only after the ramp up

Table 5-1. HTTP Protocol Attacks

HTTP Code Field	String 1	String 2	String 3
Accept	"../../../../..///"	"../../#$%&#$ %&"	"99999999999 9999999"
Keep-Alive			
Connection			
Content-Type			
Content-Length			

period of the workload, and it is not executed if the benchmark is running in baseline/inspection mode.

It is worth noting that the malicious requests are sent to the vulnerable component one at a time. More specifically, attacks targeting a particular input parameter are done several times in the same benchmark run. However, no other input parameters are exploited and no other attack types are executed. For example, while Cross-site scripting attacks are done against the search input parameter, no SQL injection, Buffer Overflow, and Code Injection attacks are executed. This is done to make sure that benchmark security measurements for a given benchmark run reflects the behavior of web server in the presence of a single attack type against a single injection point.

5.4.5 Security Checker Implementation

The components of the security checker are the data collector, the confidentiality checker, the integrity checker, the availability checker, the exploitability checker, and the security report generator. After each request to the client workload, this Security Checker is executed to test if there was a partial or complete violation of security.

5.4.5.1 Data Collector

The data collector is implemented inside the Java client workload and as a monitor that run on server-side to collect CPU, Memory, and Disk Usage. The following data is captured after each web server request:

- Response Time. This is the duration the web server takes to respond a request.
- Resource counters (CPU, Memory, Disk Usage). This is captured during the execution of the benchmark.
- HTTP Response Code. HTTP Protocol provides a response code for each request. If the request has been accepted for processing, for example, the code is 202. If the web content is not found, the code is 404.
- HTML page. The entire HTML page of each request is also collected.

5.4.5.2 Confidentiality Checker

The confidentiality checker is a Java method that receives the following input parameters:

- A URL that requires user's authentication. This refers to the administrator

page of Wordpress web application.

- A URL that lists the files inside the web application. If directory listing is allowed by the web server, attackers may access confidential information that could enable them to mount attacks. Obviously, all web servers covered in our case study have ‘directory listing’ turned off.

Two tests are implemented to check confidentiality violations:

- After each client workload request, the security checker tries to access the URL that needs authentication. If the HTTP code is different than forbidden, then there was a confidentiality violation.
- Check if the URL is protected against directory listing. Basically, the expected content of baseline run is compared with the URL content during the execution of attacks.

5.4.5.3 Integrity Checker

The integrity checker is implemented in the form of a Java method and has two input parameters: the HTTP code and the content of each web page that is requested by the client workload.

Two tests are implemented to check integrity violations:

- It checks if the HTTP code of a Wordpress web page, which is expected to be properly working, returns an error.
- It checks if the content of each requested Wordpress web page is different from the one collected during the benchmark baseline run. If a response from the web server changed to the client workload (where no vulnerabilities were injected), then there was an impact on the integrity attribute.

5.4.5.4 Availability Checker

The availability checker uses the response time of the web server to check if the web server is responsive or not. This works as follows:

- For each request sent to the client workload, a response time is estimated.
- If the response timeout is reached, then the availability checker concludes that the web server is unresponsive.
- If the web content returned is null, then the security checker concludes that the web server is also unresponsive.

5.4.5.5 Exploitability Checker

The exploitability checker provides the input to the estimation of the probability of a vulnerability exploitation, considering the attack vector, the attack complexity, and the needed authentication. However, from an implementation point of view, it was not necessary to execute tests to check vulnerability exploitability. In other words, exploitability values were manually setup in the equation of the exploitability checker and no automated verification was needed during the conduction of attacks.

Considering the attacks we executed, the attacks conducted are mounted over vulnerabilities with the highest level of exploitability, i.e., they can be exploited from a remote network, with no need of authentication, and require elementary computer expertise to be exploited.

5.4.5.6 Security Report Generator

This component is implemented as a method of the Security Checker Java class. It receives the impact status (None, Partial, or Complete) provided by the security checker components and provides a report at the end of the benchmark run.

5.4.6 Dynamic Risk Calculator

The risk calculator of the dynamic part was implemented as a Java class integrated in the client workload. For each request done by the client workload, the dynamic risk calculator estimates the vulnerability risk. More specifically, the security checker verifies if there was any compromise in one of the security attributes of confidentiality, integrity, and availability and then calls the risk calculator class to estimate the impact factor. Then, the risk calculator uses the input provided by the exploitability checker to estimate the probability factor of the risk equation. Finally, the vulnerability risk is estimated, which simply consists in the product of the impact and probability factors. This risk is estimated and logged in benchmark CSV reports.

5.5 BENCHMARK RESULT CONSOLIDATION

Once the static and dynamic parts of the security benchmark are executed, two independent CSV files containing the risk of vulnerabilities discovered in the static part and successfully exploited in the dynamic part. At this stage, it is necessary to use the equation of our security benchmark and follow the steps defined in Chapter 3 to obtain the security benchmark metric. We did not implement any tool to automate this result consolidation because benchmark users are free to define the weight of the components to estimate the security risk. This

requires low computer expertise, as what needs to be done is to consolidate vulnerability risk values and follow metric composition procedures to obtain the security benchmark measure.

5.6 CONCLUSION

This chapter presented the implementation of our security benchmark methodology for web serving systems focusing on the web server component. This implementation complies with the benchmark component rules specified in Chapter 3 and 4, detailing the tools of the static (known vulnerabilities) and dynamic parts (effects of unknown vulnerabilities).

The implementation of the static part is formed by the Vulnerability Extractor Analyzer (VEXA) and Nikto Security Testing Tools, respectively targeting the two main components of the static part: the Vulnerability Repository and the Security Test Repository. The VEXA tool was implemented in Java programming language and is aimed at collecting and analyzing known vulnerabilities from vulnerabilities reported in the Open Source Vulnerability Database and the National Vulnerability Database. VEXA connects into a MySQL local instance of these public databases to get vulnerability information. These local instances are updated in a regular basis to make sure that the benchmark covers the most recent known vulnerabilities reported in the field.

Nikto is a widely used Web Server Security Tool that was incorporated in our tool suite. Nikto is open-source and written and Perl language and has a large set of security tests that represents our security test repository. It was improved to estimate the risk of vulnerabilities based on the result of each security test. Both VEXA and Nikto tools are started by the static part experiment controller, which is composed by Microsoft DOS scripts run by the benchmark user.

The implementation of the dynamic part was also provided, focusing on the description of the workload and attackload components. The workload was implemented to emulate a realistic web serving system scenario, with web servers handling the requests of a client workload and forwarding them to a PHP-based web application that connects to a MySQL database. The client workload is a Java implementation of the TPC-W Remote Browser Emulator, which makes requests to the web application following the criteria of the TPC-W specification. The web application is a widely used content management system, the Wordpress. The attackload was implemented to stress web servers and its contents (in our case, the web application) with realistic attacks, built upon an instance of the TPC-W Java application. This instance targets the web application and web server interface with attacks. Five attack types were implemented: SQL Injection; Cross-site

scripting; Buffer Overflow, Code Injection, and Denial of Service targeting the HTTP Header of web servers. The content-related attacks (SQL Injection and Cross-site script) target vulnerabilities that were injected manually in an instance of the Wordpress application. We basically weakened the code to accept requests from the client application without a proper validation and deployed it along with the applications hosted by the web servers used in our case study (Chapter 6). To coordinate the execution of these tools, we developed a set of Powershell scripts that are organized in modules to start and stop monitoring resources, databases, web servers, web applications, and client applications.

We believe that the technical details we provided in this chapter will reduce the time needed to develop new security benchmarks not only for web serving systems, but also for other domains. The reason for that is that we demonstrated how to build a vulnerability repository, to adapt a widely used security testing tool, to inject vulnerabilities into a widely used web application and mount a realistic set of attacks, also taking advantage of technologies that are largely adopted by the software development community (Java, PHP, MySQL, and etc.). We also believe that our implementation will decrease the effort to build future security benchmarks targeting web servers, as we developed tools to extract and analyze known vulnerabilities and stress their security using realistic attacks. Although our implementation was–built for the Windows Platform, the fact that we presented our implementation components along with the several code examples and implementation steps make us confident that we achieved the purpose of demonstrating the feasibility of our security benchmark methodology from a technical standpoint.

CHAPTER 6

6. CASE STUDIES

This chapter presents two case studies that demonstrate the effectiveness and practicability of our security benchmark methodology (Chapter 3 and 4) and implementation (Chapter 5) in real world experiments. These case studies are complementary, showing the applicability of our methodology over components and systems of different sizes and configurations, also enabling users to take advantage of benchmark parts in an independent way if needed. Case study supporting material is available at (Mendes 2015). This was done to increase the confidence on our case studies, enabling users to access the data used to estimate the benchmark results provided in this chapter.

The first case study shows our security benchmark targeting real web servers. Both static and dynamic parts of our methodology were applied to obtain the security benchmark metric, which enabled us to identify the most secure among the web servers under benchmark. The particularity of this case study is that it targets a specific web serving system component (web servers) and uses both benchmark parts to estimate the security level of web servers. Although we believe that this case study would be sufficient to support our claim that our methodology can be applied to measure the security of systems for comparative purposes, it is important to provide examples targeting other systems.

The second case study, which is also a benchmark illustration, covers more web serving system components (databases, web applications, web servers, and operating system), organized in two groups: components with the same technologies but with different versions and configuration (Wordpress, Apache, MySQL, Suse Linux) and systems with different technologies (Wordpress/Dot Net Nuke/Open Java, Apache/IIS/Tomcat, MysQL/Oracle/PostGreeSQL, Linux/Windows). Given the complexity and labor needed to build the experimental setup of the dynamic part covering the vulnerabilities and attacks of all systems targeted in this case study, only the benchmark static part (known vulnerabilities) is used to obtain the security benchmark metric. No attacks were

conducted against the components under benchmark. This is particularly useful to those interested in measuring system security just considering vulnerabilities that were discovered in a specific system brand and version.

The remainder of this section is described as follows. Section 6.1 presents the case study of our security benchmark for web servers, including both static and dynamic part results. Section 6.2 shows the case study of our security benchmark for web serving systems, covering only the results of the static part. Section 6.3 presents the validation of our security benchmark in what regards properties such as representativeness and portability. Section 6.4 concludes this chapter.

6.1 BENCHMARKING THE SECURITY OF WEB SERVERS

This section presents the results of our security benchmark targeting widely used web servers. The security tests of the static part and the attacks and supporting-tools of the dynamic part address vulnerabilities present in HTTP web servers.

6.1.1 EXPERIMENTAL SETUP

The experiments were conducted in a client-server Oracle Virtual Box (VirtualBox 2014) machine environment, using two distinct virtual machines, running two independent operating systems (one to host the web server with its hosted application and another one to host the workload client applications). It is worth noting that virtual machines are a central piece of cloud computing infrastructure (Armbrust et al. 2010), which has been increasingly adopted by enterprise to reduce infrastructure and license costs and, among other aspects, reduce data recovery time (Amazon has over a million customer using its cloud system in 190 countries, (AWS Amazon 2015)). Also, another reason that led us to use virtual machine was to be prepared to rapidly rebuild our experimental setup in the advent of any unexpected system disruption. The system configuration is the same on both virtual machines: Intel Quad Core 2.40 GHz, 1024 MB, 20 Gb IDE Hard Disk, and 100Mbit/s PCnet FAST III Network Adapter. The system configuration of the server that hosted the virtual machines is as follows: Intel Quad Core 2.5 GHz, 8 GB RAM, 465 GB of Hard Driver, running over Windows 7 64-bit operating system. On client side, Java technology is installed in order to enable the execution of workload tools (Java SE Runtime Environment 1.6). On server side, web servers are installed according to the version and configuration described next.

6.1.2 SYSTEMS UNDER BENCHMARK

Table 6-1 presents the systems that were benchmarked. We choose these six web

Table 6-1. System Under Benchmarking

ID	*Benchmark Target*	*Workload*		Operating System
	Web Server	**Web App**	**DBMS**	
IIS51	Internet Inf. Service 5.1	Java TPCW + Wordpress Content Management 3.0.1	MySQL 5.1.43	Windows XP Service Pack 3 + Benchmark Controllers (scripts)
AP13	Apache HTTP Server 1.3			
AP22	Apache HTTP Server 2.2			
LT14	Lighttpd 1.4			
IBM7	IBM HTTP Server 7.0			
OR11	Oracle HTTP server 11			

servers as the benchmark target since they are representative of the market share (namely Apache with 36% of market shared in December 2015 and Microsoft IIS with 27% of market shared in the same period) (Netcraft 2015), can be easily acquired (Apache and Lighttpd are in fact free), and were built by different teams (which means that probably we can find different security issues on them). On IIS and Ligtthtpd web servers, a CGI plugin was used to enable the execution of PHP pages. All web servers configured with the typical security settings, (e.g., configured to disallow directory listing). Oracle and IBM HTTP servers are Apache-based web servers, with a customized version of apache code. These servers were included in the case study to allow us to evaluate the security level of open-source code that is customized by vendors to enable the deployment of web application of large companies such as Oracle and IBM. The former is delivered inside the Oracle Web Tier portfolio, and the latter is within the IBM WebSphere application server. It is important to clarify that the Web Application, the Wordpress content management system, is the same for all the web servers. The emulation of users making request to this application was done using a Java implementation of TPCW. The fact that we use different technologies does not affect the benchmark results, as the Java TPCW application was applied to emulate the requests made to the web servers, being deployed in the client side, and only the PHP application has the vulnerabilities that are exploited by the benchmark attackload.

In each benchmark run only one out of six web servers is started at a time. If the user types a URL and no HTML page is found, instead of listing the content of the folder, a forbidden page is shown. This was done to enable our security checker component to verify confidentiality violations.

6.1.3 BENCHMARK RESULTS & ANALYSIS

The results of the security benchmark are presented in several tables. At the end of this section, we present consolidated results indicating the ranking of the most secure web servers according to our benchmark.

