Web Application Security: A Survey of Prevention Techniques Against SQL Injection[1].

Uzi Ben-Artzi Landsmann and Donald Strömberg
`uzi-b-a@dsv.su.se, donald-s@dsv.su.se`

Department of Computer and Systems Sciences
Stockholm University / Royal Institute of Technology

June 2003

[1]This thesis corresponds to 20 weeks of full-time work

Abstract

SQL injection is an attack method used by hackers to retrieve, manipulate, fabricate or delete information in organizations' relational databases through web applications. Information in databases usually constitutes an organization's most valuable asset, and attacks on it could threaten the organization's integrity, availability and confidentiality. SQL injection techniques are simple and require no special tools or expertise from the attacker, except for some basic database and server-script language knowledge. Despite the fact that such harm can be inflicted using such simple means, it seems that only minimal resources are invested in developing security standards and in-built security measures in web applications. Organizations are using a reactive approach towards these threats, instead of a proactive approach that would help avoiding them. We have surveyed SQL injection and existing prevention techniques in order to develop a consistent terminology and to find out under which circumstances they are applicable and to what extent they can be claimed effective. Most of the information we gathered was found on web sites which discuss web security, whether in advisories, articles, papers or simply exploit guides aimed for hackers. It turns out that the area of SQL injection has never been properly surveyed from a scientific perspective. We can consequently say that the combination of countermeasures implemented during the development process is vital for making secure web applications.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

1.1.1 Corporations and Web Applications

Corporations have constantly striven to enhance their communication capabilities, allowing more efficient information exchange within their own organizations as well as between partners in their value chains, i.e. suppliers, distributors and customers. In addition, corporations have sought new alternatives to gain more competitive advantages: managing customer relationships, doing business, creating alliances, moving into new markets, and promoting their products and services. [62]

The evolution of the Internet has laid a foundation for the development and usage of new categories of information technology systems operating on the web. According to Jaquith [23] and The Open Web Application Project (*OWASP*[1]) [49] and Spett [62], these systems are often referred to as *web applications* and range in complexity from simple implementations, e.g. personal web pages, to advanced business applications: informational web sites, *intranets*, *extranets*, e-commerce and business-to-business systems. Web applications provide connectivity, access to information and online services, introducing new opportunities for corporations to realize their ambitions mentioned above. Therefore, corporations were not late to exploit these possibilities and, in a steadily increasing number, they have begun to take advantage of web applications by connecting their systems to the web.

Spett as well as Levine [29] emphasize that web applications are integrated with corporations networking infrastructure and typically encompass the use of commercial components, e.g. web servers and application servers. In addition, application logic components process input, perform calculations and generate output based on received and stored data.

1.1.2 Security

Historically, information security was primarily concerned with administrative and physical means, e.g. filing cabinets where documents could be locked in. With the widespread usage of data processing equipment, an evident need for protecting files and other information stored on computers began to concern information security. This resulted in the generic name computer security, which encompasses the collection of tools designed to protect data and defend systems against *threats*. [64]

Gollmann [18] defines computer security as the prevention and detection of unauthorized actions performed by users of a computer system. In general, three key objectives or security services are pointed out: *confidentiality*, *integrity* and *availability*. In order to protect assets in computer

[1]A glossary of words shown in italics can be found in appendix A on page 74.

systems with respect to security services, several mechanisms have been invented, including *encryption*, *authentication* and *access control*. When implementing computer security in systems one must consider important design parameters, including within which layers security mechanisms should be implemented[2].

According to Spett [62], the field of computer security has evolved over time. When the only means of compromising data was by infecting a personal computer with a virus contained on a floppy disk, desktop security was applied. In parallel with the expansion of the Internet corporations developed internal and external networks, resulting in a need for network security. As corporations intensified the offering of services through applications, computer security reached its current age, also concerning the application layer of systems: application security.

Security professionals and corporations have, according to Levine [29] and OWASP [49] and Spett, traditionally spent a major part of their security efforts and resources on operating system and network security. Assessment services heavily relied on automated tools to find holes in those layers. Conventional security measures typically included network monitoring and logging, authentication protocols, *firewalls*, intrusion detection systems and encryption techniques. Furthermore, special security measures, e.g. access control mechanisms, have been integrated in Database Management Systems (*DBMS*), to ensure database security. As a result, network inherent components such as routers and web servers as well as operating systems and DBMSs are in general easy to protect. Attack methods aimed at these components have been known for some time and standardized countermeasures have been developed and implemented to prevent and detect such threats effectively.

1.1.3 Web Applications and Security

According to OWASP [49], the increased accessibility to corporations' systems through web applications has also had an impact on the ever increasing need for computer security. Web applications pose unique security challenges to businesses and security professionals in that they expose the integrity of their data to the public.

Web application security is a concept that originates from application security and mainly comprises of threats that exploit web application *vulnerabilities*. Jaquith [23] and OWASP and Spett [62] conclude that among the new threats that have emerged, many concern the application layer of web applications, including illegal access to information, data manipulation and theft. Several attempts have been made to identify and classify the threats. Among others, Jaquith and Spett divided the threats into groups aiming

[2]Gollman also discusses four other design parameters, but we consider them to be less relevant in this thesis.

at different levels of abstraction within the attacked system. Moreover, instead of focusing on abstraction levels, OWASP [50] tried to categorize the vulnerabilities into a list of broad types. Regardless of the approach chosen, the classified threats all concern the application layer.

Basically, most web servers are separated from clients by firewalls. However, from a security perspective, web applications offer users legitimate channels through firewalls into corporations' systems. When launched from within the application logic, attacks are in general harder to detect and protect against. [23, 49, 50, 62]

The traditional approach mentioned in section 1.1.2 has therefore been proven insufficient for systems that offer services through web applications. While conventional security measures were sufficient for older systems and applications, they seem to be both outdated and ineffective in compensating for *vulnerabilities* in web applications. During the last years, an increase of vulnerabilities inherent in web applications have been noted and report of attacked systems have frequently published. [16, 18, 22, 29, 43, 51, 52, 58, 62]

Another issue of concern is the choice between a reactive or proactive security approach. With a reactive approach, security measures are in general added in the later development stages or after an application has been developed and delivered, including firewalls, anti-virus software, service packs, emergency fixes and security updates. A proactive approach, on the other hand, implies a structured approach in which security measures are added during development and are heavily tested according to a strong security policy [21, 53, 62]. One reason for the strategy of using a reactive approach might be related to short term economic profits, and the relative ease of adding third party security tools and patches. Implementing security policies and routines during the system development process, and performing code inspections and extensive testing may require more effort and resources [18, 38, 42, 45, 46, 49, 59, 68]. However, the proactive security approach should be preferred, since many web application vulnerabilities stem from improper coding in the application logic [21, 53, 62].

According to Gollman, a large community of hackers use their knowledge and experience frequently to find application layer vulnerabilities in order to commit attacks on organizations' systems through that layer. Their tools are widely spread over the Internet. This is further supported by Spett, who points out that every form of corporation acting on the Internet has become a possible target of such attacks. Those corporations may have succeeded in securing the access layers, transport layers and network layers, but may still end up with insecure systems since application level attacks can bypass security mechanisms in those layers.

1.1.4 Web Applications and Data Storage

The most serious threats that web applications may be exposed to, as Eizner [12] and Harper [21] and Newman [42] point out, concern attacks aimed directly at data storage. Web servers communicate with back-end running systems such as Relational Database Management Systems (*RDBMS*), that offer persistent storage of data in *relational databases*. Relational databases are crucial components in web applications since the most valuable information assets, corporate and customer data, are stored there. Successful attacks can cause database content and structure to be exposed, manipulated and compromised [12, 21, 42].

RDBMSs offer security mechanisms, e.g. authentication, *authorization* and access control, that when properly configured can protect stored data against illegitimate usage [10]. However, Andrews [2] and Peikare and Fogie [44] and Lie [33] point out that in web applications, application logic components open database connections through web servers to RDBMS, which in turn communicate with databases. Since web applications offer open channels to corporations web servers, the same applications therefore conduct legitimate ways, or channels, for users to fetch and manipulate data in corporate databases.

Even though extensive database security mechanisms have been implemented, every web application must act upon the channels mentioned above and make sure that only legitimate usage is allowed. Unfortunately, lack of security awareness while implementing database connections can lead to open holes in the system. [23, 29, 42, 62, 63]

While security flaws in the RDBMS are most certainly often to be found due to improper security configuration, a common mistake is to think of the RDBMS as being improperly secured. Unfortunately, as Peikare and Fogie along with Lie emphasize, an RDBMS accepts any valid and well-formed query built using a Structured Query Language (*SQL*), and as long as users have the required privileges, queries are executed on a relational database that contains the data and the result is returned. This property is one factor contributing to a new menace against relational databases connected to the web.

A chain is never stronger than its weakest link. The RDBMS, web server and application logic components are all parts of a larger system, the web application. A strong security policy for the database therefore cannot compensate for poor security in the application logic, since the overall security in the system will be equal to the component with the weakest security.

1.1.5 SQL Injection

SQL injection[3] is a particulary dangerous threat that exploits application layer vulnerabilities inherent in web applications. Instead of attacking instances such as web servers or operating systems, the purpose of SQL injection is to to attack RDBMSs, running as back-end systems to web servers, through web applications. [12, 21, 36, 42, 49, 53, 62]

More specifically, attackers can bypass existing security mechanisms implemented to enforce security services, and may therefore gain access to and manipulate information assets outside their privileges. This is accomplished by modifying input parameters expected in fields of forms embedded in web pages, in order to change the underlying queries built with SQL and passed to the database through the web server. Another method is to insert arbitrary SQL directly in the query string potion of an *URL* in the address field of web browsers. [2, 5, 12, 13, 14, 15, 21, 28, 32, 36, 38, 43, 44, 58, 62, 63]

It is difficult to prevent SQL injection attacks through a reactive security approach, mentioned in section 1.1.3, still commonly used among many developers. In addition, SQL injection requires neither specialized tools nor extensive experience and knowledge. A web browser is sufficient in order to perform SQL injection attacks against web applications, as long as the attacker has basic knowledge of HTTP, relational databases and SQL [12]. Even if the RDBMS is secured through proper configuration, the database can still be vulnerable to SQL injection attacks conducted by malicious users, as mentioned in section 1.1.4. As also mentioned there, SQL queries are executed in RDBMSs as long as they are valid and well-formed and users have the required privileges. Therefore, while security flaws are often to be found in the RDBMS due to improper security configuration, Andrew [2] and Peikari and Fogie [44] and Liu [33] conclude that one must instead consider that database security as a single measure is not sufficient to guarantee protection of data in web applications.

Every web application, using a relational database, can theoretically be a subject for SQL injection attacks. Those databases usually contain corporations' most valuable information assets: corporate and customer data. Those data are vital for the functions of a corporation's web applications, but often even more crucial and valuable for the corporation itself: user credentials, sensitive financial information, preferences, invoices, payments, inventory data etc. If successful, SQL injection attacks may therefore result in exposure of and serious impact on the corporations most valuable information assets. These attacks may in the worst case result in a completely destroyed database schema, which in turn may affect a corporation's ability to perform business. [12, 21, 27, 42]

[3]We have found several synonyms to SQL injection, but since most authors use this term, we will use it throughout this thesis.

1.2 Problem

Given the perspective of time passed since web applications entered the commercial market, SQL injection is hardly a new threat [49]. The problem has been described by many security professionals and hackers, and the information is widely spread on the Internet. Many attempts have also been made in order to find countermeasures that can contend with and overcome SQL injection threats. These countermeasures build on earlier work that covers broader aspects of computer security, including database security issues mentioned above, and the software development process itself. In addition, the countermeasures constitute new solutions regarding application layer vulnerabilities in general and SQL injection threats in particular. [2, 3, 5, 12, 13, 14, 18, 21, 28, 32, 36, 38, 41, 43, 45, 49, 50, 58, 62, 63]

It even seems to be a somewhat common assumption among writers to think that protecting web applications from SQL injection is an easy task, as long as you have an understanding of the SQL injection threat [42, 63].

Nevertheless, corporations, security professionals and hackers continue to announce that SQL injection vulnerabilities are inherent in web applications and reports of compromised applications are frequently published [5, 8, 23, 29, 36, 49]. This clearly indicates that there still seems to exist a lack of awareness, knowledge and respect of SQL injection threats inherent in web applications among security professionals.

One reason might be that software development companies and third party vendors do not take a structured security approach when developing web applications, as mentioned in section 1.1.2. Another reason is that software developers compete in introducing software to the market. We can think of two other reasons as well. First, the vast amount of information published, including detailed step-by-step guides of how to attack web applications with SQL injection, is of course also available for potential attackers. The other reason, further discussed in section 1.7, constitutes the problem for this thesis: we think that the area of SQL injection has never been properly surveyed and that the countermeasures and prevention techniques proposed has not always been systematically composed. We believe that some of the proposed prevention techniques may contain weaknesses and that they therefore can not adequately cope with SQL injection.

1.3 Purpose

We intend to perform an extensive literature survey in order to chart SQL injection and capture as many aspects of the area as we can find. Along the way, a consistent terminology will be created, existing attack methods, countermeasures and prevention techniques compiled, and security aspects regarding SQL injection expressed in terms of general criteria[4]. A general security model for SQL injection will be created and used for evaluation of existing prevention techniques.

1.4 Scope

Multi-part security standards such as Common Criteria [46, 47, 48] attempt to assure a high level of security in systems by defining security requirements that can be used for both development of computer based products and for security validation assessments. Such standards are therefore meant for use by both quality managers and developers as well as customers. Additionally, development disciplines like Software Engineering involve best practices of software development and cover areas such as process improvement and all aspects of software development processes and activities. By following this structured approach in software development, programs will be more predictable and have higher quality and security [59].

While these standards and disciplines may be important when trying to achieve a higher degree of quality and security in software products, including web applications, an examination of them lies outside the scope of this thesis. Our focus rest mainly on technical aspects of a demarcated security problem and its existing prevention techniques, and we consider therefore security standards, policies and development disciplines to be overall issues concerning a software development organization as a whole.

It should be stressed that the problem of SQL injection applies to any application that directly or indirectly communicates with an RDBMS, though we intend in this thesis to limit our research and discussions to web applications. Web applications can take many forms, as pointed out in section 2.1 on page 14. While the results of this thesis will be applicable to any application that uses an RDBMS for persistent storage of data, connected to the web or not, we intend to focus on highly sophisticated web applications, also referred to as business web applications[5] and further discussed in section 2.1.2 on page 14. The main reasons for this strategy is that corporate databases seem to be attacked more often using SQL injection and the consequences of successful SQL injection attacks affect corporations harder with respect of the inherent assets in their web applications. SQL injection has therefore mainly been discussed in that context by authors [12, 21, 27, 42].

Furthermore, as discussed in section 2.1.1 on page 14, we do not consider the concept of web services to be a part of our definition of a general web application, defined in section 2 on page 14.

[4]We use the term general in several sections, and by that we mean containing all components found during our survey, classified into coherent groups using a consistent terminology.

[5]Unless stated otherwise, we will refer to business web applications when we use the term web application.

Our examples of SQL injection attacks will not consider how data is stored in different formats, e.g. XML in an RDBMS, as discussed in section 3.1 on page 26.

We will not attempt to present new countermeasures or prevention techniques for SQL injection. In addition, when examining SQL injection, we will not consider programming language specific issues or different relational database management system vendors. Rather, we will in our survey and discussions attempt to approach SQL injection with a focus on general characteristics and mechanisms inherent in business web applications and relational database management systems.

1.5 Methodology

Our research is based on an extensive literature study which surveys SQL injection, as mentioned in section 1.3. We attempt to chart and gain as large understanding of the area as we can. Along the way, we intend to develop a uniform terminology of SQL injection and capture its characteristics, attack methods, countermeasures and existing prevention techniques. The survey also covers conditions under which SQL injection works and within which context it operates. Therefore, web applications, relational databases and SQL will also be examined, though from a security perspective. We also think it is necessary to briefly review the area of computer security.

The research is qualitative in nature, since it does not attempt to measure the effects of SQL injection attack methods and the effectiveness of existing prevention techniques empirically. Instead, we theorize about the effects of attack methods and the effectiveness of existing prevention techniques. As discussed in section 1.7, we believe that SQL injection has never been studied comprehensively and the existing countermeasures and prevention techniques have not been systematically composed. Therefore, we consider ourselves approaching an area unknown area, and under such circumstances a qualitative approach is the natural choice [20].

1.5.1 Method

We have gained inspiration and guidance from the field of security when choosing a proper method for conducting our survey. The steps in our method are therefore influenced by the notion of risk analysis, a process with high bearing on security issues, as stated by Anderson [1] and Connolly et al. [10] and Sommerville [59]. According to Sommerville, risk analysis as viewed in the context of software systems involves analyzing a system and its operational environment. The objective is to discover potential threats, their root causes and associated risks. While Sommerville emphasizes critical systems, we think that risk analysis can be used for any software system. A general web application is considered to be our system and we will focus on threats and risks associated with SQL injection. We considered the stages in the process of risk analysis as defined by Connolly et al. suitable as a base for choosing steps in our method, described below.

In the first step, we will profile web applications by exploring their scope, implementations, architecture, inherent components and communication principles. We also identify assets that we believe many web applications share.

In step two, basic principles of RDBMS systems as well as SQL will be explored. In addition, we identify security issues that we consider relevant.

Step three involves studying general aspects of computer security.

Step four constitutes our survey over SQL injection in which we classify the area in terms of general criteria. Along the way, we try to capture as many general aspects as we can: definitions, characteristics, conditions, vulnerabilities, attack methods and countermeasures. The different ways in which SQL injection-related attacks may be carried out on web applications are examined, and the attack methods will be divided into broader groups related to threats, and mapped to different security services, e.g. availability, confidentiality and integrity. In this section, we also present our general security model for SQL injection based on the general criteria. Finally, we intend to identify which contributors have taken which aspects of our model into account in their documents.