6.1.4 STATIC PART RESULTS

Table 6-2 presents the results of the static part benchmark. Our observations about the data gathered in the static part are as follows:

- 304 web servers vulnerabilities were collected from the field and analyzed by our vulnerability extractor tool.
- 65% of the collected vulnerabilities can be categorized in terms of their representativeness: Information Disclosure (17%), Cross-Site Scripting (14%), Input Validation (13%), Resource Management Errors (10%), Directory Traversal (8%). One example of input validation vulnerability we found was the lack of scape methods on HTTP Header fields. XSS was found in contents present in a default web server installation.
- 26% of the vulnerabilities are from IIS51, 21% are from AP13, 15% are from AP22, IBM7 and OR11, and 8% are from LT14.

Table 6-2. Static Part Benchmark Results

Vulnerabilities	**IIS51**	**AP22**	**AP13**	**LT14**	**IBM7**	**OR11**	***All***
#Unique Vulnerabilities (VEXA)							
Dir. Traversal	16%	0%	16%	4%	0%	0%	8%
Inf. Disclosure	18%	15%	17%	26%	15%	15%	17%
Input Validation	24%	11%	8%	9%	11%	11%	13%
Res. Errors	4%	17%	6%	30%	17%	17%	13%
XSS	9%	17%	17%	0%	17%	17%	14%
Others	29%	39%	36%	30%	39%	39%	35%
Total Vul.	***79***	***46***	***64***	***23***	***46***	***46***	**304**
# Security Tests (Nikto)							
Negative %	99.94	99.97	99.92	99.95	99.95	99.95	99.69
Positive #	4	2	5	3	3	3	20
Total Tests	*6456 against each web sever*						
$SSRL_{SP}$	126.3	91.9	115.1	42.9	94,8	94,8	
$SSRM_{SP}$	182.6	50.8	123.8	49.5	50,8	50,8	
$SSRH_{SP}$	63.8	10.0	30.0	7.3	10	10	
$SBench_{SP}$	**96.6**	**24.3**	**57.7**	**19.6**	**24,4**	**24,4**	

The results of the static part are as follows:

- 0.3% of 6456 security tests were positive (positive here meaning that a known vulnerability was found, in this case, by using Nikto), and different vulnerabilities were found across the evaluated web servers. This low number is due to the fact that hundreds of Nikto tests are mounted for servers that were not targeted in our case study. Table 6-3 details the vulnerabilities that were found on each web server. As can be seen, only vulnerabilities with low risk level were detected. The reason for this is justified by presence of web server content that leads to information disclosure (e.g., a PHP file that contains information on server configuration).

  Another important finding is that, even with directory listing disabled, Nikto tests were able to bypass this and discover folders under Apache 1.3 web server. This suggests that, even by applying security measures such as disabling directory listing, it is necessary to test if a security configuration or code patching actually avoid a given vulnerability. From a security benchmarking standpoint, this strengthens our approach of executing security tests in the static part. Additionally, Wordpress application hosted by web servers is also a point of concern for Nikto,

Table 6-3. Vulnerabilities discovered by Nikto

Discovered Vulnerability	**VR**	**IIS 5.1**	**APA 2.2**	**APA 1.3**	**LT 1.4**	**IBM 7**	**OR 11**
OSVDB-3233: /phpinfo.php: Contains PHP configuration information	2.9	X	X	X	X	X	X
OSVDB-3092: /manual/: Web server manual found.	2.9	-	-	X	-	-	X
OSVDB-3268: /icons/: Directory indexing found.	2.9	-	-	X	-	X	-
/icons/README: Apache default file found	2.9	-	-	-	-	X	X
OSVDB-3268: /manual/images/: Directory indexing found.	2.9	-	-	X	-	-	-
OSVDB-561: /server-status: This reveals web server information.	2.9	-	-	-	X	-	-
OSVDB-3092: /iishelp/iis/misc/default.asp: Default IIS page found.	2.9	X	-	-	-	-	-
/wordpress/: A Wordpress installation was found.	2.9	X	X	X	X	-	-
/webresource.axd?d=junk: ASP.NET reveals its version in error messages when verbose debugging is enabled.	2.9	X	-	-	-	-	-

since this application is a potential target of attack. Overall, since the discovered vulnerabilities have low risk, they do not contribute to change the benchmark result of the static part.

- Considering the estimated component security risk of the static part only, we can rank the evaluated web servers just using the *SBench-sp* metric: LT14 (19.6), AP22 (24.3), IBM7 and OR11 (24.4), AP13 (57.7), IIS51 (96.6). LT14 is the most secure since it has the lowest *SBench-sp* measure.

The following analyses are pertinent considering the results of the static part:

- LT14 is the most secure among the evaluated web servers, having lowest security risk measure.
- AP22, IBM7, and OR11 results were very close to LT14 on *SBench-sp* metric: 24 *SBench-sp* against 19.6. However, if we take into the account only the low security risk metric (SSRL) then LT14 remains the most secure, since it has the lowest score by far: 42.9 SSRL against 91.9 of AP22.
- Security Risk results are not related only to the number of known vulnerabilities present in the system. For example, AP22 has nearly the double of vulnerabilities of LT14, but *SBench-sp* metric is very similar for both systems. This is because both servers have nearly the same amount of vulnerabilities of high-risk category. If we compare the results of ISS51 and AP13, which have a similar number of detected known vulnerabilities, the difference is more visible. Considering only these two servers, AP13 is by far the most secure, since it has fewer vulnerabilities with high security risk.

6.1.5 DYNAMIC PART RESULTS

The purpose of the dynamic part is to assess the behavior of each targeted web server when attacks are executed against its interface (HTTP protocol) and its content (web application). Two classes of vulnerabilities were manually injected in the hosted Wordpress web application: Cross Site Scripting (XSS) & SQL Injection (SQL). The vulnerabilities exploited in the web server HTTP protocol are those related with Code Injection (CI), Buffer Overflow (BO). Also, in order to test the capacity of our web servers to handle a large number of requests we also executed Denial of Service (DoS) attacks.

Table 6-4 summarizes the operational aspects of the dynamic part execution. The number of vulnerabilities (VL) and the number of attacks (AT) for each server

Table 6-4. Total Attacks & Vulnerability & Benchmark Duration

	Per Vulnerability					Per Web Server			After 3 Runs		3 runs for 6 servers	
	#VL	#AT	#USR	#ATK	Dur (min)	#VL	#AT	Dur (min)	#AT	Dur (min)	#AT	Dur (h)
XSS	1	20	1	1	3	5	100	15	300	45	1800	4.5
SQL	1	20	1	1	3	5	100	15	300	45	1800	4.5
CI	1	20	1	1	3	5	100	15	300	45	1800	4.5
BO	1	20	1	1	3	5	100	15	300	45	1800	4,5
DoS	1	200	5	20	4	5	1000	20	3000	60	18000	6.0
	5	280				25	1400	80	4200	240	25200	24

during the dynamic part is given. Each simulated attacker (ATK) executes 20 attacks per benchmark run, and we executed three benchmark runs for each server. The workload is executed via a simulated user (USR) making non-malicious requests to the targeted web servers. In the case of DoS attacks, multiples requests are sent from the attackload client, simulating 20 attackers (each one of them simultaneously executing 10 attacks per benchmark run), while the workload client emulate 5 users sending non-malicious requests to the workload application hosted in the web server. A total of 1400 attacks were executed against each web server having 5 attack categories, with an average duration of 80 minutes per benchmark run (covering all attacks categories). For each server and vulnerability, three benchmark runs were executed, and the result is the average of the measurements collected during each run. This means that for each web server, 4200 attacks were executed within 4 hours. If we consider the 6 benchmarked web servers, we have 25200 attacks executed in 24 hours, which is the total time spent to execute the dynamic part of our security benchmark.

In this case study, the attacks were not able to compromise the confidentiality of web servers (e.g., disclose unauthorized areas of the web site). Therefore, we focus on the analysis of the attributes of availability and integrity only: from a benchmark/comparison perspective, confidentiality is the same for all the systems observed. Table 6-5 presents the loss of availability (percentage of loss relative to baseline) for each of the 6 benchmarked systems. The availability is computed based on the number of requests that the workload client was able to conclude while the attacks were executed. It is possible to observe the following:

- IS51 and LT14 were the only web servers that had loss of availability. It is worth noting that the other servers – including ORA11 and IBM7 – are apache-based web servers, built upon the same technology, but with some customization. We believe that this is the reason why no availability loss

was observed for those web servers.

- Concerning availability, LT14 appears to be more robust to attacks than IS51, since the average loss of availability is lower in LT14 (average of 0.4% for non-DoS attacks against 5% for IS51 for DoS attacks). However, LT14 was affected only by non-DoS attacks, while IS51 was affected during DoS attacks.

Table 6-5. Loss of Availability Results

	AVG Loss of Availability (%) in 3 Benchmark Runs					
	AP22	**AP13**	**IS51**	**LT14**	**IBM7**	**OR11**
XSS03	0,00	0,00	0,00	0,33	0,00	0,00
	0%	0%	0%	0,4%	0%	0%
XSS04	0,00	0,00	0,00	0,33	0,00	0,00
	0%	0%	0%	0,5%	0%	0%
XSS05	0,00	0,00	0,00	0,33	0,00	0,00
	0%	0%	0%	0,4%	0%	0%
SQL02	0,00	0,00	0,00	0,67	0,00	0,00
	0%	0%	0%	1%	0%	0%
SQL03	0,00	0,00	0,00	0,33	0,00	0,00
	0%	0%	0%	0,4%	0%	0%
SQL04	0,00	0,00	0,00	0,67	0,00	0,00
	0%	0%	0%	1%	0%	0%
SQL05	0,00	0,00	0,00	0,33	0,00	0,00
	0%	0%	0%	0,7%	0%	0%
CIG01	0,00	0,00	0,00	0,33	0,00	0,00
	0%	0%	0%	1%	0%	0%
CIG02	0,00	0,00	0,00	0,33	0,00	0,00
	0%	0%	0%	0,5%	0%	0%
CIG03	0,00	0,00	0,00	0,67	0,00	0,00
	0%	0%	0%	0,9%	0%	0%
CIG04	0,00	0,00	0,00	0,33	0,00	0,00
	0%	0%	0%	0,4%	0%	0%
CIG05	0,00	0,00	0,00	0,33	0,00	0,00
	0%	0%	0%	0,4%	0%	0%
BOG02	0,00	0,00	0,00	1,00	0,00	0,00
	0%	0%	0%	1%	0%	0%
BOG04	0,00	0,00	0,00	0,33	0,00	0,00
	0%	0%	0%	0,4%	0%	0%
BOG05	0,00	0,00	0,00	0,33	0,00	0,00
	0%	0%	0%	0,4%	0%	0%
DoS01	0,00	0,00	8,33	0,00	0,00	0,00
	0%	0%	2%	0%	0%	0%
DoS02	0,00	0,00	175,33	0,00	0,00	0,00
	0%	0%	9%	0%	0%	0%
DoS03	0,00	0,00	9,33	0,00	0,00	0,00
	0%	0%	3%	0%	0%	0%
DoS04	0,00	0,00	64,33	0,00	0,00	0,00
	0%	0%	5%	0%	0%	0%
DoS05	0,00	0,00	96,33	0,00	0,00	0,00
	0%	0%	7%	0%	0%	0%

- The LT14 availability issues were not repeatable across the 3 benchmark runs. In fact, availability compromise happened only one time in a particular run. It is possible to see that 2 out of 5 XSS Injection attacks caused no harm to the web server.
- IS51 has its availability impacted during DoS attacks, with an unavailability rate ranging from 2% to 9% of the total requests performed by the workload client. IS51 also had consistent, repeatable availability compromise, since this happened during the 3 benchmark runs and for all DoS attacks conducted.

Table 6-6 presents the results for the loss of integrity (shown in percentage). This was estimated based on the correctness of the replies received back by the workload client. We can observe that:

- Only IS51 and LT14 had loss of integrity to some extent.
- For XSS attacks, both web servers were impacted, having at least 3 out of 5 attacks compromising at least one request during all benchmark runs.
- For SQL attacks, the major impact was on LT14, with 4 out 5 attacks results in at least 1 request with integrity compromise.
- FOR CI attacks, LT14 was the only web server impacted, having from 0.4% to 1% of loss of integrity during the execution of attacks. Also, it was the only one impacted during the BO attacks.
- For DoS attacks, IS51 was the only one with integrity issues. The loss of integrity rate ranges from 21% to 62% in a particular DoS benchmark run. In this former case, most of the requests were either incorrect or missing for the workload client during the execution of DoS attacks.
- 22 out of 25 benchmark runs resulted in impact to at least one of the web servers. Considering that each benchmark run exploit one vulnerability, we have the following result about attack successfulness: LT14 was compromised at least once in 15 runs; IS51 was compromised in 12 of the runs.
- The attack tolerance of the benchmarked web servers is not uniform (results are quite different). This strengthens our belief in using these differences in a security benchmark to rank those systems.