In step five, we analyze our security model constructed in step four. We also conduct an examination of an existing prevention technique against SQL injection in the perspective of our security model. Then, we try to identify required countermeasures needed in a prevention technique to adequately cope with SQL injection. Moreover, we capture which countermeasures that have been chosen by the conductors of existing prevention techniques. The results are presented in a matrix, and further compared and evaluated to identified countermeasures required. These processes also take as input the results from the final task of mapping aspects to contributors in step four. The objectives with our classification, security model and matrix are further explained in section 1.6. Finally, we theorize and analyze the necessity and complicity of the proposed countermeasures in perspective of our security model and matrix, in order to find out under which circumstances existing prevention techniques are applicable and to what extent they can be claimed effective.

1.5.2 Method for Data Collection

In order to survey SQL injection, we will first attempt to find all synonyms for SQL injection and use them as search criteria. Those criteria will be used to find any conceivable material available, published in both books and on the Internet, including proceedings, articles, white papers, technical reports, and other documents. The information about SQL injection in general and its attack methods in particular will be complemented by security advisories, FAQs and discussions within security communities and organizations on the Internet. Furthermore, discussions in hacker communities on the Internet will be considered. The process of collecting data about SQL injection will undergo several iterations.

The information about web applications will be gathered in the same way, though not equally extensive. We consider relational database management systems, SQL and computer security to be well researched subjects and therefore more documented, meaning that relevant books will probably be enough to cover them. However, where certain database specific issues are discussed in the context of SQL injection, we will check them out. We will search published guides for information about web application vulnerabilities and security issues regarding secure web application development.

1.5.3 Method Criticism

The gathered research material may not be exhaustive or even fully representative. This originates from the fact that our search criteria may not find all information that covers every aspect of SQL injection. Another reason, as mentioned in section 1.7, stems from our assumption that the available documents might not be comprehensive. However, we consider the choice of a qualitative approach to be the main problem with our method, since as such it may suffer from a lack of verification [20]. Definitions and categorizations made in this thesis are based on our assumptions: assets, threats, vulnerabilities, characteristics of attack methods and properties of existing prevention techniques. Consequently, our results also rest on those assumptions.

Furthermore, new attack methods and prevention techniques may be developed and used, even in combination with existing ones, and the results of our survey may not cope with those future challenges.

In summary, we can neither test nor try to prove our results and our research could best serve as a more systematic approach to examining and describing SQL injection.

1.6 Goals and Expected Results

This thesis addresses how an adequate chart over SQL injection, with a comparison between known prevention techniques applied in the business world and existing theory within the field of security, can complement each other and at the same time reveal weaknesses. One important aspect that will be revealed is whether the existing prevention techniques can deal with SQL injection. Another factor is whether they even aggravate the problem by introducing new security flaws. We therefore set out two objectives regarding our classification of SQL injection, security model and matrix.

The first objective concerns prevention techniques in general. We hope that our security model can be used for determining the following aspects of existing prevention techniques:

- which security services (see section 4.2 on page 35) they try to implement
- if they conform to all our defined general criteria that encompass SQL injection attack methods
- if they prevent our identified web application vulnerabilities that SQL injection may exploit
- if they overcome our defined threats associated with SQL injection
- if they help overcome other SQL injection-related vulnerabilities

In addition, we believe such a model to be a tool for evaluating material on the subject of SQL injection as well as enabling support for decision-making when evaluating web applications for SQL injection related vulnerabilities, and guidance for developing secure web applications.

Our second objective concerns only one aspect but cover all existing prevention techniques we identified. We assume that our matrix will:

- reveal weaknesses in existing prevention techniques against SQL injection with respect to proposed countermeasures
- give information that enables a comparison between the countermeasures we identified as necessary and those proposed in existing prevention techniques

1.7 Related Work

Detailed guides for building secure web applications as well as documentation over common web application vulnerabilities have been developed and made available for system architects, developers, vendors, consumers and security professionals involved in different phases of the system development process: design, development, deployment and testing [38, 45, 49, 50]. Particularly guides published by OWASP [49, 50] have been found relevant.

According to Kok [28] and Finnigan [14], the person with the pseudonym "Rain Forest Puppy" was among the first to review the techniques of SQL injection in writing. Today, extensive material on SQL injection is available on the Internet. [2, 5, 12, 13, 14, 15, 21, 28, 32, 36, 38, 43, 44, 58, 62, 63]

SQL injection has frequently been discussed in and among various security communities, organizations, conferences, forums, hacker sites and software manufacturers (see appendix B on page 77). We have also found material written about related techniques such as *cross-site scripting* and URL header manipulation as well as other features needed to take full advantage of SQL injection. Such features include poor management of generated error messages, access control structures and stored procedures in databases. [7, 31]

Indeed, the attempts to describe and overcome the problem of SQL injection are many, and they have been conducted and documented in various forms: proceedings, articles, white papers, technical reports, advisories and other documents. However, as pointed out in section 1.5.3, an overwhelming part of the documents does not consist of proceedings and articles, but of white papers, technical reports and advisories. In fact, many of the more interesting discussions of the problem comes from advisories, error reports from database manufacturers, and discussion forums and hacker sites on the Internet. The sources of information available all contain inconsistencies related to the terminology used and the scope of the problem. Moreover, often specific database vendors, products and programming languages are discussed, and we think that the overall picture is missed. In summary, we believe that an attempt to fully cover the problem in a structured and general way is needed.

Nevertheless, the material of SQL injection constitutes the information available and will, along with the material that covers the broader aspects, serve as basis for our research.

1.8 Artifacts

In order to achieve a better understanding of how SQL injection attacks can be performed and in order to verify the content of the articles we read, we have designed a simple system that allows for testing of SQL injection. The system enables the user to log in into different system configurations that use different web servers, script languages, application logic components and relational databases. This allows for testing of SQL injection attack methods and countermeasures within these configurations.

1.9 Outline

In section 2 of this thesis, we discuss common web applications including their domain, architecture and inherent components and present our own model of a general web application. We also describe how communication takes place in web applications and which assets they contain.

In section 3, RDBMSs, SQL and related security issues are covered.

In section 4, we briefly cover various aspects of computer security, including security services and mechanisms, threats, vulnerabilities, risks and countermeasures.

In section 5, we will present the first part of our contribution to the problem. We discuss and classify SQL injection, present a general security model for SQL injection, and give attack examples.

In section 6, the second part of our contribution, we will first analyze our security model of SQL injection. Then, we explore and analyze whether the existing prevention techniques is adequate to cover for the threats and vulnerabilities related to SQL injection. This is achieved both by confronting one technique against our security model and present a matrix that displays which countermeasures are discussed by which authors.

In section 7, we discuss implications of the analysis performed in section 6. We attempt to make generalizations about the existing prevention techniques against SQL injection in conjunction with security aspects within the software development process. Moreover, we briefly discuss what we believe to be interesting directions for the continued study of SQL injection. Finally, we summarize the findings of this study and attempt to draw conclusions about the existing prevention techniques against SQL injection.

2 Web Applications

In this section we discuss web applications, their architecture and the components they are composed of. We also describe how communication takes place in web applications and identify web application assets. Along the way, we intend to give our own model of a general web application which we will refer to throughout the rest of this thesis. This enables us to study our problem from a general perspective, without having to consider architectural or implementation specific details, i.e. number of tiers, and the choice of components.

2.1 Domain

People that browse the web use web applications in one form or another though, as OWASP [49] notes, the everyday web user may not be aware of that fact because of the ubiquity of web applications:

> "When one visits cnn.com and the site automatically knows you are a US resident and serves you US news and local weather, it is all because of a web application. When you transfer money, search for a flight, check out arrival times or even the latest sports scores online, you are using a web application."

2.1.1 Web Services

Before proceeding, we think it is justified to mention a few words about the concept of *web services*. As noted by OWASP [49], web services or the similar term inter-web applications, is subject for an ongoing discussion that treats web services as either the largest technology breakthrough since the web itself or simply further evolved web applications. The standpoint taken in this matter will not have an impact in this thesis but the differences and similarities between the concepts of web services and web applications will. The two concepts both ultimately face the same security issues and taking this under consideration, SQL injection, and therefore our results, is of equal relevance to web services. However, due to differences in e.g. architecture, languages and protocols between web services and web applications, we prefer to leave web services outside our definition of our general web application.

2.1.2 Business Web Applications

OWASP [49] states that any software application built on client-server technology that operates on the web and that interacts with users or other systems using HTTP, could be classified as a web application. The client-server architecture is discussed in section 2.2 and HTTP is discussed in section 2.4.3.