Considering the estimated component security risk of the dynamic part (Table 6-7), we can rank the evaluated web servers as follows: AP2, AP13, IBM7, and OR11 (0 *SBench-dp*, no security compromise across all benchmark run), LT14

Table 6-6. Loss of Integrity Results

	AVG Loss of Integrity (%) in 3 Benchmark Runs					
	AP22	**AP13**	**IS51**	**LT14**	**IBM7**	**OR11**
XSS02	0,00	0,00	0,33	0,00	0,00	0,00
	0%	0%	0.2%	0,0%	0%	0%
XSS03	0,00	0,00	0,33	0,33	0,00	0,00
	0%	0%	0.2%	0.4%	0%	0%
XSS04	0,00	0,00	0,33	0,33	0,00	0,00
	0%	0%	0.2%	0,5%	0%	0%
XSS05	0,00	0,00	0,33	0,33	0,00	0,00
	0%	0%	0.2%	0,4%	0%	0%
SQL01	0,00	0,00	0,33	0,00	0,00	0,00
	0%	0%	0.2%	0%	0%	0%
SQL02	0,00	0,00	0,33	0,67	0,00	0,00
	0%	0%	0.1%	1%	0%	0%
SQL03	0,00	0,00	0,00	0,33	0,00	0,00
	0%	0%	0%	0,4%	0%	0%
SQL04	0,00	0,00	0,00	0,67	0,00	0,00
	0%	0%	0%	1%	0%	0%
SQL05	0,00	0,00	0,33	0,33	0,00	0,00
	0%	0%	0.2%	0,7%	0%	0%
CIG01	0,00	0,00	0,00	0,33	0,00	0,00
	0%	0%	0%	1%	0%	0%
CIG02	0,00	0,00	0,00	0,33	0,00	0,00
	0%	0%	0%	0,5%	0%	0%
CIG03	0,00	0,00	0,00	0,67	0,00	0,00
	0%	0%	0%	0,9%	0%	0%
CIG04	0,00	0,00	0,00	0,33	0,00	0,00
	0%	0%	0%	0.4%	0%	0%
CIG05	0,00	0,00	0,00	0,33	0,00	0,00
	0%	0%	0%	0,4%	0%	0%
BOG02	0,00	0,00	0,00	1,00	0,00	0,00
	0%	0%	0%	1%	0%	0%
BOG04	0,00	0,00	0,00	0,33	0,00	0,00
	0%	0%	0%	0,4%	0%	0%
BOG05	0,00	0,00	0,00	0,33	0,00	0,00
	0%	0%	0%	0,4%	0%	0%
DoS01	0,00	0,00	66,33	0,00	0,00	0,00
	0%	0%	21%	0%	0%	0%
DoS02	0,00	0,00	1222,00	0,00	0,00	0,00
	0%	0%	62%	0%	0%	0%
DoS03	0,00	0,00	93,67	0,00	0,00	0,00
	0%	0%	31%	0%	0%	0%
DoS04	0,00	0,00	486,00	0,00	0,00	0,00
	0%	0%	39%	0%	0%	0%
DoS05	0,00	0,00	682,33	0,00	0,00	0,00
	0%	0%	48%	0%	0%	0%

(0.06), IIS51 (2.38). Apache-based web servers are the most secure web servers since they suffered no harm during attack execution and ISS51 is the most insecure.

Not all attacks are equally harmful and results about the effects of each attack are

Table 6-7. Dynamic Part Results: Security Risk

	AVG Security (SBench) Risk in 3 Runs					
RUNS	**APA22**	**AP13**	**IS51**	**LT14**	**IBM7**	**OR11**
$DSRL_{DP}$	0.00	0.00	6.77	1.24	0.00	0.00
$DSRM_{DP}$	0.00	0.00	8.14	0.00	0.00	0.00
$DSRH_{DP}$	0.00	0.00	0.00	0.00	0.00	0.00
$SBench_{DP}$	**0.00**	**0.00**	**2.38**	**0.06**	**0.00**	**0.00**

pertinent to security practitioners and academia. Considering our experiments, we can extract the following considerations relative to the attacks:

- DoS was the most harmful attack. It was especially harmful to IIS51, since several attacks impact the integrity and availability of this web server. The reason behind DoS attacks being the most harmful is that they extenuated the capacity of web server resources.
- Command Injection and Buffer Overflow attacks against HTTP protocol caused few harm to the web server.
- XSS attacks and SQL Injection attacks caused virtually no harm to the benchmark target across the benchmark runs. On IIS51 and LT14 we observed an integrity impact due to the inability of the web server to provide the expected response collected during the baseline runs.
- Robustness attacks against the system interface are more dangerous over attacks conducted against the web application. This suggests that the weakness propagation between web application and web servers does not represent a threat to the evaluated servers.

6.1.6 BENCHMARK RESULTS

Table 6-8 presents the consolidated results of each part of our security benchmark runs over the web servers we evaluated. According to the results, the most secure web-server is LT14 web server (19.7 *SBench*), which is immediately followed by AP22 (24.3), IBM7 and OR11 (24.4), AP13 (57.7 *SBench*) and ISS51 (99 *SBench*).

Although Apache-based web servers (including IBM7 and OR11) resisted to most of the attacks executed in the dynamic part (lowest rate of unsuccessful requests), they did not obtain the highest security score since its results of the static part affected the benchmark metric. IIS web server was the one with highest percentage of known vulnerability and was not able to cope with multiple attacks, being the most unsecure of the evaluated web serving system components.

Table 6-8. Web Server Ranking

Rank	System	SRH	SRM	SRL	SBench
1st	**LT14**	7,3	49,5	44,1	**19,7**
2nd	**AP22**	10	50,8	91,9	**24,3**
3rd	**IBM7**	10	50,8	94,8	**24,4**
	ORA11	10	50,8	94,8	**24,4**
4th	**AP13**	30	123,8	115,1	**57,7**
5th	**IIS51**	63,8	190,7	133,1	**99,0**

It is worth noting that our benchmark results allow a clear comparison among the benchmark we evaluated, both in the static and dynamic parts, helping users to select the most secure web server. These results are consistent, since the benchmark measures were not biased by the number of vulnerabilities in the static part and reflected the different behavior of the web servers when facing real attacks. This suggests that our methodology is valid and provides meaningful results and it can help to the definition of widely accepted security benchmark standards.

6.2 BENCHMARKING THE SECURITY OF WEB SERVING SYSTEMS BASED ON KNOWN VULNERABILITIES

This section presents the results of our security benchmark targeting different configuration and technologies of web serving systems. Vulnerabilities numbers regarding Apache and Wordpress are slightly different than the ones presented previously since data gathering was performed earlier (Mendes, Duraes, and Madeira 2011). Note that some of the components targeted in the previous case study (Apache and IIS) were also covered in this one.

The results presented in this section were obtained considering the static part execution of our security benchmark. More specifically, we used the Vulnerability Extractor and Analyzer tool to collect known vulnerability from the components brand and versions under benchmark. Then, we estimated the security benchmark metric adding up the risk of the collected vulnerabilities, without the execution of security tests to identify additional vulnerability and without stressing the systems with attacks. It is worth noting that the dynamic part was not applied due to the complexity and labor necessary to build the experimental setup covering all systems targeted by this second case study. In other words, it is possible to execute only one of the benchmark parts to measure the security of systems, taking into account the impact over the accuracy of benchmark results, as our methodology recommends the execution of both parts to have accurate results.

One important aspect in this case study is that two groups of systems are targeted: (1) systems with the same technology but with different configuration and versions and (2) systems with different technologies. Our intent is to demonstrate how our methodology can help users to measure the impact of configuration settings to the security of the system, as a misconfiguration can lead to the successful exploitation of system vulnerabilities. In the same way, administrators can be subjected to different set of vulnerabilities depending on the system technology deployed on their environment and our methodology can be used to help them to identify the most secure.

6.2.1 SYSTEMS UNDER BENCHMARK

Table 6-9 presents the web serving system components under benchmark, the number of component vulnerabilities extracted by VEXA, and the vulnerability sum grouped by web applications (APP), by web servers (WS), by database (DB), and by operating systems (OS). The percentages presented are based on the number of vulnerabilities collected for a given component. This information is characterized according to the following attributes:

- Component identification (CID), brand (Name), version (Ver), and configuration (Conf). This former attribute can be Def (default configuration), Module (referring to the module activated by a configuration directive), and All (full configuration).
- Number of collected vulnerabilities (#CV). This corresponds to the actual number of vulnerabilities extracted by the tool. Users trying to repeat our case study probably will see a similar number from tool reports (these numbers may be somewhat different due to the fact that new vulnerabilities are reported in a daily basis).
- Number of vulnerabilities automatically classified by VEXA (#AC) and which classification remains undefined (#UC).
- Number of multiple vulnerabilities (#ML). This refers to vulnerability reports that include two or more vulnerabilities.
- Number of false-positives (#FP). This includes multiple vulnerabilities and those that, although collected, are not related to the target component (termed as false-positives).
- Number of unique vulnerabilities (#UV). This corresponds to the subtraction of collected vulnerabilities (#CV) with false-positives (#FP). These vulnerabilities are the one considered in the remainder of this section.

Table 6-9. Vulnerability Distribution By System Component

CID	Name	Ver	Conf	#CV	#AC	#UC	#ML	#FP	#UV
Web Applications (APP)									
APP1	**Wordpress PHP CMS**	All	All	266	92 (35%)	17 (6%)	42 (16%)	43 (16%)	**223 (84%)**
APP1a	Wordpress PHP CMS	2.0.1	All	42	17 (40%)	0	9 (11%)	7 (17%)	**35 (83%)**
APP2	**Dot Net Nuke CMS**	All	All	22	9 (41%)	3 (13%)	1 (5%)	1 (5%)	**21 (95%)**
APP2a	Dot Net Nuke CMS	4.0	All	10	1 (10%)	2 (20%)	0	0	**10 (100%)**
APP3	**Open Java CMS**	All	All	15	6 (40%)	0	1 (7%)	3 (20%)	**12 (80%)**
APP3a	Open Java CMS	6.0.2	All	7	6 (86%)	0	0	54 (15%)	**7 (100%)**
APP1 + APP2 + APP3		All	All	303	107 (35%)	20 (7%)	44 (15%)	47 (16%)	**256 (84%)**
Web Servers (WS)									
WS1	**Apache HTTP Server**	All	All	170	122 (72%)	2 (1%)	14 (8%)	14 (8%)	**156 (92%)**
WS1a	Apache HTTP Server	1.3	All	28	14 (50%)	0	0	9 (32%)	**19 (68%)**
WS1b	Apache HTTP Server	1.3	Def	10	8 (80%)	0	0	2 (20%)	**8 (80%)**
WS1c	Apache HTTP Server	1.3	Proxy	13	8 (62%)	0	0	2 (15%)	**11 (85%)**
WS1d	Apache HTTP Server	2.0	All	70	43 (61%)	1 (1%)	3 (4%)	27 (39%)	**43 (61%)**
WS1e	Apache HTTP Server	2.0	Def	35	27 (77%)	0	2 (6%)	17 (49%)	**18 (51%)**
WS1f	Apache HTTP Server	2.0	Proxy	42	27 (64%)	1 (2%)	2 (5%)	20 (48%)	**22 (52%)**
WS1g	Apache HTTP Server	2.0	SSL	46	36 (78%)	0	2 (4%)	19 (41%)	**27 (59%)**
WS2	**Microsoft IIS**	All	All	143	107 (75%)	6 (4%)	4 (3%)	4 (3%)	**139 (97%)**
WS2a	Microsoft IIS	5.0	All	70	54 (77%)	4 (6%)	2 (3%)	2 (3%)	**68 (97%)**
WS3	**Apache Tomcat**	All	All	85	38 (45%)	8 (9%)	5 (6%)	8 (9%)	**77 (91%)**
WS3a	Apache Tomcat	6.0.11	All	22	1 (5%)	0	0	0	**22 (100%)**
WS1 + WS2 + WS3		All	All	398	267 (67%)	16 (4%)	23 (6%)	26 (7%)	**372 (93%)**
Databases (DB)									
DB1	**MySQL Database**	All	All	240	98 (41%)	50 (21%)	15 (6%)	122 (51%)	**118 (49%)**
DB1a	MySQL Database	5.0.0	All	27	10 (37%)	5 (19%)	0	0	27 (100%)
DB2	**Oracle Database**	All	All	386	67 (17%)	279 (72%)	28 (7%)	120 (31%)	**266 (69%)**
DB2a	Oracle Database	10.1.0.5	All	169	26 (15%)	125 (74%)	16 (9%)	16 (9%)	**153 (91%)**
DB3	**PostgreSQL**	All	All	75	23 (31%)	20 (27%)	7 (9%)	12 (16%)	**63 (84%)**
DB3a	PostgreSQL	7.2.1	All	24	8 (33%)	5 (21%)	3 (13%)	3 (13%)	**21 (88%)**
DB1 + DB2 + DB3		All	All	701	188 (27%)	349 (50%)	50 (7%)	254 (36%)	**447 (64%)**
Operating Systems (OS)									
OS1	**Suse Linux**	All	All	53	28 (53%)	13 (25%)	0	13 (25%)	**40 (75%)**
OS1a	Suse Linux	10.0	All	19	8 (42%)	5 (26%)	0	4 (21%)	**15 (79%)**
OS2a	**Windows XP**	XP	All	41	21 (51%)	18 (44%)	0	7 (17%)	**34 (83%)**
OS1 + OS2		All	All	94	49 (52%)	31 (33%)	0	20 (21%)	**74 (79%)**
Total: APP + WS + DB + OS		All	All	1496	611 (41%)	416 (28%)	117 (8%)	347 (23%)	**1149 (77%)**

We formed two groups of web serving systems (Table 6-10). The first one is composed by components with the same technologies but with different versions and configuration (each line of the first column of the table represents a system under this group). For example, SUB1.A contains the components described in Table 6-9, such as APP1a (Wordpress PHP CMS), WS1a (Apache HTTP Server 1.3), DB1a (MySQL Database), and OS1a (Suse Linux). The second group is

composed by systems with different technologies (each line of the second column). The idea is to evaluate how much the security risk of these systems varies when different components or technologies are used, and when different configurations are used. Table 6-10 lists the systems under benchmark (SUB) that were built from the several components/configurations listed in Table 6-9 (identified by the CID attributes).

6.2.2 VULNERABILITY REPORT RESULTS

The total number of collected vulnerabilities is 1496. 41% of these vulnerabilities were automatically classified by VEXA and 23% were discarded during the validation process as they contained multiple vulnerabilities in a single report. The top 5 components with the most unique vulnerabilities reported were the Oracle Database (266), the WordPress Content Management System (223), the Apache HTTP Server (156), the Microsoft IIS (139), and the Apache Tomcat (77).