Jaquith [23] and OWASP and Spett [62] note that web applications provide connectivity, access to information and online services for users. Web applications can, according to Spett, today be implemented in various degrees of complexity, and each implementation has a distinct purpose: "... an informational website, an e-commerce website, an extranet, an intranet, an exchange, a search engine, a transaction engine, an e-business." The functions performed can therefore, as OWASP notes, range from relatively simple tasks like searching a local directory for a file, to highly sophisticated applications that perform real-time sales and inventory management across multiple vendors, including both Business to Business and Business to Consumer e-commerce, flow of work and supply chain management, and legacy applications. Corporations make use of sophisticated web applications, also referred to as business web applications, in order to accomplish more efficient information exchange within their own organizations as well as between partners in their value chains, i.e. suppliers, distributors and customers. In addition, web applications offer corporations management of customer relationships, new ways of doing business and creation of alliances, movement into new markets, and promotion of their products and services. In order to be useful and comply with their purpose, web applications require persistent storage for their data. Usually, a RDBMS is chosen in order to achieve this [12, 21, 42].

2.2 Architecture

From a hacker's perspective, a corporation's web application can be viewed as a horizontal value chain of layers [62]. As we shall see in section 5, this point of view is important to consider, when discussing the analogy of an attack against a given web application. However, we think that in the context of web application architecture, focusing on where different kinds of tasks are processed is more appropriate. Therefore, an examination of the client-server architecture is motivated.

2.2.1 Client-Server

According to Connolly et al. [10], the web itself is comprised of a network of computers, and each computer acts in different roles: as a client, a server or both. In order to accommodate an increasingly decentralized business environment, web applications operating on the web use the client-server architecture. The term client-server, as mentioned by Connolly et al., refers to the processes with which software components interact to form a system, i.e. client processes require resources provided by server processes. Combinations of the client-server architecture, or topology, include: (a) single client, single server; (b) multiple clients, single server; (c) multiple clients, multiple servers. The client in a web application is usually represented by a web browser like Internet Explorer or Netscape Navigator. Servers typically include web servers, e.g. Microsoft Internet Information Server, Apache and Tomcat [6, 10, 49].

In a client-server architecture, applications can be modelled as consisting of different logical strata. One problem is that there are different opinions regarding about the meaning of logical strata, i.e whether to view them as layers or tiers. We prefer the approach of dividing strata into a software view and a hardware view. In the software view, the strata consist of layers and in the hardware view, tiers represent the strata. [9, 10, 17, 35, 49, 59, 67]

2.2.2 The Client-Server Architecture and Layers

Before proceeding, we stress that the layers we will refer to throughout this thesis concern responsibilities and task processing in web applications, and not how communication in networks are organized into abstraction levels. Therefore, we do not consider the layered approach taken in models such as the Open Systems Interconnection (OSI) reference model to be relevant in our discussions.

In a client-server architecture, applications can be modelled as consisting of logical layers. While there exist different conventions for naming those layers, we conclude that the following three different layers are included [9, 10, 17, 35, 49, 59, 67]:

Presentation The layer where information is being presented to users and which constitutes the interaction point between users and the application. This layer is actually constituted of two parts, where one part is dedicated to the client-side and the other part concerns the server-side. While this layer generates and decodes web pages, it can also be responsible for presentation logic, meaning that components of this layer can reside both on the client-side and server-side. Distributed logic needed to connect to a proxy layer on the server-side along with a proxy tier in order to make use of middle-ware, e.g. CORBA and RMI, could also reside here.

Application logic The layer where application logic and business logic and rules are implemented. This layer processes user input, makes decisions, performs data manipulation and translation into information, including calculations and validations, manages work flow, e.g. keeping track of session data, and handles data access for the presentation layer.

Data management The layer responsible for managing both temporary and permanent data storage, including database operations.

2.2.3 The Client-Server Architecture and Tiers

We have found several different models which describe how web applications are composed using the logical layers mentioned in section 2.2.2 [9, 10, 17, 35, 49, 59, 67]. One main characteristic shared by those models constitutes the combination of logical layers into a 2, 3 or n-tier architecture[6] in order to provide for a separation of tasks, where a tier is defined as one of two or more rows, levels and ranks arranged one above another. While the different tiered approaches turn out to be irrelevant in our general model, as explained in section 2.2.4, we think that they are worth mentioning for the sake of clarity.

Two-Tier Architecture
In the two-tier client-server architecture, mentioned by Connolly et al. [10] as the basic model for separating tasks, clients constitute the first tier and servers the second tier. A client is primarily responsible for presentation services, including handling user interface actions, performing application logic and presentation of data to the user and performing the main business application logic. The server is primarily concerned with supplying data services to the client. Data services provide limited business application logic, typically validation of the client and access control to data. Typically, the client would run on end-user desktops and interact with a centralized DBMS over a network.

Three-Tier Architecture
In a three-tier architecture, the first tier still constitutes the client which is now considered a thin client, i.e. is only responsible for the applications's user interface and possibly simple logic processing, such as input validation. The core business logic of the application now resides in its own tier, the middle tier, that runs on a server and is often called the application server. The third tier constitutes an RDBMS, which stores the data required by the middle tier, and may run on a separate server called the *database server.*

N-Tier Architecture
This type of architecture simply implies any number of tiers. One example of this is when the web server and database server reside in separate computers. Another example is when several database servers are used and one computer is dedicated responsible of managing access to each database server, running on separate computers. [9, 10]

[6]Our discussions of tiered architectures are made from the database context as explained by Connolly et al. [10].

2.2.4 General Web Application Architecture

We would like in this thesis to address the common characteristics of web applications and discuss web applications from a security perspective. Irrespective of the differences found in implementations of web applications discussed in section 2.1 and in the choice of tiers, web applications all contain the three logical layers mentioned in section 2.2.2. Since we in this thesis intend to center our discussions around one consistent model, we will discard arguments for different tiered models, i.e. how layers are composed into a number of tiers, and instead concentrate on the processes inside and between the different layers, defined in section 2.2.2. We have also found several reasons that support our intention.

As applications became more complex and potentially could be deployed to hundreds or even thousands of end-users, the traditional two-tier architecture was challenged. According to Connolly et al., this mainly stems from the fact that the client-side presented two problems that prevented true scalability. First, they rely on fat clients, i.e. clients that require considerable resources on the client's computer to run effectively: disk space, work memory and processor power. Secondly, a significant client-side administration overhead is required. The solution became the three tiered model, each tier potentially running on a different platform. Hence, the arguments for a tiered model of choice more depends on factors such as performance and scalability rather than security.

Among others, Jaquith [23] and Spett [62] furthermore argue that architecture issues regarding tiers and layers are less relevant when looking at web applications from the perspective of the vulnerabilities that SQL injection exploits. Weaknesses related to security most certainly exist in system architectures. However, web application vulnerabilities more heavily stem from weaknesses both in practices such as application design, coding and implementation in the development processes, as well as in the process of system configuration. When discussing system architecture, the components chosen for each layer are more important than how layers are combined into tiers. Furthermore, different authors assume different tiered architectures when discussing web applications and SQL injection. For example, Anley has wrote two documents where he discusses a two-tier and a three-tier architecture respectively, and argues that in order to protect web applications from SQL injection attacks, the tiered architecture of choice is not the main factor of concern [3, 4].

In OWASP's guide for developing secure web applications [49], which we think is both extensive and representative, all discussions around architecture issues relating to web application security focus on layers, not tiers.

2.3 Components

In order to be able to define our own model of a general web application and its inherent components, we have studied a number of existing models. [9, 10, 14, 17, 19, 35, 49, 56, 59, 67]

OWASP [49] points out that the discussion of what components a web application actually consists of is rife with confusion. We can also conclude that the terms found in different models have no clear definition, are used differently by different authors, and even overlap each other. Among others, Jaquith [23] argues that not all applications are created equally with respect to the chosen components. Most of the components are largely interchangeable. Development languages, web servers, application servers, middleware and databases can all be used in secure and insecure manners. Hence, vulnerabilities inherent in web applications do not solely rely on the type of components chosen. Rather, vulnerabilities stem from weaknesses in practices such as design and coding in the development process, as mentioned in section 2.2.4.

For these reasons, we selected a collection of components from the available material that we consider representative in order to compose the general web application we will refer to throughout this thesis. These components were then grouped according to their logical layer membership defined in section 2.2.2 and furthermore divided into different types. Note that the presentation layer is divided into two layers, where one layer is dedicated to the client-side and the other layer resides on the server-side. Moreover, we consider some components as occurring both at the client-side and the server-side. This stems from the fact that some components, e.g. web pages, are stored or generated at the server-side, but they are sent to the client-side. We think it is necessary to consider them as belonging to both sides in the perspective of SQL injection, as we shall see in section 5.

2.3.1 Component Types

We have identified three distinct types of components that we consider meaningful in order to avoid overwhelming this thesis with complexity and to allow a discussion of SQL injection in respect of where different tasks are processed within a web application:

Software components if they are a part of software implementation.

Data-processing components if they operate on data.

Data components if they are stored or processed data.

2.3.2 A General Business Web Application Model

In this section, we describe our model of a general web application[7] with its components mapped to layers. The components chosen and described are not meant to be exhaustive. Rather, our intention is to present a foundation for our discussions in section 5. Our model is also illustrated in figure 1.

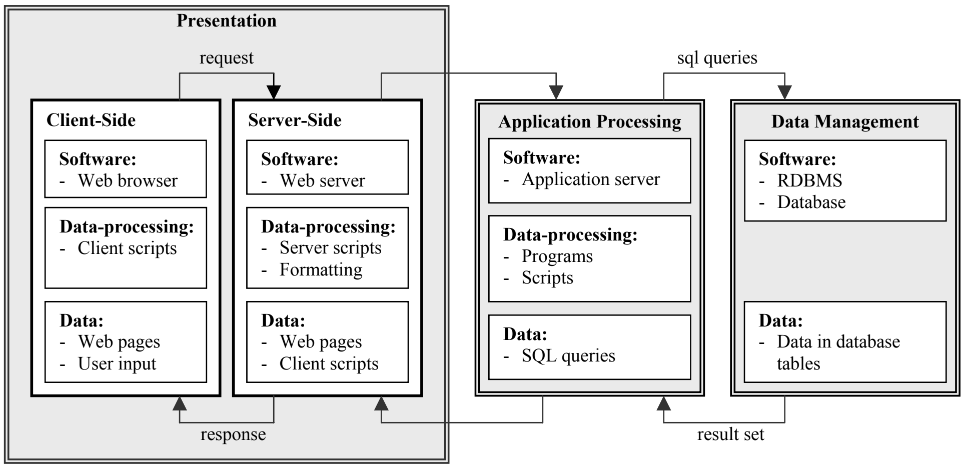

Figure 1: Components within layers

Presentation layers

Software components

- A web browser, e.g. Microsoft Explorer, Netscape Navigator, Mozilla and Opera, which operates at the client-side. A browser loads static and dynamic web pages along with client-scripts and stylesheets from a web server, and presents them in a graphical user interface. This component may also constitute the point where users interact with the application.
- A web server, e.g. iPlanet, Apache, Zeus, Microsoft IIS and Netscape Enterprise, which operates at the server-side, receives requests from the client-side and processes them. The server returns responses containing data components such as static or dynamic web pages. The dynamic web pages may be generated by the server itself or by data-processing components.

[7]Unless stated otherwise, we will refer to our model of a general business web application when we use the term web application.

Data-processing components

- Web browser scripts written in script languages such as JavaScript or VBScript that constitute presentation logic on the client-side. They extend user interactivity and perform tasks, e.g. input validation.
- Server scripts such as Active Server Pages (ASP) or Java Server Pages (JSP) that perform presentation logic on the server-side and generate dynamic web pages.
- Formatting components written in stylesheet languages, including Cascading Style Sheets (CSS) and The Extensible Stylesheet Language Transformations (XSLT). XSLT is designed for use as part of XSL, a stylesheet language for XML, and transforms XML documents into other XML documents.

Data components

- Web pages written in e.g. *HTML* and the Wireless Markup Language (WML), that are stored on the server-side, loaded into the client-side and presented in the web browser. They can be static, i.e. hard coded, or dynamic, i.e. generated by server-side scripts.
- User input data that is entered in forms of web pages or the URL header in the web browser.
- Client scripts sent to the client-side.
- Request objects sent from the client-side to the server-side using HTTP, *HTTPS* or XML.
- Response objects sent from the server-side to the client-side using HTTP, HTTPS or XML.
- Data sent between the server-side of the presentation layer and application processing layer.

Application processing layer

Software components

- An application server that serves as a framework or development environment for technologies that implement application or business logic in stand-alone programs. Examples of application servers include IBM Websphere, BEA Weblogic, JBoss and iPlanet, Zone and Zend. Technologies include Servlets and Enterprise Java Beans written in Java and the Common Gateway Interface (CGI) that takes advantage of scripts written in various languages, including C, Perl and Python. Other examples of technologies include PHP, Visual Basic (VB) and .NET services.

Data-processing components

- Stand-alone programs, e.g. Servlets or Enterprise Java Beans.
- CGI scripts written in languages such as C, C++, Java or Perl.

Data components

- Data sent between the application processing layer and the server-side presentation layer.

Data management layer

Software components

- A RDBMS that inserts, retrieves and manipulates data in a database through SQL queries, as well as controls access to the database through access control mechanisms.
- A relational database that contains valuable data.

Data components

- Data in database tables.
- Data and information sent between the application processing layer and data management layer, e.g. SQL queries and result sets containing data from the database.

2.4 Communication

A typical communication exchange in a business web application, according to Connolly et al., is initiated by users that request information. The client takes a user's request, checks the syntax and generates database requests in e.g. SQL. Then, the client transmits the message to the server, waits for a response, and formats the response for the end-user. The server accepts and processes the database requests, then transmits the results back to the end-user.

2.4.1 Information

Information on the web is stored in documents and the formatting language, or system, most commonly used is the HTML. Using HTML, documents are marked up, or tagged, to allow for publishing on the web in a platform-independent manner. HTML documents are displayed in web browsers, that understand and interpret HTML. [10]

2.4.2 Content

HTML documents stored in files constitute static content, i.e. the content of the document does not change unless the file itself is changed. However, documents resulting from requests such as queries to databases need to be generated by the web servers. These documents are dynamic content and as databases are dynamic, changing as users create, insert, update, and delete data, the generation of dynamic web pages is a more appropriate approach than static content, particularly in web applications. [10]

2.4.3 Protocol

The exchange of information in web applications is mainly governed by protocols such as HTTP or HTTPS, which define how clients, i.e. web browsers, and servers, i.e. web servers, communicate. HTTP relies on a request-response paradigm and a transaction consists of the following stages [10, 19, 49]:

Connection The client establishes a connection with the web server.

Request The client sends a request message to the web server

Response The web server sends a response, i.e. a HTML document, back to the client.

Close The connection is closed by the web server.

Basically, a request in a HTTP connection constitutes an object containing, e.g. a requested resource. Consequently, a response is the result to be presented in the web browser. When a user visits a page, web pages, client-side scripts and formatting components are sent back to the client for rendering and presentation. In the case a user requests data contained in a relational database, user input parameters are typically embedded in the request. Those parameters can be included as arguments to methods in application processing components that dynamically build SQL queries. They may also indicate which SQL query is to be executed, in case that static SQL is used, e.g. stored procedures or prepared statements (see section 3.3.2). The response object will contain data for presentation in the web browser. That data may have been parsed and prepared by either application processing components, server-side scripts or both for rendering purposes: either for tailoring the graphical design or ease the rendering process in the web browser.

HTTP brings up several security weaknesses. A HTTP request is composed of different parts and attackers can manipulate those parts in order to try to bypass security mechanisms. The web server listens on an open port for incoming requests from clients. For general web traffic, i.e. HTTP, port 80 is often used as the default port and for encrypted traffic, i.e HTTPS, port 443 is normally chosen. However, each web server requires a unique port to listen to and since corporations can have several web servers, the port of each server has to be configured. Moreover, application servers require open ports as well. While this means that an attacker can not always assume that the web server of choice listens to port 80, the important issue is that there exists an open port through security mechanisms such as firewalls into a corporations web server. [62]

2.4.4 URL Encoding

According to OWASP [49], a server can receive input from a client in two basic ways: either data is passed in HTTP headers or it can be included in the query portion of the requested Uniform Resource Locator (*URL*), which uniquely defines where resources can be found on the Internet. Both methods correspond to two methods used when including input in client requests: GET and POST. Manipulation of a URL or a form is simply two sides of the same issue.

However, when data is included in a URL, it must be specially encoded to conform to proper URL syntax. Unfortunately, as OWASP notes, the URL encoding mechanism allows virtually any data to be passed from a client to the server. Proper precautions must be taken by the application logic, as discussed in section 5 when accepting URL encoded data since this mechanism can be used for disguising malicious code.

2.5 Assets

The task of defining web application assets is not entirely simple. Various types of implementations exist and they all have a distinct purpose. While applications may conform to the same type of implementation, e.g. an intranet, each corporation may view their assets differently. We consider that to be one explanation why defining a set of assets suited for our model was not a clear case.

Some authors, including Connolly et al. [10] generalize computer system assets into distinct categories. Others give concrete examples, rather than definitions, of assets that mainly concern information contained in web applications [12, 21, 42]. Since we consider web applications to be, or at least be part of, computer systems our list of assets contain compiled generalizations and further complemented with more specific examples that conform to our definition of a business web application.

We conclude that assets can either be tangible or intangible. The list below constitutes tangible assets:

Hardware Network infrastructure components such as routers, hubs, switches, gateways and cables that connect these devices. Hardware also includes computers running server and client software, and components inherent in computers, e.g. hard drives. Network printers and scanners are other examples of hardware.

Software Any software or data-processing component defined in section 2.3.2.

Data Data components defined in section 2.3.2. Data stored in relational databases include configuration data, database tables mentioned in section 3.1 and data contained in those tables, e.g. user credentials, sensitive financial information, preferences, invoices, payments, and inventory data.

Intangible assets are less obvious and more difficult to define, rank and evaluate. Connolly et al. give organization credibility and customer confidence as examples of intangible assets.