6.2.3 COMPONENT BENCHMARK RESULTS

Table 6-11 presents the benchmark results grouped by system components. The Component Risk Measure (CR) was estimated by adding the security risk of vulnerabilities present in a given version and configuration of the targeted component according to the rules of the *SBench-sp* Metric equation (a full description of the measure equation can be found at Chapter 3). The comparison of Vulnerability Total with *SBench-sp* shows us that that the number of vulnerabilities is not directly related with the system security risk. The system with the least number of vulnerabilities (WS1b) was scored in the 4th position in our security benchmark rank, since it has several vulnerabilities with medium risk.

The most evident aspect from the component benchmark results is the high *SBench-sp* obtained by the DB2a component, which is the most risky component by a very large margin (351.8 SBench-sp against 62.9 SBench-sp of WS2a). This

Table 6-10. Web Serving Systems Groups

SUB group 1	SUB group 2
SUB1.A: APP1a WS1a DB1a OS1a **SUB1.B:** APP1a WS1b DB1a OS1a **SUB1.C:** APP1a WS1c DB1a OS1a **SUB1.D:** APP1a WS1d DB1a OS1a **SUB1.E:** APP1a WS1e DB1a OS1a **SUB1.F:** APP1a WS1f DB1a OS1a **SUB1.G:** APP1a WS1g DB1a OS1a	**SUB1.D:** APP1a WS1d DB1a OS1a **SUB2.A:** APP2a WS2a DB2a OS2a **SUB2.B:** APP2a WS2a DB1a OS2a **SUB3.A:** APP3a WS3a DB3a OS2a **SUB3.B:** APP3a WS3a DB1a OS1a

is mainly due to the high number of risky vulnerabilities reported to Oracle Database 10.1.5.0. Additionally, DB1a is the less risky database, even though it has more reported vulnerabilities than DB3a (which shows that the number of vulnerabilities alone is not enough to compare components and systems). This means that vulnerabilities reported to DB3a may cause more security violations than those present in DB1a (because they are more exploitable, have more impact, or both).

Considering web applications and operating systems components, our benchmark showed that APP1a and OS2 are the most risky components. The first one is a widely used PHP Content management system with 35 vulnerabilities reported to version 2.0.1. OS2a security risk (33.7 SBench-sp) is much higher than OS1a (4.5 SBench-sp).

Web servers WS1a, WS1b, and WS1c belong to the same brand and version (Apache HTTP Server 1.3) but have different configurations. The benchmark results of these components indicate that the more modules are activated, the higher is their security risk (as expected). It also shows how much the security risk metric decreases by turning off (unnecessary) modules. WS1b security risk is more than two times lower (6.7 SBench-sp) than that of WS1a (14.0 SBench-sp), which have all modules turned on.

An interesting conclusion is that, for this combination of components, a default configuration is less risky (WS1e, 8.1 SBench-sp) than those with all vulnerable modules active (WS1d, 25.4 SBench-sp). Additionally, our results also show that web servers with the same number of activated modules have different risk values (WS1f and WS1g).

These results show the effectiveness of our method and tool to evaluate the security risk of functionally equivalent components, even taking into account factors as the versions and different types of component configurations.

6.2.4 SYSTEM BENCHMARK RESULTS

Figure 6-1 shows the benchmark results for web serving systems with the same technologies but with different versions and configuration (the components integrating these systems are listed in Table 6-10). The bar sections indicate the security risk of each risk category.

In this case study, we consider that all components have the same security relevance level, which means that we assigned the value 1 to the weight factor presented in the benchmark metric proposed in Chapter 3.

Our benchmark results indicate that the system with the less risky web server

Table 6-11. Security Risk By System Component

#	Comp	Description	VUL	CRL	CRM	CRH	SBench-sp
1	**APP3a**	**Open Java 6.0.2 All**	**12**	**10,5**	**10,6**	**0,0**	3,2
2	**APP2a**	**Dot Net Nuke 4.0 All**	**21**	**21,4**	**11,9**	**0,0**	4,0
3	**OS1a**	**Suse Linus 10.0 All**	**15**	**33,7**	**11,3**	**0,0**	4,5
4	**WS1b**	Apache HTTP 1.3 Def	8	11,2	24,7	0,0	6,7
5	**WS1e**	Apache HTTP 2.0 Def	18	38,4	24,7	0,0	8,1
6	**WS3a**	Apache Tomcat 6.0.11 All	22	42,9	24,6	0,0	8,3
7	**WS1c**	Apache HTTP 1.3 Def + Proxy	11	16,6	30,2	0,0	8,4
8	**WS1f**	Apache HTTP 2.0 Proxy	22	48,7	24,7	0,0	8,6
9	**DB1a**	MySQL 5.0.0 All	27	43,1	46,5	0,0	13,8
10	**WS1a**	Apache HTTP 1.3 All	19	32,8	49,4	0,0	14,0
11	**DB3a**	PostGreSQL 7.2.1	21	21,2	71,7	0,0	19,0
12	**WS1g**	Apache HTTP 2.0 SSL	27	50,4	44,4	10,0	20.6
13	**APP1a**	WordPress 2.0.1 - All	35	64,0	63,7	8,6	25.1
14	**WS1d**	Apache HTTP 2.0 All	43	87,1	56,3	10,0	25.4
15	**OS2a**	Windows XP All	34	64,5	37,9	30,0	33.7
16	**WS2a**	Microsoft IIS 5.0 All	68	108,9	156,9	26,0	62.9
17	**DB2a**	Oracle 10.1.0.5 All	153	192,6	237,3	404,0	351.8

obtained the best security risk score (SUB1.B, 50.2 *SBench-sp* - Apache Version 1.3 in a default configuration).

The top 3 most secure systems can then be ranked as follows: 1st SUB1.B, 2nd SUB1.E, and 3rd SUB1.C. They are immediately followed by SUB1.F, SUB1.A, SUB1.G, and SUB1.D, which is the most risky system due to the high number of risky vulnerabilities reported for Apache 2.0 (all modules activated).

Figure 6-2 shows the benchmark results for web serving system with different technologies (e.g., Linux-MySQL-Apache-PHP against Windows-Oracle-IIS-.Net).

The difference shown in the security measure among the five benchmarked systems is quite evident. The security risk rank of these systems is as follows: 1st – SUB3.B (29.8 *SBench-sp*), 2nd – SUB3.A (64.2 *SBench-sp*), 3rd SUB1.D (68.9 *SBench-sp*), 4th SUB2.B (114.4 *SBench-sp*), and 5th SUB2.A (452.4 *SBench-sp*).

Although the winning system was built on a Linux-MySQL-Tomcat-Java-based technology, this does not mean that it is the most secure in all aspects. This means

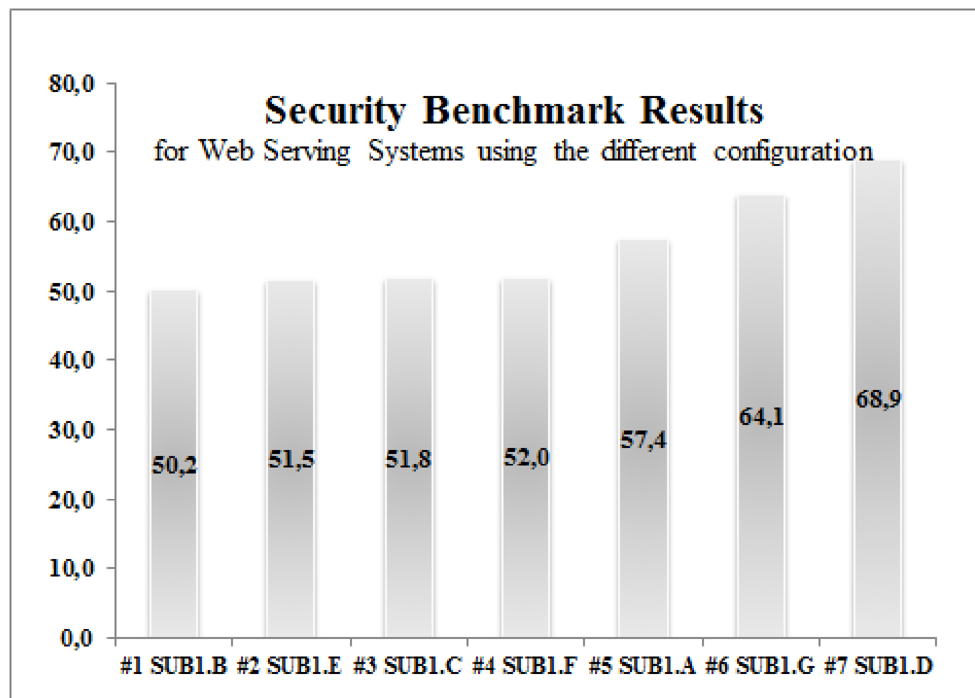

Figure 6-1. Security Benchmark Results for Web Serving Systems using different configuration

that considering the benchmarked components versions and under certain configuration, it obtained the best security risk score. These results cannot be generalized to other scenarios or environments. A simple configuration or component change in this system (e.g., database replacement to DB2a) would drastically alter this benchmark result. The score of the worst ranked system is mainly due to the risky vulnerabilities reported to the Oracle Database.

An important point here is that our method and tool do enable its users to evaluate and compare the security risk of different types of components and web serving systems and observe the security risk variance after changing the version or configuration of a system. This was only possible due to the extraction and analysis of real vulnerabilities from the field by using the VEXA tool.

6.3 BENCHMARK PROPERTIES VALIDATION

This section presents our work to validate our security benchmark against classical benchmark properties (representativeness, repeatability, portability, non-intrusiveness, feasibility). This is done to increase the confidence of benchmark users on our security benchmark methodology and to provide means to the

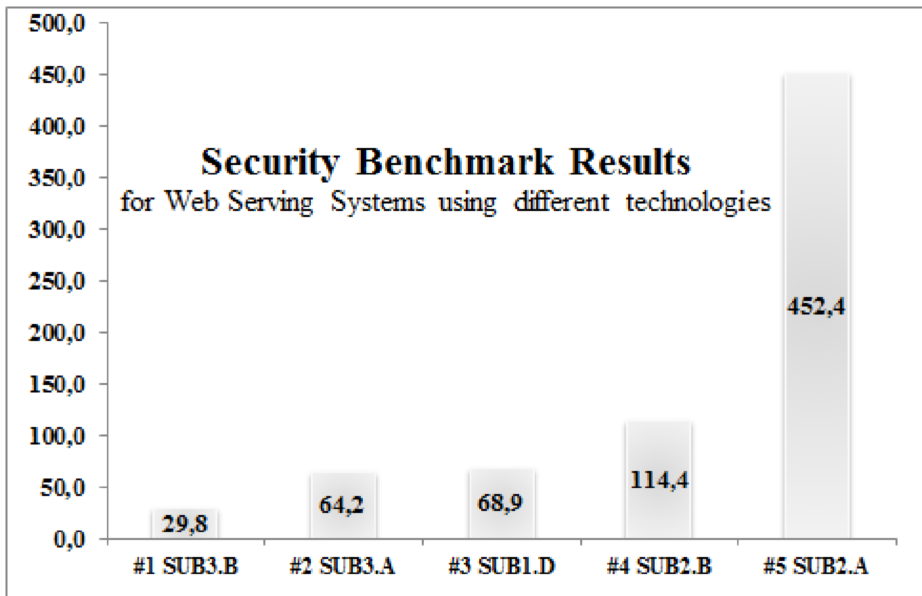

Figure 6-2. Benchmark Results for Web Serving Systems with different technologies

acceptance of our proposal to fundament a future standard security benchmark research.

6.3.1 Representativeness

To be representative, a benchmark should reflect how well the benchmark components characterize the actual context of real systems. As introduced in Chapter 3, our security benchmark is formed by key elements such as metric, workload, attackload, vulnerability and security tests repositories and so on. In this sense, the representativeness of our security benchmark is given by the representativeness of its main components. We believe that the components we chose to form our security benchmark are representative due to the following factors:

- The benchmark **metric** uses the notion of risk, which is strongly related to the benchmark security context and takes into account the vulnerability risk estimation of the Common Vulnerability Scoring System (Mell, Scarfone, and Romanosky 2007).
- The benchmark **workload** is formed by a web content management application used by millions of users (the Wordpress web application) and by a TPC-W component to emulate dozens of clients making requests

according to the requirements of TPC-W Benchmark specification.

- The **vulnerability injector** and the **attackload** target the most representative vulnerabilities for PHP web applications (Cross-site scripting and SQL Injection) and common attacks against web server interface (Denial of Service, Buffer Overflow and Code Injection).
- The **vulnerability repository** was built by replicating local instances of widely used on-line vulnerabilities repositories, such as the Open Source Vulnerability Database and the National Vulnerability Database.
- The **security repository** uses a popular web server-scanning tool (Nikto) that contains thousands of tests covering different web server brands and configuration.

6.3.2 Repeatability

The purpose of repeatability is to verify if our security benchmark provides similar results at different runs. To this end, we decided to repeat the first case study for three times, since it covers both benchmark parts. It is clear that more runs of the benchmark could be executed in order to get more data about benchmark results variation. However, in our opinion, and specially considering the dynamic part of the benchmark, three executions were enough to observe any significant change on results variation. Also, this was the number of times that the DBench-OLTP (Marco Vieira 2005) was validated for the repeatability attribute.