Tangible assets seem to be most frequently discussed. One reason for this might be that, from a computer security perspective, security measures for intangible assets can be more difficult to depict. The effects of an unavailable web server or loss of data in a corporate database will probably be noted quickly whereas customer dissatisfaction may take longer to be revealed. Nevertheless, intangible assets such as customer confidence ultimately can be transformed into tangible assets, e.g. clients. If organizations begin to lose clients or face difficulties in acquiring new ones, they begin to lose money. Therefore, as we also shall see in section 5, we think that excluding intangible assets will be a mistake that seriously affect an organizations efforts spent on securing web applications, irrespective whether the organization is a software developer, contractor or buyer.

3 RDBMS and SQL

In this section we give an introduction to Relational Database Management Systems (RDBMS) and the Structured Query Language (SQL), its syntax and usage. We do not intend to a give a complete review of these subjects, as we consider this to be outside this thesis' boundaries. This section relays heavily on the book written by Connolly et al. [10] and its comprehensive explanation of SQL.

3.1 RDBMS

Connolly et al. [10] defines a database as "A shared collection of logically related data (and a description of this data), designed to meet the information needs of an organization." The database is a shared resource of data which is used by organizations and is among the most crucial components in web applications [21, 42]. A database management system, DBMS, is used to allow user interaction with the database. Such DBMSs are defined by Connolly et al. as "A software system that enables users to define, create, and maintain the database and provide controlled access to this database."

According to Gollman [18], the most widely used database model in databases today is the relational model, which is used to organize relational databases. A DBMS which relies on the relational model (an RDBMS) is a system through which users can administrate a database which is perceived as a collection of tables. These tables (or relations) are organized as a two dimensional array containing rows and columns. A table's rows (or tuples) are the elements of the table, and its columns (or attributes) are the names of data that is represented.

As Connolly et al. state, the Structured Query Language (SQL) has become the standard language used in relational databases and is the only database language to gain wide acceptance. This language allows users to administrate databases using an RDBMS as well as communicating with them.

3.2 SQL

The SQL standard, according to Connolly et al. [10], was defined by the American National Standards Institute (ANSI) and was later adopted by the International Standards Organization (ISO). Its objectives are to allow users to create database and relation structures, managing tables by inserting, modifying, and deleting data as well as retrieve information from the database through queries. SQL queries are commands that are passed to the RDBMS, and specify which data is to be gathered from one or more tables and how it should be arranged. We intend to follow the ISO SQL standard used by Connolly et al.[8], and will be using it throughout this thesis unless stated otherwise.

SQL consists of two major components: the Data Manipulation Language (DML) and the Data Definition Language (DDL). Using the DML, users can manipulate data stored inside tables in the database, while the DDL allows creating and destroying database objects such as schemas, domains, tables, views and indices.

3.2.1 DML

The Data Manipulation Language has four available statements, namely SELECT, INSERT, UPDATE and DELETE. We describe each of these statements according to Connolly et al. [10] using the syntax described in table 1.

Symbol	Represents
SELECT, INSERT, ...	reserved words
table_name, column_list, ...	user-defined words
|	choice among alternatives
{}	required element, for example {a}
[]	optional element, for example [a]
...	optional repetition (zero or more times)

Table 1: SQL syntax

SELECT used for retrieving information from one or more tables in the database and displaying it.

The syntax of the SELECT statement is given below:

```
SELECT [DISTINCT|ALL]
    {*|column_expression[AS new_name]][,...]}
FROM table_name [alias][,...]
[WHERE condition]
[GROUP BY column_list][HAVING condition]
[ORDER BY column_list]
```

where **column_expression** represents a column name or expression, **new-_name** is a new temporary name to use for the column_expression, **table_name** represents the name of the database table or view table to select from, **alias** represents an optional name for the table_name, **condition** is the condition upon which selection is made or a condition for display (see HAVING), and **column_list** represents the list of table columns to group or order the result upon.

[8]ISO 9075:1992(E)

The sequence of the SELECT statement processing, and the meaning of the reserved words are:

FROM	specifies which tables to choose from
WHERE	filters the selected data rows due to a condition
GROUP BY	groups together rows with same column value
HAVING	filters the selected groups due to a condition
SELECT	specifies which column should appear in the result
ORDER BY	specifies the order to sort the output upon

INSERT used for adding new data rows in a table.

The syntax of the INSERT statement is given below:

```
INSERT INTO table_name[(column_list)]
VALUES(data_value_list)
```

where **table_name** represents the name of the database table or view table, **column_list** represents the list of table columns to update, and **data_value-_list** represents the list of values to enter into each column in the new row. The number, position, and type of data values must correspond to the table's column list.

UPDATE used for modifying data rows in a table.

The syntax of the UPDATE statement is given below:

```
UPDATE table_name
SET column_name1 = data_value1
 [, column_name2 = data_value2...]
[WHERE search_condition]
```

where **table_name** represents the name of the database table or view table, **column_name** represents the column name to modify, and **data_value** represents the new value to enter into the column. The new given value must correspond to the table's column. The WHERE clause specifies which row is to be modified, according to the **search_condition**. If omitted, the whole table will be affected.

DELETE used for removing data rows from a table.

The syntax of the DELETE statement is given below:

```
DELETE FROM table_name
[WHERE search_condition]
```

where **table_name** represents the name of the database table or view table, and the WHERE clause specifies which row is to be modified, according to the **search_condition**. If omitted, all data in the table will be deleted.

SELECT statements can be used to retrieve data in many different ways. In order to explain this, we give here a list of different query formulations and the way they are commonly used according to Connolly et al.

- **Simple Queries** can be used to retrieve either all or a selection of columns and rows from one or more tables. A condition can also be specified to minimize the selection.
- **Sorting Results** can be achieved by using the ORDER BY clause.
- **Aggregate Functions** are used to retrieve numeric information about the data. The clauses COUNT, SUM, AVG, MIN and MAX are used to retrieve number of rows, sum of values, values average, minimum value and maximum value, respectively.
- **Grouping Results** can be achieved using the GROUP BY clause.
- **Sub queries** can help creating complex queries wherein result from a secondary query can used for instance as a condition for the primary query.
- **The ANY and ALL Clauses** can be used to compare results of a primary query with all or any of the results of a secondary query.
- **Multi-Table Queries** are used to combine columns from different tables through usage of different JOIN clauses.
- **The EXISTS and NOT EXISTS Clauses** can be used to check if a value exists or not in a table or in a result from a secondary query.
- **Combining Result Tables** can be made using the UNION, INTERSECT and EXCEPT clauses.

3.2.2 DDL

Connolly et al. [10] defines the Data Definition Language (DDL) as "A descriptive language that allows the DBA or user to describe and name the entities required for the application and the relationships that may exist between the different entities." Thus, the DDL is used when manipulating the database's meta-data, which describes the objects contained in the database and allows access to them. The DDL does not allow users to manipulate data stored in the database.

3.3 Query Techniques

An SQL query to be executed in a RDBMS can be constructed using two techniques. Either the query is allowed to be dynamically tailored with respect of both SQL keywords and query arguments, or the query syntax is unchangeable, only allowing arguments to be passed [10].

3.3.1 Dynamic SQL

Dynamic SQL refers to the concept of allowing an SQL query to be dynamically built by concatenating statements and using variables that supply the query with dynamic values. According to Connolly et al. [10] and Harper [21] and Khatri [25], the query is typically stored in a variable and the query builders consist of application logic components that adds SQL syntax and arguments to the variable in a process governed by specified conditions. Such queries are interpreted and compiled at run-time by the RDBMS, meaning that the query will be compiled every time it is executed. Since dynamic SQL allows SQL syntax to be added, both SQL keywords and values may be passed as arguments to queries. This may cause unexpected results, further discussed in section 5.

3.3.2 Static SQL

Static SQL refers to the concept of using fixed and unchangeable SQL queries. Such queries are predefined and compiled and are not permitted to add SQL keywords, defined in DDL or DML. Only arguments to clauses, e.g. WHERE, may be allowed to be passed to the queries. Either the query is embedded in application logic code in form of *prepared statements* or it resides in RDBMS as stored procedures.

Stored procedures are pre-compiled collections of SQL statements, or sub-routines, that reside in the RDBMS. Either they are supplied by the database vendor, i.e. system stored procedures, or additionally constructed by system developers, database administrators or application programmers. They allow a developer to access and manipulate databases quickly and efficiently. Since they are compiled in advance, they possess the property of being executed faster than dynamic SQL. Another property is that once created, stored procedures cannot be modified via dynamic SQL. Stored procedures are executed by invoking a command that includes the procedure identifier. This can be done either from a command prompt or from application programs written in languages such as C or Visual Basic. Several RDBMS supports this feature, but the set of stored procedures that follow with the installation and the syntax for invoking them varies. [10, 13, 21, 25]

3.4 Error Messages

RDBMSs have in-built error handling mechanisms that may generate error messages when for example an SQL query could not be executed. The error message format used and degree of details embedded in generated messages vary from RDBMS to RDBMS. Nevertheless, if you run a query and accidentally make a mistake by entering e.g. a table that does not exist in the database, the RDBMS may return an error message containing information about the error. Some RDBMSs even react to all errors in the same manner, whether those errors are generated by users, databases, objects, or the system. [31, 61, 63]

Error messages are typically propagated back to the source that caused the error. The web server or application server will propagate an error page that displays the message to the client. Web application developers can take advantage of these messages for debugging purposes, as noted by Spett [63]. However, as we shall see in section 5, it is not a wise error-handling approach to let web servers display these error messages in error pages to users of web applications.