Table 6-12 presents the comparison of the benchmark metric for each one of the benchmark runs (Apache-based web servers, IBM7 and OR11, are represented by AP22). It is clear here that there was no change regarding the benchmark results of the static part. This is due to the fact that the analysis of known vulnerabilities is done from the same source and the benchmark tools simply repeat the same operation to collect vulnerability information. However, we can observe a variation in the results of the dynamic part. This is justified by the fact that the web servers responded differently for the attacks conducted in the benchmark runs. This behavior of the benchmark is expected since the attacks are done in a dynamic way. Even so, it is possible to observe that the benchmark ranking of the web servers remained the same across all benchmark runs. This led us to the conclusion that our security benchmark provides repeatable results.

6.3.3 Portability

To be portable, a security benchmark must fit for any component of software-based system. The case studies we have conducted show that our security

Table 6-12. Security Benchmark Results for Repeatability validation

Metrics	1st RUN				2nd RUN				3rd RUN			
Static Part	**IIS51**	**AP22**	**AP13**	**LT14**	**IIS51**	**AP22**	**AP13**	**LT14**	**IIS51**	**AP22**	**AP13**	**LT14**
CRL_{VEXA}	114.7	86.1	100.6	34.2	114.7	86.1	100.6	34.2	114.7	86.1	100.6	34.2
CRM_{VEXA}	182.6	50.8	123.8	49.5	182.6	50.8	123.8	49.5	182.6	50.8	123.8	49.5
CRH_{VEXA}	63.8	10.0	30.0	7.3	63.8	10.0	30.0	7.3	63.8	10.0	30.0	7.3
CRL_{NIKTO}	11.6	5.8	14.5	8.7	11.6	5.8	14.5	8.7	11.6	5.8	14.5	8.7
CRL_{SP}	126.3	91.9	115.1	42.9	126.3	91.9	115.1	42.9	126.3	91.9	115.1	42.9
CRM_{SP}	182.6	50.8	123.8	49.5	182.6	50.8	123.8	49.5	182.6	50.8	123.8	49.5
CRH_{SP}	63.8	10.0	30.0	7.3	63.8	10.0	30.0	7.3	63.8	10.0	30.0	7.3
Dynamic Part	**IIS51**	**AP22**	**AP13**	**LT14**	**IIS51**	**AP22**	**AP13**	**LT14**	**IIS51**	**AP22**	**AP13**	**LT14**
CRL_{DP}	6.77	0.0	0.0	1.24	5.88	0.0	0.0	1.28	5.62	0.0	0.0	1.41
CRM_{DP}	8.14	0.0	0.0	0	6.61	0.0	0.0	0	10.67	0.0	0.0	0
CRH_{DP}	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Security Risks	**IIS51**	**AP22**	**AP13**	**LT14**	**IIS51**	**AP22**	**AP13**	**LT14**	**IIS51**	**AP22**	**AP13**	**LT14**
SRL_{SP+DP}	**133.1**	**91.9**	**115.1**	**44.1**	**132.2**	**91.9**	**115.1**	**44.2**	**131.9**	**91.9**	**115.1**	**44.3**
SRM_{SP+DP}	**190.7**	**50.8**	**123.8**	**49.5**	**189.2**	**50.8**	**123.8**	**49.5**	**193.3**	**50.8**	**123.8**	**49.5**
SRH_{SP+DP}	**63.8**	**10**	**30**	**7.3**	**63.8**	**10**	**30**	**7.3**	**63.8**	**10**	**30**	**7.3**
SBench	**99**	**24.3**	**57.7**	**19.7**	**98.6**	**24.3**	**57.7**	**19.7**	**99.6**	**24.3**	**57.7**	**19.7**

benchmark may be applied in systems of different brands, versions, configuration and sizes. In the first case study, we addressed widely used web servers (Apache, IIS, Lighttpd) running over Windows XP and we used an analytical and experimental approach to obtain the benchmark metric.

In the second case study, we formed different set of web serving systems, covering different operating systems (Suse Linux and Windows XP), databases, (MySQL, Oracle), web applications and web servers. Although we focus only on the static part to obtain the benchmark metric, this demonstrated that our benchmark is portable to different components and systems.

6.3.4 Non-intrusiveness

To be non-intrusive, a security benchmark should require minimum changes in the system environment to be executed, and its components should not affect the expected output of the workload.

The static part of the benchmark requires no change in the system environment. To extract known vulnerabilities, what is needed is the list of component brands and versions that can be provided by the benchmark user. To run security tests, the web server scanning tools just requires the IP address and port of the

components under benchmark.

The dynamic part of the benchmark has a small degree of intrusiveness, since the Benchmark Management System is deployed in the same box that hosts the system under benchmark. However, these tools run common tasks (stopping starting services, monitors collectors) that are commonly executed by operating systems. Also, the injection of vulnerabilities and mounting of attacks are done in a preparation benchmark phase and there is no need to change any configuration in the system during the benchmark execution.

6.3.5 Feasibility

The feasibility of our security benchmark is given by the simplicity to build, run, and consolidate the results of our security benchmark. In other words, the feasibility is directly related to the cost of the benchmark. Considering that we adapted several existing tools, we were able to build the full set of scripts and tools that compose our security benchmark in 1 month (this corresponds to the allocation of one developer during the entire building process). If more resources were available, we would be able to reduce the development and test time, and consequently the cost. Even so, we believe that future proposals can reuse several parts of our benchmark, since key components were developed in Perl and Java technology that runs in a multi-platform environment.

Another important aspect we analyzed was the cost in terms of required disk space and execution time. Table 6-13 presents the execution time to the completion of each part of our security benchmark. Measurements presented in this table were collected directly from our experiments.

The total time to the execution of the static part in all evaluated web servers was 15 min, while the dynamic part took 24 hours to be completed. This difference resides in the complexity of injecting dynamic attacks and collecting all measurements in a real web serving system environment.

The disk space usage for all output generated in the static part was 4.5 MB. The dynamic phase generated 164 MB in output files per benchmark run (web server log files, workload outputs, resource consumption metrics, benchmark summarization files, etc.). Since this case study was repeated three times to confirm benchmark repeatability, the total disk usage consumption was 505.5 MB (4.5 x 3+164 x 3).

Finally, the effort needed to consolidate the benchmark results is very low, as few computer expertise is needed to put together benchmark CSV reports and use the equation of chapter 3 to define the proper weight of benchmark components and

Table 6-13. Benchmark Execution Time

Dur	Description
Static Security Benchmark	
15 min	Average time to the automated collection and analysis of 304 vulnerabilities (VEXA tool) from six different web servers on Case Study 1.
3.8 min	Average time to the execution of 6456 non-intrusive security tests (Nikto tool) and vulnerability risk estimation in each web server. The total time was 1368 seconds (~23 minutes).
Dynamic Security Benchmark	
3 min	Average time to the execution of each benchmark run. The first 30 seconds were devoted to the ramp up of the web server.
1,3 H	Average time to the execution of a whole benchmark campaign in each server (8 Hours in six servers). This includes the execution of 31 execution runs (6 inspection runs and 25 attacks runs). Each attack run exploited a different vulnerability in the faulty web application.

estimate the security risk based on the vulnerabilities risk listed in these reports.

6.4 CONCLUSION

This chapter presented two case studies benchmarking the security of real-world web serving system components. In the first case study, we benchmarked the security of widely used web servers (Microsoft Internet Information Service 5.1, Apache HTTP Server 1.3, Apache HTTP Server 2.2, and Lighttpd Web Server 1.4, IBM HTTP Server 11, Oracle HTTP Server 7) and used the full set of benchmark tools (covering both static and dynamic part) to obtain the benchmark metric. The results of this first case study showed that the combination of the static and dynamic analysis is a useful way to select the most secure of functionally equivalent systems. For example, Apache web servers resisted very well to attacks executed in the dynamic part, but did not obtain the highest security score as the evaluated versions contain many known vulnerabilities (static part). The most secure in our case study was Lighttpd 1.4 web server, which received the lowest risk in the benchmark metric. The total execution duration of the security benchmark in this case study was 24 hours (0.5 hour for the static part and 24 hours for the dynamic part).

In the second case study, different web serving systems components brands and versions were targeted (PHP, Java, and .Net Content Management Systems; Apache HTTP Server, Tomcat, and Microsoft IIS Web servers; MySQL, PostgreSQL, and Oracle Databases; Linux and Windows operating systems). However, we used a partial set of benchmark tools (more specifically, the VEXA

- Vulnerability Extractor and Analyzer of the static part) to benchmark the security of the targeted components due to the complexity of mounting attacks to exploit representative vulnerabilities of each component targeted in the second case study. VEXA extracted nearly 1500 software vulnerabilities reported in more than 10 years and provided statistical results that allowed us to reach very important observations, for example: the degree in which the configuration of the web server component affects the security risk of the whole web serving system. The results obtained in this case study represent a advantage to those interested in comparing the security of functionally equivalent systems and choosing the less risky based on known vulnerabilities.

We expect that the results presented in this chapter help users and system integrators to compare and choose among web serving system components, and also to determine how vulnerable and exposed these components are. Also, we expect that the benchmark analysis we provided serve to increase the acceptance of this work and helps the security community in the definition of a standard security benchmark.

7. CONCLUSION

This thesis presented a novel methodology to benchmark the security of software-based systems. The main goal of this research work was to propose and exemplify a methodology to benchmark the security of computer-based systems in a feasibly and useful way. The benchmark methodology should allow the comparison of functionally equivalent systems, helping users, developers and integrators in making use or purchasing decisions.

Our contribution is two-fold. First, we propose a methodology for security benchmarking in a generic way, not tied up to any particular class of systems. Then, following the methodology proposed, we present and demonstrate a security benchmark for the specific class of web-serving systems. While the first can be used by anyone to define a benchmark for any class of systems, the second serves as an example case-study to demonstrate the usefulness of the methodology, and also as a practical and ready to use benchmark for the important class of web serving systems.

The contributions of our work are organized in this thesis as follows:

State-of-the-art on the field of security benchmarking. In Chapter 2, we described concepts related with computer security, presented previous works about the characterization and representativeness of vulnerabilities and attacks, and covered the topics of security assessment, security metrics, and security benchmarking and, where relevant, contextualizing our work with other types of benchmark such as performance and dependability benchmarks. This survey not only provides a comprehensive background in benchmarking and security, but it also contributes to substantiate the options taken in the definition of security benchmarking methodology proposed in this work.

Generic methodology to benchmark the security of software-based systems.

In Chapter 3, we presented the specification of our security benchmark methodology for software-based systems, which is organized in two parts: one static and one dynamic. The static part was designed to measure the security risk posed by existing and already discovered vulnerabilities (known vulnerabilities). This is done by using two complementary approaches. First, known vulnerabilities are extracted from public vulnerabilities databases and analyzed. Second, a comprehensive set of security tests is executed to verify the presence of existing vulnerabilities. The result of the static part is an assessment of the risk caused by vulnerabilities previously know that may still be present in the system.

The dynamic part of our security benchmark methodology addresses the risk related to vulnerabilities not yet discovered (unknown vulnerabilities). To assess the risk related to vulnerabilities that are not known, a part of the system is seeded with vulnerabilities that are representative for that type of system, and then attacks are directed to the system. The part of the system that is seeded with vulnerabilities is not evaluated in itself: instead, the remaining parts of the system are observed to understand and measure how the exploitation of vulnerabilities in a component can affect the larger system. The vulnerabilities and the attacks used in this part of the methodology are defined based on field studies and information existing. The attacks are of several types to mimic real attacks.

One important clarification here is that the proposed approach is not intended to identify unknown vulnerabilities. Instead it is aimed at assessing the effects of these vulnerabilities to the whole system. To the best of our knowledge, this is a very original contribution to the security field, as our methodology is designed to cover any class of software-based system and has a complementary approach to cover vulnerabilities in an analytical and experimental way (the static and the dynamic part).

Security risk as the single benchmark metric. The benchmark metric of our methodology is security risk, which is based on the risk of individual vulnerabilities detected during the benchmark run. More specifically, security risk is estimated by the weighted sum of the security risk resulting from the benchmark executions, providing a numeric value that indicates the level of security of the benchmarked system. To the best of our knowledge, it was the first time that security risk was used in a security benchmark methodology.

Example of a concrete security benchmark for web serving systems. In Chapter 5, we described our security benchmark prototype for web serving systems. The goal was to provide to benchmark users with a clear example on how to implement benchmark components following the specifications that were defined in Chapter 3 and the procedures and rules defined in Chapter 4. For the

static part, we described our vulnerability analyzer and extractor (VEXA), which speeds up the collection of vulnerability information by performing queries over popular vulnerability databases. We also showed how a widely used security scanner tool (Nikto) was adapted to confirm the existence of known vulnerabilities over web servers. For the dynamic part, we described how to define the set of vulnerabilities, the set of attacks, and the workload. We injected vulnerabilities in a Wordpress web application to enable Cross-site scripting and SQL injection attacks and adapted the TPC-W benchmark and used it as our workload. We also described how the attackload was implemented to automatically exploit these vulnerabilities. This attackload also included attacks to exploit vulnerabilities present in the web server interface, including Buffer OverFlow and Code Injection in the HTTP protocol. The result we expect from exemplifying our implementation is to help users to develop new security benchmarks in a faster and more effective way.

Case studies to demonstrate the validity and applicability of our security benchmark methodology. In Chapter 6, we presented two case studies involving real and widely used web serving system components. These case studies enabled us to compare the security of popular software-based systems and demonstrated that the combination of the static and dynamic analysis is a useful way to select the most secure among functionally equivalent systems.

Future work prospects are centered on the improvement of our methodology components and in the extension of the benchmark implementation and case studies to other domains. Our future work plan is summarized next.

Develop a more effective vulnerability platform to support security benchmark executions. The vulnerability repository we presented in the static part of our methodology implementation used local instances of the Open Source Vulnerability Database and of the National Vulnerability Database. Although this was useful to demonstrate the effectiveness of our approach, it is necessary to reduce our dependency on such databases by proposing a platform to be maintained by benchmark users.

Use other security testing tools to identify known vulnerabilities. The web server security scanner used in our case study, although effective, is not the sole one available in the market. In this sense, it is necessary to investigate the coverage of competing security testing tools within the context of our benchmark methodology. By using different security testing tools, and creating a repository to cover a large range of security test, we expect to discover additional vulnerabilities that were not detected by a particular scanner.

Develop an automated tool to inject web application vulnerabilities integrated with the attackload and security checker. The vulnerabilities injected in the Wordpress web application were done manually. Next steps will include the automation of this injection procedure by taking advantage of the tool implementation proposed in (J. Fonseca, Vieira, and Madeira 2009). This is aimed at speeding up the preparation stage of the security benchmark run and significantly increasing the number of vulnerabilities injected in a given component. The challenge here is that this tool must target injection points that are reachable by attackers and that somehow interacts with the benchmark target.

Inject new vulnerability types. We injected Cross-site scripting, XSS vulnerabilities into the Wordpress web application as they are representative of the vulnerabilities that usually affect PHP web applications (J. Fonseca and Vieira 2008). However, in order to check the impact of vulnerability exploitation in different scenarios, it is necessary to move forward and target other types of vulnerabilities.

Select different components as the benchmark and vulnerable targets. In the case study we presented, web serving system was the system under benchmark, having the web server was benchmark target and the vulnerable component was the web application. Future work includes the selection of other web serving system components as the benchmark and vulnerable targets in order to provide to community more benchmark results.

Add more tests in the security checker components. The security verifications performed to assess the impact of attacks were sufficient to support the security measurements. However, we expect to include a more extensive set of tests, which will make the benchmark methodology stronger and more useful to users.

Implement our security benchmark methodology for other domains. More examples of security benchmarks are expected to be developed complying with the procedures and rules of our security benchmark methodology specification. This is an important step to demonstrate that our methodology can be applied for different classes of software-based systems, which will certainly contribute to increase the acceptance and adoption of our methodology by the security community.

Devise effective ways to use our security methodology as the starting point of a potential security benchmark agreement. A possible path to achieve this goal is to take advantage of an European research framework to propose the definition of a standard security benchmark specification, using our security benchmark methodology as a start point. This proposal is particularly relevant to reach a

benchmark agreement regarding the benchmark parameters and requirements that are necessary for each software-based system domain. This agreement is an essential part of any widely used benchmark, as can be verified in the benchmarks proposed in the performance and dependability fields.

8. REFERENCES

Acunetix. 2012. "Web Application Security - Acunetix Web Vulnerability Scanner." http://www.acunetix.com/vulnerability-scanner/.

Alberts, C. J., S. G. Behrens, R. D. Pethia, and W. R. Wilson. 1999. "Operationally Critical Threat, Asset, and Vulnerability Evaluation SM (OCTAVE SM) Framework, Version 1.0." CMU/SEI-99-TR-017, ADA 367718). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, June 1999.< http://www. sei. cmu. edu/publications/documents/99. reports/99tr017/99tr017abstract. html.

Alhazmi, O.H., and Y.K. Malaiya. 2006. "Prediction Capabilities of Vulnerability Discovery Models." In *Reliability and Maintainability Symposium, 2006. RAMS '06. Annual*, 86–91. doi:10.1109/RAMS.2006.1677355.

Antunes, N., and M. Vieira. 2010. "Benchmarking Vulnerability Detection Tools for Web Services." In *Web Services (ICWS), 2010 IEEE International Conference on*, 203–10. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5552783.

Apache HTTPD. 2015. "The Apache HTTP Server Project." http://httpd.apache.org/.

Arbaugh, W. A., W. L. Fithen, and J. McHugh. 2000. "Windows of Vulnerability: A Case Study Analysis." *Computer* 33 (12): 52–59.

Ardi, Shanai, David Byers, and Nahid Shahmehri. 2006. "Towards a Structured Unified Process for Software Security." In *Proceedings of the 2006 International Workshop on Software Engineering for Secure Systems*, 3–10. Shanghai, China: ACM. doi:10.1145/1137627.1137630.

Armbrust, Michael, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, et al. 2010. "A View of Cloud Computing." *Communications of the ACM* 53 (4): 50–58.

Arthur, Charles. 2012. "Cyber-Attack Concerns Raised over Boeing 787 Chip's

'Back Door' | Guardian.co.uk." http://www.guardian.co.uk/technology/2012/may/29/cyber-attack-concerns-boeing-chip.

AS/NZS 4360. 1999. "AS/NZS 4360:1999 - Australian Standard / New Zealand Risk Management Standard."

Attrition, J. 2008. "Vulnerability Counts and OSVDB Advocacy." April. http://blog.osvdb.org/2008/04/03/vulnerability-counts-and-osvdb-advocacy/.

Avizienis, A., J. C. Laprie, B. Randell, and C. Landwehr. 2004. "Basic Concepts and Taxonomy of Dependable and Secure Computing." *IEEE Transactions on Dependable and Secure Computing*, 11–33.

Avizienis, A., J. C Laprie, B. Randell, and University of Newcastle upon Tyne. Computing Science. 2001. "Fundamental Concepts of Dependability." *TECHNICAL REPORT SERIES-UNIVERSITY OF NEWCASTLE UPON TYNE COMPUTING SCIENCE*.

AWS Amazon. 2015. "Amazon Web Services (AWS) - Cloud Computing Services." https://aws.amazon.com/.

Barnum, S., and A. Sethi. 2007. "Attack Patterns as a Knowledge Resource for Building Secure Software." In *OMG Software Assurance Workshop: Cigital*. http://www.orkspace.net/secdocs/Conferences/BlackHat/Federal/2007/Attack%20Patterns%20-%20Knowing%20Your%20Enemies%20in%20Order%20to%20Defeat%20Them-paper.pdf.

Behnia, Armaghan, Rafhana Abd Rashid, and Junaid Ahsenali Chaudhry. 2012. "A Survey of Information Security Risk Analysis Methods." *Smart Computing Review* 2 (1): 79–94.

Berry, Mike, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, et al. 1989. "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers." *International Journal of High Performance Computing Applications* 3 (3): 5–40.

Bishop, M. 1999. "Vulnerabilities Analysis." In *Proceedings of the Recent Advances in Intrusion Detection*, 125–36. ftp://birr.mirrorservice.org/pub/security/development/secure-programming/bishop-1999-vulnerabilities-analysis.pdf.

———. 2003. *Computer Security: Art and Science*. Addison-Wesley.

———. 2004. *Introduction to Computer Security*. Addison-Wesley Professional.

Bodoni, Stephanie. 2013. "Sony Fined $394,500 Over Hacker Attack on PlayStation Data - Bloomberg." *Bloomberg*, January. http://www.bloomberg.com/news/2013-01-24/sony-fined-394-000-over-

2011-hacker-attack-on-playstation-data.html.

Bondavalli, A., P. Lollini, R. Barbosa, A. Ceccarelli, L. Falai, J. Karlsson, I. Kocsis, et al. 2009. “D3.2: Final Research Roadmap, Formal Deliverable AMBER Project – Assessing, Measuring and Benchmarking Resilience, IST – 216295 AMBER, EU FP7 Program.” European Commission. http://www.amber-project.eu.

Brocklehurst, S., B. Littlewood, T. Olovsson, and E. Jonsson. 1994. “On Measurement of Operational Security.” *Aerospace and Electronic Systems Magazine, IEEE* 9 (10): 7–16.

Browne, Peter S. 1976. “Computer Security: A Survey.” In *Proceedings of the June 7-10, 1976, National Computer Conference and Exposition*, 53–63. AFIPS ’76. New York, NY, USA: ACM. doi:10.1145/1499799.1499809.

Buecker, Axel, Danny Allan, Tim Hahn, Andras Szakal, and Jim Whitmore. 2010. “Security in Development: The IBM Secure Engineering Framework.” IBM Redbooks. http://www.redbooks.ibm.com/abstracts/redp4641.html.

Cain, H.W., R. Rajwar, M. Marden, and M.H. Lipasti. 2001. “An Architectural Evaluation of Java TPC-W.” In *The Seventh International Symposium on High-Performance Computer Architecture, 2001. HPCA*, 229–40. doi:10.1109/HPCA.2001.903266.

CAPEC. 2014. “Common Attack Pattern Enumeration and Classification.” https://capec.mitre.org.

CAPEC-18. 2014. “CAPEC-18: Embedding Scripts in Non-Script Elements (Version 2.6).” https://capec.mitre.org/data/definitions/18.html.

CAPEC-100. 2014. “CAPEC-100: Overflow Buffers (Version 2.6).” https://capec.mitre.org/data/definitions/100.html.

CAPEC-152. 2014. “CAPEC-152: Injection (Version 2.6).” https://capec.mitre.org/data/definitions/152.html.

Carreira, J., H. Madeira, and J. G Silva. 1998. “Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers.” *IEEE Transactions on Software Engineering* 24 (2): 125–36.

Cavusoglu, H., and S. Raghunathan. 2007. “Efficiency of Vulnerability Disclosure Mechanisms to Disseminate Vulnerability Knowledge.” *Software Engineering, IEEE Transactions on* 33 (3): 171–85. doi:10.1109/TSE.2007.26.

CC Protection Profiles. 2012. “Protection Profiles : The Common Criteria Portal.” http://www.commoncriteriaportal.org/pps/.

Chen, Peter Pin-Shan. 1976. “The Entity-Relationship Model—toward a Unified View of Data.” *ACM Transactions on Database Systems (TODS)* 1 (1):

9–36.

Chillarege, R., I.S. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, and M.-Y. Wong. 1992. "Orthogonal Defect Classification-a Concept for in-Process Measurements." *Software Engineering, IEEE Transactions on* 18 (11): 943–56.

Christey, S. 2005. "PLOVER: Preliminary List of Vulnerability Examples for Researchers." In *NIST Workshop Defining the State of the Art of Software Security Tools*.

Christey, S., and R. A Martin. 2007. "Vulnerability Type Distributions in CVE." *Mitre Report, May*.

CIS. 2008. "Center for Internet Security Benchmark for Apache Web Server v2.1." http://benchmarks.cisecurity.org/tools2/apache/CIS_Apache_Benchmark_v2.1.pdf.

———. 2012. "Center for Internet Security." CIS. *The Center for Internet Security*. http://www.cisecurity.org/.

Clark, D. D, W. E Boebert, S. Gerhart, J. V Guttag, R. A Kemmerer, S. T Kent, S. M.M Lambert, et al. 1991. *Computers at Risk: Safe Computing in the Information Age*. National Research Council, National Academy Press.

Clark, J. A., and D. K. Pradhan. 1995. "Fault Injection: A Method for Validating Computer-System Dependability." *Computer* 28 (6): 47–56.

CNCI. 2008. "The Comprehensive National Cybersecurity Initiative | The White House." http://www.whitehouse.gov/cybersecurity/comprehensive-national-cybersecurity-initiative.

Common Criteria (CC). 2009. "Common Criteria for Information Technology Security Evaluation." CCMB-2009-07-001(003). http://www.commoncriteriaportal.org/cc/.

Common Methodology (CEM). 2009. "Common Methodology for Information Technology Security Evaluation." CCMB-2009-07-004. http://www.commoncriteriaportal.org/cc/.

Coverity. 2009. "Coverity Scan Open Source Report 2009."

———. 2011. "Coverity Scan - 2011 Open Source Integrity Report." http://www.coverity.com/.

CSI. 2012. "Computer Security Institute." http://gocsi.com/.

Curphey, M., and R. Arawo. 2006. "Web Application Security Assessment Tools." *Security & Privacy, IEEE* 4 (4): 32–41.

CVE. 2014. "CVE - Common Vulnerabilities and Exposures (CVE)." http://cve.mitre.org/.

CVE-2013-2205. 2013. "XSS Vulnerability in Wordpress before 3.5.2." http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-2205.

CWE. 2012. "CWE - Common Weakness Enumeration." http://cwe.mitre.org/.

CWE-79. 2014. "CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-Site Scripting') (2.8)." http://cwe.mitre.org/data/definitions/79.html.

CWSS. 2011. "CWE - Common Weakness Scoring System (CWSS)." http://cwe.mitre.org/cwss/index.html.

Cybenko, George, Lyle Kipp, Lynn Pointer, and David Kuck. 1990. *Supercomputer Performance Evaluation and the Perfect Benchmarks*. Vol. 18. 3b. ACM. http://dl.acm.org/citation.cfm?id=255163.

Das, R., S. Sarkani, and T.A Mazzuchi. 2012. "Software Selection Based on Quantitative Security Risk Assessment." *IJCA Special Issue on Computational Intelligence & Information Security (CIIS)*, November, 45–46.

DBench. 2004. "Dependability Benchmarking Project." IST-2000-25425. http://www2.laas.fr/DBench/.

Dowd, M., J. McDonald, and J. Schuh. 2006. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional.

Duraes, J. A., and H. S. Madeira. 2006. "Emulation of Software Faults: A Field Data Study and a Practical Approach." *Software Engineering, IEEE Transactions on* 32 (11): 849–67.

Duraes, J., and H. Madeira. 2004. "Generic Faultloads Based on Software Faults for Dependability Benchmarking." In *2004 International Conference on Dependable Systems and Networks*, 285–94. doi:10.1109/DSN.2004.1311898.

Durães, J., M. Vieira, and H. Madeira. 2004. "Dependability Benchmarking of Web-Servers." *Proc. 23rd International Conference, SAFECOMP*.

Du, W., and A. P. Mathur. 1998. "Categorization of Software Errors That Led to Security Breaches." In *Proceedings of the 21st National Information Systems Security Conference (NISSC'98)*.

Easttom, C., and J. Taylor. 2010. *Computer Crime Investigation and the Law*. Course Technology PTR.