3.5 Security

Database security, according to Connolly et al. [10], concerns "The protection of the database against intentional or unintentional threats using computer-based or non-computer-based controls." Besides the effect that poor database security can have on the database, it may also threaten other parts of a system and thus an entire organization. The risks related to database security are:

- **Theft and fraud** which are activities made intentionally by people. This risk may result in loss of confidentiality or privacy.
- **Loss of confidentiality** which refers to loss of organizational secrets.
- **Loss of privacy** which refers to exposure of personal information.
- **Loss of integrity** which refers to invalid or corrupt data.
- **Loss of availability** which means that data or system cannot be reached.

Threats that correspond to those risks are such situations or events in which it is likely that an action, event or person will harm an organization. Threats can be tangible, that is, cause loss of hardware or software, or intangible, as in with loss of credibility or confidence. In order to be able to face threats, a risk analysis should be conducted, in which a group of people in an organization tries to identify and gather information about the organization's assets, the risks and threats that may harm the organization and the countermeasures that can be used to face those risks. Decisions made using such risk analysis are thereafter used to implement security measures in the system. These security measures can be computer-based controls or non-computer-based controls.

3.5.1 Computer-Based Controls

According to Connolly et al. [10], computer-based controls are used for protecting DBMS through means of authorization, views, backup and recovery, integrity, encryption and associated procedures.

Authorization
Authorization is used to define which activities (or privileges) are granted to different users (or subjects), which allows them to manipulate or retrieve information from different database objects. In order to ensure that the user is who she claims, authentication is used. Usually, a simple mechanism of usernames and passwords is used, whether in the DBMS or in combination with the operating system where the DBMS resides. A user is asked to fill her name and password, and the authentication mechanism confirms that the user is who she claims to be by comparing the password with the corresponding password in a list it maintains.

The DBMS usually maintains a list of privileges that subjects have on certain database objects. A DBMS that operates as a closed system, maintains a privileges list in which users are not allowed to operate on any objects except the ones in the list. A DBMS that operates as an open system, on the other hand, allows users to operate on all objects except those that are explicitly removed and listed in the privileges list.

Privileges may also be group-based or role-based. Both users and objects may be joined in a group and privileges may be given to a group of users or objects. Certain roles can also be given privileges on objects, and a number of different users may undertake a certain role.

Views
Views are virtual tables that are created through some operations on database objects. By removing certain columns or rows and combining certain tables, such views can be used to limit the scope of objects that users can manipulate or retrieve information from.

Backup and Recovery
In order to be able to recover from a failure, a DBMS must regularly make a copy of the database and log files. Log files are a list of activities made in the database that can be used to recover the database after a failure. Checkpoints made in certain time intervals can assure that the backup and

log files are synchronized. This allows for safe recovery since operations that are listed in the log file need only be carried out from the point in time when the last backup was made.

Integrity
Integrity controls can be used to see to that data in database does not get corrupt. Such controls are called relational integrity controls, and are rules that some databases implement internally to maintain data validity. Other database do not implement those controls and it is up to the application programmer that uses the database to see to that data validity is being maintained.

Encryption
Encryption is a method that is used for encoding the data so that other programs cannot read it. Some DBMSs contain an internal encryption mechanism, while other relays on the operating system or third-party programs.

Associated Procedures
Connolley et al. describe some associated procedures that should be used to further protect the database:

- **Authorization and authentication:** in order for these mechanisms to work properly, a password policy should be maintained, which regulates matters like minimum passwords length, how often they should be replaced, as well as revoking old passwords.
- **Backup:** procedures should regulate how often backups should be made as well as what parts of the database backups should include. Furthermore, backups should be kept in a safe place.
- **Recovery:** recovery mechanisms should regulate how backups and logs can be used in case of failure. These procedures should also be tested regularly.
- **Audit:** audits should be carried out regularly to control the security and to see to that all mechanisms are adequately functioning.
- **Installation of new software:** before any new software is to be installed, it should be properly tested so that it would not harm any data or mechanisms.
- **Installation/upgrading of system software:** any system upgrades should by documented and reviewed. Before such upgrades take place, the risks of such an act should be considered and plans should be made for possible failures and changes.

3.5.2 Non-Computer-Based Controls

The most important countermeasure among non-computer-based controls is the security policy and the contingency plan. A security policy concerns security maintenance in an organization, and contains "...a set of rules that state which actions are permitted and which actions are prohibited." [18] A contingency plan is a detailed description of the actions that should be taken in order to deal with unusual events, such as sabotage, fire or flood.

4 Computer Security

In this chapter, we define and discuss various components of computer security. Furthermore, we try to give a comprehensive image of what computer security is and which role it has in organizations.

4.1 Assets

The goal of security, according to Gollmann [18], is to protect an organization's assets, meaning to prevent assets from being damaged, detect when such damage occurs, and then react in order recover the assets.

4.2 Services

Computer security, according to Gollmann [18] "...deals with the prevention and detection of unauthorized actions by users of a computer system." A few aspects of the above mentioned assets must be protected in order to maintain computer security. Gollmann mentions that three aspects are most frequently proposed, namely confidentiality, integrity and availability. Other aspects that are worth mentioning, according to Gollmann, are authenticity and accountability. Below we list Gollmann's definitions to these aspects:

Confidentiality prevention of unauthorized disclosure of information

Integrity prevention of unauthorized modification of information

Availability prevention of unauthorized withholding of information or resources

Authenticity verification of claimed identity

Accountability ability to trace responsible party for information audition

Stallings [64] calls these aspects security services, and points out that "A service [...] enhances the security of the data processing systems and the information transfers of an organization". Stallings adds two services to the above list:

Nonrepudiation as a corollary of accountability, an object cannot deny its part in an event

Access Control limitation and control of the access to host systems and applications via communication links

These services, according to Stallings, attempt to resist security threats, using one or more security mechanisms.

4.3 Threats

Connolly et al. [10] defines threats as any event that could adversely affect a system, and consequently an organization. This affect can harm a system in various ways, and can be caused by person or an action, whether intentionally or unintentionally. In this thesis we intend to concentrate our discussion about intentional threats, or security attacks, as Stallings [64] refers to them. These threats can be grouped by the security service that they threat:

- **Interruption**: an attempt to destroy an asset or make it unusable, i.e. a threat to availability
- **Interception**: an attempt to gain access to an asset, i.e. a threat to confidentiality
- **Modification**: an attempt to tamper with an asset, i.e. a threat to integrity
- **Fabrication**: an attempt to create objects in the system, i.e. a threat to authenticity

Furthermore, Stallings suggests that attacks (which are source to threats) can be divided into two categories: passive attacks, in which the attacker is trying to eavesdrop or monitor information, and active attacks, which involves modification or fabrication of data.

4.4 Mechanisms

Stallings [64] defines security mechanisms as any "...mechanism that is designed to detect, prevent, or recover from a security attack." These mechanisms are thus used to implement the security services and maintain their specific security aspect. Examples of different mechanisms are given below:

- **detection:** intrusion detection mechanisms continuously monitor the system, perform event logging and alert administrators of detected attacks.
- **prevention:** firewalls prevent unauthorized information traffic to pass into or out from internal networks.
- **recovering:** logs and backups enable a system to recover after system failure or successful attacks, which might have damaged parts of the system.

4.5 Vulnerabilities

Anderson [1] and Stallings [64] refer to vulnerabilities as breeches in security mechanisms that can be used to preform attacks and thus constitute a threat to a computer system. Examples of vulnerabilities are given below:

- lack of implemented security mechanisms, e.g. ignoring the virus threat by not installing anti-virus programs.
- deficient configuration of security mechanisms, e.g. configuring firewalls 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.

4.6 Relation Between Security Components

In order to summarize this discussion we present the security components discussed above and the relations between them in figure 2.

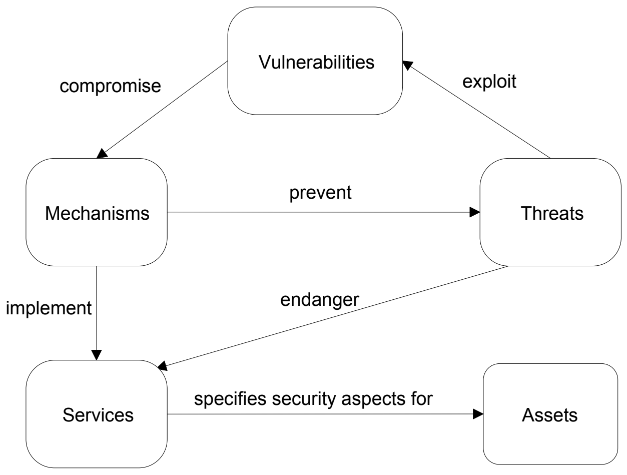

Figure 2: Relation between security components

5 SQL Injection

This section constitutes the first part of our attempt to contribute to the SQL injection problem. The area of SQL injection, including definitions, context of operations, conditions, vulnerabilities, attack methods, and existing prevention techniques will be explained and classified. Furthermore, we present a model to be used as a base for discussion when evaluating existing prevention techniques against SQL injection. Finally, we give some attack examples.