Eckert, John. 1964. ENIAC - Electronic Numerical Integrator and Computer. 3,120,606, issued 1964. http://s7.computerhistory.org/is/image/CHM/500004299-05-01?$re-medium$.

Fabre, J. C, F. Salles, M. R Moreno, and J. Arlat. 1999. "Assessment of COTS

Microkernels by Fault Injection.” In *Dependable Computing for Critical Applications 7, 1999*, 25–44.

Fielding, R., J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. 1999. *Hypertext Transfer protocol–HTTP/1.1*. RFC 2616, June.

Fonseca, Jose. 2011. “Evaluating the [In]security of Web Applications.” PhD Thesis, University of Coimbra.

Fonseca, J., and M. Vieira. 2008. “Mapping Software Faults with Web Security Vulnerabilities.” In *IEEE International Conference on Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008*, 257–66.

Fonseca, J., M. Vieira, and H. Madeira. 2009. “Vulnerability & Attack Injection for Web Applications.” In *Dependable Systems & Networks, 2009. DSN’09. IEEE/IFIP International Conference on*, 93–102.

Futcher, L., and R. von Solms. 2008. “Guidelines for Secure Software Development.” In *Proceedings of the 2008 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries: Riding the Wave of Technology*, 56–65. http://dl.acm.org/citation.cfm?id=1456667.

Gates, Bill. 2002. “Trustworthy Computing.” January. http://news.cnet.com/2009-1001-817210.html.

Goertzel, K. M., T. Winograd, H. L. McKinley, P. Holley, and B. A. Hamilton. 2006. “Security in the Software Lifecycle.” *Department of Homeland Security, Version* 1.

Goldstein, M., N. Perlroth, and M. Corkery. 2014. “Neglected Server Provided Entry for JPMorgan Hackers.” *New York Times*, December. http://dealbook.nytimes.com/2014/12/22/entry-point-of-jpmorgan-data-breach-is-identified/?_r=0.

Gray, J. 1990. “A Census of Tandem System Availability between 1985 and 1990.” *Reliability, IEEE Transactions on* 39 (4): 409–18.

Group, MM. 2011. “World Internet Usage Statistics News and World Population Stats.” March. http://www.internetworldstats.com/stats.htm.

Hansman, S., and R. Hunt. 2005. “A Taxonomy of Network and Computer Attacks.” *Computers & Security* 24 (1): 31–43.

Han, S., K. G. Shin, and H. A. Rosenberg. 1995. “Doctor: An Integrated Software Fault Injection Environment for Distributed Real-Time Systems.” In *Computer Performance and Dependability Symposium, 1995. Proceedings., International*, 204–13. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=395831.

Hassan, A. E, and R. C Holt. 2002. “Architecture Recovery of Web

Applications.” In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, 349–59.

Hatton, L. 2007. “The Chimera of Software Quality.” *Computer* 40 (8): 104–103.

Hoglund, G., and G. McGraw. 2004. *Exploiting Software: How to Break Code*. Pearson Education India.

Houmb, Siv Hilde, Virginia NL Franqueira, and Erlend A. Engum. 2010. “Quantifying Security Risk Level from CVSS Estimates of Frequency and Impact.” *Journal of Systems and Software* 83 (9): 1622–34.

Howard, J. D, and T. A Longstaff. 1998. “A Common Language for Computer Security Incidents.” *Sandia Report: SAND98-8667, Sandia National Laboratories, Http://www. Cert. org/research/taxonomy_988667. Pdf*.

Howard, J. D, and P. Meunier. 2002. “Using a ‘common Language’ for Computer Security Incident Information.” *FLY*, 301.

Howard, M., and D. LeBlanc. 2002. *The STRIDE Threat Model. From the Book’Writing Secure Code’*. Microsoft Press.

Howard, M., D. LeBlanc, Safari Tech Books Online, and Safari Books Online (Firme). 2003. *Writing Secure Code*. Vol. 2. Microsoft press Redmond, WA.

Howard, M., D. LeBlanc, and J. Viega. 2005. *19 Deadly Sins of Software Security*. McGraw-Hill/Osborne New York.

HP Fortify. 2012. “HP Fortify | Comprehensive Software Security Assurance Solutions, Products, and Services.” https://www.fortify.com/.

HP WebInspect. 2012. “HP Fortify | HP WebInspect.” https://www.fortify.com/products/web_inspect.html.

Hsueh, Mei-Chen, Timothy K. Tsai, and Ravishankar K. Iyer. 1997. “Fault Injection Techniques and Tools.” *Computer* 30 (4): 75–82.

IBM. 2014. “IBM Company.” http://www.ibm.com/us/en/.

IBM Appscan. 2012. “IBM Software - IBM Security AppScan Family.” http://www-01.ibm.com/software/awdtools/appscan/.

IBM X-Force. 2012. “IBM X-Force Threat Reports - United States.” March. http://www-935.ibm.com/services/us/iss/xforce/trendreports/.

ISO 17799, ISO. 2005. “IEC 17799: 2005.” *Information technology–Code of Practice for Information Security Management*.

Jaquith, A. 2007. *Security Metrics: Replacing Fear, Uncertainty, and Doubt*. Addison-Wesley Professional.

Java TPC-W. 2013. “Java TPC-W Implementation Distribution - University of Wisconsin - Madison.” http://pharm.ece.wisc.edu/tpcw.shtml.

Kalakech, A., K. Kanoun, Y. Crouzet, and J. Arlat. 2004. "Benchmarking the Dependability of Windows NT4, 2000 and XP." In *Dependable Systems and Networks, 2004 International Conference on*, 681–86.

Kalyanakrishnam, M., Z. Kalbarczyk, and R. Iyer. 1999. "Failure Data Analysis of a LAN of Windows NT Based Computers." In *Reliable Distributed Systems, 1999. Proceedings of the 18th IEEE Symposium on*, 178–87. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=805094.

Kanawati, G. A., N. A. Kanawati, and J. A. Abraham. 1992. "FERRARI: A Tool for the Validation of System Dependability Properties." In *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, 336–44. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=243567.

Karabacak, Bilge, and Ibrahim Sogukpinar. 2005. "ISRAM: Information Security Risk Analysis Method." *Computers & Security* 24 (2): 147–59.

Karger, Paul A., and Roger R. Schell. 1974. "Multics Security Evaluation Volume II. Vulnerability Analysis." http://stinet.dtic.mil/oai/oai?&verb=getRecord&metadataPrefix=html&identifier=ADA001120.

Kaufman, L.M. 2009. "Data Security in the World of Cloud Computing." *IEEE Security Privacy* 7 (4): 61–64. doi:10.1109/MSP.2009.87.

Killourhy, K. S., R. A. Maxion, and K. M. C. Tan. 2004. "A Defense-Centric Taxonomy Based on Attack Manifestations." *Dependable Systems and Networks, 2004 International Conference on*, 102–11.

Kirkpatrick, R. J., J. A. Walker, and R. Firth. 1992. "Software Development Risk Management: An SEI Appraisal." *SEI Technical Review*.

Koopman, P., and H. Madeira. 1999. "Dependability Benchmarking & Prediction: A Grand Challenge Technology Problem." In *1st IEEE Int. Workshop on Real-Time Mission-Critical Systems: Grand Challenge Problems*.

Koopman, P., J. Sung, C. Dingman, D. Siewiorek, and T. Marz. 1997. "Comparing Operating Systems Using Robustness Benchmarks." In *Reliable Distributed Systems, 1997. Proceedings., The Sixteenth Symposium on*, 72–79.

Kothari, C. R. 2008. *Research Methodology: Methods and Techniques*. New Age International.

Krsul, I. V. 1998. "Software Vulnerability Analysis." PhD Thesis, Purdue University.

Kumar, S. 1995. "Classification and Detection of Computer Intrusions." PhD Thesis, Purdue University.

Landwehr, C. E., A. R. Bull, J. P. McDermott, and W. S. Choi. 1994. "A

Taxonomy of Computer Program Security Flaws." *ACM Computing Surveys (CSUR)* 26 (3): 211–54.

Laranjeiro, N., S. Canelas, and M. Vieira. 2008. "Wsrbench: An on-Line Tool for Robustness Benchmarking." In *Services Computing, 2008. SCC'08. IEEE International Conference on*, 2:187–94.

Laranjeiro, N., M. Vieira, and H. Madeira. 2008. "Experimental Robustness Evaluation of JMS Middleware." In *Services Computing, 2008. SCC'08. IEEE International Conference on*, 1:119–26.

Lee, Dave. 2013. "Global Internet Slows after 'Biggest Attack in History.'" *BBC News*, March. http://www.bbc.co.uk/news/technology-21954636.

Lee, I., and R. K. Iyer. 1995. "Software Dependability in the Tandem GUARDIAN System." *Software Engineering, IEEE Transactions on* 21 (5): 455–67.

Lindqvist, U., and E. Jonsson. 1997. "How to Systematically Classify Computer Security Intrusions." In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, 154–63.

Liu, S., and B. Cheng. 2009. "Cyberattacks: Why, What, Who, and How." *IT Professional* 11 (3): 14–21.

Logman. 2013. "Microsoft Windows XP - Logman." http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/nt_command_logman.mspx?mfr=true.

Lowrance, W. W. 1976. *Of Acceptable Risk: Science and the Determination of Safety. Los Altos, CA: William Kaufmann*. Inc.

Madeira, H., K. Kanoun, J. Arlat, D. Costa, Y. Crouzet, M. D. Cin, P. Gil, and N. Suri. 2002. "Towards a Framework for Dependability Benchmarking." *Proc. of the 4th European Dependable Computing Conference (EDCC'02)*.

Madeira, H., and P. Koopman. 2001. "Dependability Benchmarking: Making Choices in an N-Dimensional Problem Space." In . http://repository.cmu.edu/isr/662/.

Martin, B., M. Brown, A. Paller, and D. Kirby. 2010. "CWE/SANS Top 25 Most Dangerous Software Errors." *MITRE, SANS*. cwe.mitre.org/top25/.

Martin, R. A. 2008. "Making Security Measurable and Manageable." In *Military Communications Conference, 2008. MILCOM 2008. IEEE*, 1–9. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4753203.

Martin, R. A, and S. Barnum. 2008. "Common Weakness Enumeration (CWE) Status Update." *ACM SIGAda Ada Letters* 28 (1): 88–91.

Martin, R. A, S. M Christey, and J. Jarzombek. 2005. "The Case for Common Flaw Enumeration." In *NIST Workshop on Software Security Assurance*

Tools, Techniques, and Metrics.

Martins, E., C. M. F. Rubira, and N. G. M. Leme. 2002. "Jaca: A Reflective Fault Injection Tool Based on Patterns." In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, 483–87. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1028934.

McCartney, S. 1999. *ENIAC: The Triumphs and Tragedies of the World's First Computer*. Walker & Company.

McGraw, G. 2004. "Software Security." *Security & Privacy, IEEE* 2 (2): 80–83. doi:10.1109/MSECP.2004.1281254.

———. 2008. "Automated Code Review Tools for Security." *Computer* 41 (12): 108–11. doi:10.1109/MC.2008.514.

Meier, J. D., Alex Mackman, Michael Dunner, Srinath Vasireddy, Ray Escamilla, and Anandha Murukan. 2003. *Improving Web Application Security: Threats and Countermeasures*. Microsoft Redmond, WA.

Mell, P., K. Scarfone, and S. Romanosky. 2007. "A Complete Guide to the Common Vulnerability Scoring System Version 2.0." *Published by FIRST-Forum of Incident Response and Security Teams*.

Mendes, N. 2015. "PhD Thesis Supporting Material." *PhD Thesis Supporting Material*. https://eden.dei.uc.pt/~naaliel/thesis/.

Mendes, N., J. Duraes, and H. Madeira. 2011. "Benchmarking the Security of Web Serving Systems Based on Known Vulnerabilities." In *2011 5th Latin-American Symposium on Dependable Computing (LADC)*, 55–64. doi:10.1109/LADC.2011.14.

Mendes, N., A. A. Neto, J. Durães, M. Vieira, and H. Madeira. 2008. "Assessing and Comparing Security of Web Servers." In *Proceedings of the 2008 14th IEEE Pacific Rim International Symposium on Dependable Computing-Volume 00*, 313–22. IEEE Computer Society Washington, DC, USA.

Metasploit. 2015. "Metasploit Penetration Testing Software | Metasploit Framework | Metasploit Project." http://www.metasploit.com/.

MetricsCenter. 2012. "Metrics Center - Quantitative Analysis for Better Decisions." http://www.metricscenter.org/.

Meunier, P. 2008. "Classes of Vulnerabilities and Attacks." *Wiley Handbook of Science and Technology for Homeland Security*.

Microsoft IIS. 2015. "Overview : The Official Microsoft IIS Site." http://www.iis.net/overview.

Microsoft SecBulletin. 2012. "Security Bulletin Severity Rating System." http://technet.microsoft.com/en-us/security/gg309177.aspx.

MilW0rm. 2010. “milw0rm - Exploits : Vulnerabilities : Videos : Papers : Shellcode.” http://web.archive.org/web/20100528020113/http://milw0rm.com/.

MITRE Corp. 2012. “MITRE--Applying Systems Engineering and Advanced Technology to Critical National Problems.” http://www.mitre.org/.

MySQL. 2012. “MySQL :: The World’s Most Popular Open Source Database.” http://www.mysql.com/.

NCP. 2012. “National Vulnerability Database (NVD) National Checklist Program Repository.” http://web.nvd.nist.gov/view/ncp/repository.

NCSD. 2012. “National Cyber Security Division.” http://www.dhs.gov/xabout/structure/editorial_0839.shtm.

Netcraft. 2015. “Netcraft - September 2015 Web Server Survey.” http://news.netcraft.com/archives/category/web-server-survey/.

Neto, A.A. 2012. “Security Benchmarking of Transactional Systems.” PhD Thesis, University of Coimbra.

Neto, A. A., and M. Vieira. 2008. “Towards Assessing the Security of DBMS Configurations.” In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, 90–95. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4630074.

———. 2010. “Benchmarking Untrustworthiness: An Alternative to Security Measurement.” *International Journal of Dependable and Trustworthy Information Systems (IJDTIS)* 1 (2): 32–54.

neuroFuzz. 2012. “neuroFuzz Application Security.” http://www.neurofuzzsecurity.com/.

Neves, Nuno, Joao Antunes, Miguel Correia, Paulo Verissimo, and Rui Neves. 2006. “Using Attack Injection to Discover New Vulnerabilities.” In *Proceedings of the International Conference on Dependable Systems and Networks*, 457–66. IEEE Computer Society.

Nibaldi, G. H. 1979. “Proposed Technical Evaluation Criteria for Trusted Computer Systems.” DTIC Document.

Nikto2. 2015. “Nikto2 | CIRT.net.” http://cirt.net/nikto2/.

NIST. 1998. “History of Computer Security.” http://csrc.nist.gov/publications/history/.

NIST-SP800-12, NIST. 1995. *800-12: An Introduction to Computer Security–The NIST Handbook*. October.

NVD. 2014. “National Vulnerability Database Home.” http://nvd.nist.gov/.

NVD-CWE. 2013. “National Vulnerability Database & CWE - Common Weakness Enumeration.” http://nvd.nist.gov/cwe.cfm#cwes.

Omar H. Alhazmi, and Yashwant K. Malaiya. 2006. “Measuring and Enhancing Prediction Capabilities of Vulnerability Discovery Models for Apache and IIS HTTP Servers.” In *Software Reliability Engineering, 2006. ISSRE ’06. 17th International Symposium on*, 343–52. doi:10.1109/ISSRE.2006.26.

OSVDB. 2014. “OSVDB: The Open Source Vulnerability Database.” http://www.osvdb.org/.

Ounce Labs. 2012. “IBM Acquires Ounce Labs.” http://www-01.ibm.com/software/rational/welcome/ouncelabs/.

OWASP. 2012. “The Open Web Application Security Project.” https://www.owasp.org/index.php/Main_Page.

———. 2013. “2013 OWASP Top 10 Application Security Risks.” https://www.owasp.org/index.php/Top_10_2013-Top_10.

OWASP-CLASP. 2012. “CLASP Project - Comprehensive, Lightweight Application Security Process.” https://www.owasp.org/index.php/Category:OWASP_CLASP_Project.

Ozment, Andy. 2007. “Improving Vulnerability Discovery Models.” In *Proceedings of the 2007 ACM Workshop on Quality of Protection*, 6–11. Alexandria, Virginia, USA: ACM.

Ozment, A., and S. E. Schechter. 2006. “Milk or Wine: Does Software Security Improve with Age.” In *15th Usenix Security Symposium*. http://www.usenix.org/event/sec06/tech/full_papers/ozment/ozment_html/.

Perl. 2013. “The Perl Programming Language - Www.perl.org.” http://www.perl.org/.

Pettit, S. 2001. “Anatomy of a Web Application: Security Considerations.” *Sanctum Inc. July*.

Piessens, F. 2002. “A Taxonomy of Causes of Software Vulnerabilities in Internet Software.” *Supplementary Proceedings of the 13th International Symposium on Software Reliability Engineering*, 47–52.

Pixy. 2012. “Pixy: XSS and SQLI Scanner for PHP.” http://pixybox.seclab.tuwien.ac.at/pixy/.

Ponemon. 2012. “2012 Cost of Cyber Crime Study: United States - News and Press Releases.” Ponemon Institute. http://www.ponemon.org/news-2/44.

———. 2015. “2015 Cost of Data Breach Study: Global Analysis.” http://www-03.ibm.com/security/data-breach/.

Pothamsetty, V., and B. Akyol. 2004. “A Vulnerability Taxonomy for Network Protocols: Corresponding Engineering Best Practice Countermeasures.” *Communications, Internet, and Information Technology*.

Pothamsetty, Venkat. 2005. “Where Security Education Is Lacking.” In *Proceedings of the 2nd Annual Conference on Information Security Curriculum Development*, 54–58. InfoSecCD ’05. New York, NY, USA: ACM. doi:10.1145/1107622.1107635.

Powell, D., R. Stroud, and others. 2003. “Conceptual Model and Architecture of MAFTIA.” *University of NewCastle Upon Tyne Computing Science* Technical Report.

Powershell. 2013. “Windows PowerShell.” http://msdn.microsoft.com/en-us/library/windows/desktop/dd835506(v=vs.85).aspx.

PsExec. 2013. “PsExec Remote Control Program.” http://technet.microsoft.com/en-us/sysinternals/bb897553.aspx.

Pye, G., and M. J. Warren. 2007. “A Model and Framework for Online Security Benchmarking.” *INFORMATICA-LJUBLJANA-* 31 (2): 209.

Radatz, J., A. Geraci, and F. Katki. 1990. “IEEE Standard Glossary of Software Engineering Terminology.” *IEEE Standards Board, New York, Standard IEEE Std*, 610–12.

Ranum, M. 1997. *A Taxonomy of Internet Attacks, Web Security Sourcebook*. John Wiley & Sons.

Rowe, W. D. 1977. “An Anatomy of Risk John Wiley and Sons.” *New York, NY*.

Saitta, P., B. Larcom, and M. Eddington. 2005. *Trike v1 Methodology Document*. Retrieved 2007-11-01, from http://www.octotrike.org/papers/Trike_v1_Methodology_Document-draft.pdf.

Sanger, David. 2012. “Obama Ordered Wave of Cyberattacks Against Iran - NYTimes.com.” http://www.nytimes.com/2012/06/01/world/middleeast/obama-ordered-wave-of-cyberattacks-against-iran.html?_r=1.

SANS Institute. 2012. “SANS Information, Network, Computer Security Training, Research, Resources.” http://www.sans.org/.

SANS @Risk. 2012. “SANS: @Risk: The Consensus Security Alert.” http://www.sans.org/newsletters/risk/.

SANS Trends. 2009. “SANS: Top Cyber Security Risks - Vulnerability Exploitation Trends.” http://www.sans.org/top-cyber-security-risks/trends.php.

Schechter, S. E. 2002. “Quantitatively Differentiating System Security.” In *The First Workshop on Economics and Information Security*, 16–17. California University. http://www.cl.cam.ac.uk/~rja14/econws/31.pdf.

———. 2004. “Computer Security Strength & Risk: A Quantitative Approach.” PhD Thesis, Havard University.

http://research.microsoft.com/pubs/192264/thesis.pdf.

Schiller, W. L. 1975. "The Design and Specification of a Security Kernel for the PDP-11/45,." http://stinet.dtic.mil/oai/oai?&verb=getRecord&metadataPrefix=html&identifier=ADA011712.

Schneider, F. B. 1999. *Trust in Cyberspace*. National Academies Press.

Schneier, B. 2009. *Schneier on Security*. Wiley.

SecToolMarket. 2012. "SecTool Market - Security Test of Web Application Scanners." http://www.sectoolmarket.com/.

SecTools. 2014. "SecTools.Org Top Network Security Tools." http://sectools.org/.

Seixas, N., J. Fonseca, M. Vieira, and H. Madeira. 2009. "Looking at Web Security Vulnerabilities from the Programming Language Perspective: A Field Study." In *20th International Symposium on Software Reliability Engineering, 2009. ISSRE '09*, 129–35. doi:10.1109/ISSRE.2009.30.

Shafranovich, Y. 2005. "Common Format and MIME Type for Comma-Separated Values (CSV) Files."

Shapiro, J. S. 2003. "Understanding the Windows EAL4 Evaluation." *Computer*, 103–5.

Siewiorek, D. P, J. J Hudak, B. H Suh, and Z. Segal. 1993. "Development of a Benchmark to Measure System Robustness." In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, 88–97.

SOE. 2011. "Sony Online Entertainment Announces Theft of Data from Its Systems." May. https://www.soe.com/securityupdate/pressrelease.vm.

Spainhower, L., and K. Kanoun. 2007. "White Book on Dependability Benchmarking." *IEEE Computer Society Eds*.

SPEC. 1988. "Standard Performance Evaluation Corporation." http://www.spec.org/.

Stallings, W. 1999. *Network Security Essentials: Applications and Standards*. Prentice Hall PTR Upper Saddle River, NJ, USA.

Stoneburner, Gary, Alice Y. Goguen, and Alexis Feringa. 2002. "Sp 800-30. Risk Management Guide for Information Technology Systems." http://dl.acm.org/citation.cfm?id=2206240.

Stoneburner, Gary, Clark Hayden, and Alexis Feringa. 2004. "SP 800-27 Rev. A. Engineering Principles for Information Technology Security (A Baseline for Achieving Security), Revision A." Gaithersburg, MD, United States: National Institute of Standards & Technology.

Sullivan, M., and R. Chillarege. 1992. "A Comparison of Software Defects in Database Management Systems and Operating Systems." In *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, 475–84. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=243586.

Symantec. 2014. "Symantec Internet Security Threat Report 2014." http://www.symantec.com/.

———. 2015. "2015 Internet Security Threat Report." Symantec. https://www4.symantec.com.

Thompson, H. H., and S. G. Chase. 2007. *The Software Vulnerability Guide*. Firewall Media.

Titterington, Graham. 2010. *Trends in Security Attacks and Incidents*. Ovum Summit.

TPC. 1988. "Transaction Processing Performance Council." August 10. http://www.tpc.org/.

Tsai, T. K., R. K. Iyer, and D. Jewitt. 1996. "An Approach towards Benchmarking of Fault-Tolerant Commercial Systems." In *Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on*, 314–23. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=534616.

Tsipenyuk, K., B. Chess, and G. McGraw. 2005. "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors." *Security & Privacy, IEEE* 3 (6): 81–84. doi:10.1109/MSP.2005.159.

U.S. Census Bureau. 2009. "Computer and Internet Use - Publications - U.S. Census Bureau." October. http://www.census.gov/hhes/computer/publications/index.html.

US-CERT. 2006. "The Continuing Denial of Service Threat Posed by DNS Recursion."

———. 2014. "US-CERT - United States Computer Emergency Readiness Team." http://www.us-cert.gov/.

Van Der Steen, Aad J. 1991. "The Benchmark of the EuroBen Group." *Parallel Computing* 17 (10): 1211–21.

Vass, J., J. Harwell, H. Bharadvaj, and A. Joshi. 1998. "The World Wide Web." *Potentials, IEEE* 17 (4): 33–37.

Véras, P., E. Villani, A. Ambrósio, R. Pontes, M. Vieira, and H. Madeira. 2010. "Benchmarking Software Requirements Documentation for Space Application." *Computer Safety, Reliability, and Security*, 112–25.

Verizon. 2012. "Verizon Enterprise Solutions Worldwide Site." http://www.verizonbusiness.com/.

Vieira, M., N. Antunes, and H. Madeira. 2009. "Using Web Security Scanners to Detect Vulnerabilities in Web Services." In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, 566–71. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5270294.

Vieira, Marco. 2005. "Dependability Benchmarking for Transactional Systems." University of Coimbra.

Vieira, M., N. Laranjeiro, and H. Madeira. 2007. "Benchmarking the Robustness of Web Services." In *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*, 322–29. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4459677.

Vieira, M., and H. Madeira. 2003a. "A Dependability Benchmark for OLTP Application Environments." In *Proceedings of the 29th International Conference on Very Large Data Bases-Volume 29*, 742–53.

———. 2003b. "Benchmarking the Dependability of Different OLTP Systems." In *The International Conference on Dependable Systems and Networks, DSN2003, San Francisco, CA, June*, 22–25.

———. 2005. "Towards a Security Benchmark for Database Management Systems." In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, 592–601.

VirtualBox. 2014. "Oracle VM VirtualBox." https://www.virtualbox.org/.

Walton, G. H, T. A Longstaff, and R. C Linger. 2009. "Computational Evaluation of Software Security Attributes." In *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*, 1–10.

Wang, Ju An, Hao Wang, Minzhe Guo, and Min Xia. 2009. "Security Metrics for Software Systems." In *Proceedings of the 47th Annual Southeast Regional Conference*, 47:1–47:6. ACM-SE 47. New York, NY, USA: ACM. doi:10.1145/1566445.1566509.

Ware, Willis. 1970. "Security Controls for Computer Systems: Report of Defense Science Board Task Force on Computer Security." Department of Defense of the United States of America.

Weber, D. J. 1998. "A Taxonomy of Computer Intrusions." Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science.

Weber, S., P. A. Karger, and A. Paradkar. 2005. "A Software Flaw Taxonomy: Aiming Tools at Security." In *ACM SIGSOFT Software Engineering Notes*, 30:1–7.

Weik, M. H. 1961. "The ENIAC Story." *ORDNANCE J. Amer. Ordnance Assoc. Http://ftp.arl.mil/~mike/comphist/eniac-Story.html*.

Weston, Greg. 2011. "Foreign Hackers Attack Canadian Government - Politics -

CBC News." *CBC News*, February. http://www.cbc.ca/news/politics/story/2011/02/16/pol-weston-hacking.html.

Weyuker, E. J. 1998. "Testing Component-Based Software: A Cautionary Tale." *Software, IEEE* 15 (5): 54–59.

Woo, S. W., O. H. Alhazmi, and Y. K. Malaiya. 2006. "Assessing Vulnerabilities in Apache and IIS HTTP Servers." In *Dependable, Autonomic and Secure Computing, 2nd IEEE International Symposium on*, 103–10.

Wordpress. 2015a. "Stats — WordPress.com." https://wordpress.com/activity/.

———. 2015b. "WordPress › About." https://wordpress.org/download/.

WSDigger. 2012. "WSDigger | McAfee Free Tools." http://www.mcafee.com/br/downloads/free-tools/wsdigger.aspx.