5.1 Introduction

5.1.1 Scope

We will view SQL injection as the majority of the authors do: a technique used for manipulating server-side scripts that send SQL queries to an RDBMS. This is done by manipulating client-side data, including changing SQL values and concatenations of SQL statements, which are sent to a web server embedded in HTTP requests. Once the web server receives a request, it forwards the information in it to a script which uses that information to build SQL queries. The goal of the attacker who uses SQL injection is to manipulate with the SQL query used by the script so that it would yield unwanted results, such as fetching, inserting, manipulating or deleting protected rows or tables in the database. [2, 5, 8, 12, 13, 14, 15, 21, 24, 27, 28, 38, 43, 52, 62, 63].

Before proceeding, we think that a discussion of the scope of SQL injection is necessary. Attack methods of SQL injection have by some authors been classified into direct and indirect attacks.

Using direct attacks, an attacker tries to take control of an RDBMS. The purpose of such attacks is to further take control of other host computers and compromise a network. First, attackers scan for open ports that database servers are listening to. If such ports are found, they continue with executing system commands through a command console, communicating with the RDBMS directly. [3, 4, 28, 37, 44, 53]

Indirect attacks, on the other hand, are performed through web applications. True, it is possible to execute commands by embedding calls to stored procedures in dynamic SQL and that may cause devastating results if successful [2, 3, 13, 14, 37, 53, 63]. However, the main purpose is to directly attack the RDBMS in general and its stored data in particular [2, 5, 8, 12, 13, 14, 15, 21, 24, 27, 28, 32, 36, 38, 39, 42, 43, 52, 55, 60, 62, 63, 65].

A majority of the authors do not mention or discuss direct attacks. This may stem from the fact that they either are not aware of such flaws or that they do not consider them as falling into the scope of SQL injection. Regardless of the reason, direct attacks are conducted through the RDBMS and aims at the network infrastructure. When discussing direct attacks, authors refer to direct communication with the RDBMS and not attacks on the RDBMS itself. It seems to be true that attackers can take advantage of some aspects of SQL injection when performing such attacks. However, we consider the concept of direct attacks to be somewhat misleading since it does not relate to web applications. Furthermore, from a security perspective, we think that direct attacks relate to network security rather than application security. Flaws like open ports that allow attackers to communicate with the RDBMS using arbitrary protocols from command consoles can be prevented by existing network security countermeasures as well as database security configuration, e.g securing the system administrator account. Therefore, we consider direct attacks to be outside the scope of this thesis.

5.1.2 Basics

As noted by Spett [63], a web application can, from a hackers perspective, be viewed as consisting of the following layers: desktop layer, transport layer, access layer, network layer and application layer. At the desktop layer, computers with web browsers acting as clients are used for accessing a system. The transport layer represents the web, and the access layer constitutes the entrance point into a corporation's internal system from the web. The network layer consists of the corporation's internal network infrastructure and finally, the application layer includes web servers, application servers, application logic and data storage. Every layer may have its own implemented countermeasures in order to detect, prevent and recover from attacks, as shown in figure 3.

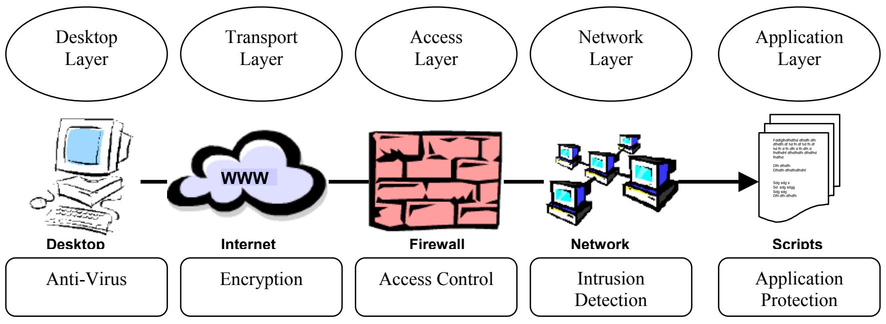

Figure 3: Security layers in web applications [62]

Unfortunately, SQL injection attacks can only be prevented in application logic components such as scripts and programs in the application layer. No matter how many resources and how much effort a corporation spends in the other layers, if application security has not been properly applied in the application layer, their web applications may contain vulnerabilities that SQL injection attackers can exploit [29, 49, 62]. We do not say that countermeasures like encryption, firewalls, intrusion detection and database security are not important. They are effective when dealing with other types of attacks. However, they have been shown insufficient and ineffective regarding SQL injection and therefore we will not consider them [16, 18, 29, 22, 43, 51, 58, 62]. Encryption for example, only protects stored data or data during transport in and between lower layers. In the context of web applications, user input may be encrypted between the client-side and server-side. Furthermore, SQL queries may be encrypted during transport between components such as scripts and programs on the server-side. But in order for the server-side to construct SQL queries and for RDBMS to execute them, they must first be decrypted. The data may still be encrypted but the SQL queries could have been manipulated through SQL injection.

Basically, most web servers are protected by firewalls. However, from a security perspective, web applications offer users legitimate channels through firewalls into corporations systems. The reason for this is that when clients request services from servers on the web, the underlying communication takes place through HTTP, and web applications are no exceptions. HTTP is firewall-friendly, i.e. it is one of the few protocols most firewalls allow through. This stems from the fact that HTTP requests are considered legal, since traffic between clients and servers must be allowed in order for the web applications to be of any use. SQL injection takes advantage of this property by embedding attacks in HTTP requests. These attacks are therefore carried out behind firewalls through the application layer [23, 53, 49, 50, 62]. Therefore, SQL injection requires neither specialized tools nor extensive experience and knowledge. A web browser is sufficient in order to perform SQL injection attacks against web applications, as long as the attacker has basic knowledge of HTTP, relational databases and SQL [12]. Even if the RDBMS is secured through proper configuration, the database can still be vulnerable for SQL injection attacks. In RDBMS, SQL queries are executed as long as they are valid and well-formed and users have the required privileges. Therefore, while security flaws are often to be found in the RDBMS due to improper security configuration, Andrews [2] and Peikari and Fogie [44] and Liu [33] conclude that one must instead consider that database security as a single measure is not sufficient to guarantee protection of data in web applications.

5.1.3 Attack Procedure

We have found that attackers in general follow a procedure consisting of a series of steps. Our compiled procedure represent the set of all steps identified with respect to all available attack methods, further discussed in section 5.2.3. Attackers may combine methods in the attack in order to fulfill their objectives, but the process is executed for each attack method in an iterative manner. Depending on the attack method used, some steps may be ignored.

Setting the objective: Whether explicit or arbitrary, attackers has one or more objectives for conducting SQL injection attacks. These objectives relate to security services as shown in figure 2 on page 37. A concrete example might be that an attacker wants to access the web application in order to obtain information about a corporation's customers. This is an attack on the security service confidentiality.

Choosing the method: In some cases, the attacker is only interested in gaining access to the web application and therefore tries to bypass authentication. In other cases, bypassing authentication is only one step before he can try to reach his objectives. Hence, several methods can be chosen.

Examining prerequisites: In order to determine if the objectives can be reached, the attacker systematically checks which prerequisites is supported. Prerequisites may be necessary conditions for a given attack method, or make the attack easier to conduct.

Testing for vulnerabilities: The attacker begins testing for vulnerabilities to exploit, e.g. experimenting with input validation by entering single quotes, enumerating privileges or evaluating returned information.

Choosing means: Depending on supported prerequisites and found vulnerabilities, the attacker chooses his means for the attack.

Designing the query: The query designed by the attacker needs to follow the proper structure of an SQL query expected by the RDBMS. If not, syntax errors are generated and displayed in error messages. One example of syntax errors relates to quotation marks, i.e. if SQL injection is possible without escaping them. Another example is if parenthesis are used in the underlying query. Depending on the objective, other syntax errors that concern information retrieval of database structure may have to be overridden. Examples of such errors include table names, column names, number of columns and data types.

5.2 Nomenclature

This section presents our classification of the area of SQL injection in terms of general criteria. Surveying SQL injection from a general perspective allows us to discuss the problem irrespectively of factors such as which

RDBMS or application logic components chosen. Therefore, we choose to divide SQL injection issues into broad groups of interest, where every group represent some aspect of the area and contain a set of related items to consider. The order of the items in each group is arbitrary and we do not attempt to rank them.

We have listed which authors that discuss which items within some groups. This enables us to evaluate whether our two main objectives, stated in section 1.6 on page 10, have been reached. Furthermore, we can test whether our hypothesis, also stated in section 1.6 on page 10 is true: do existing prevention techniques contain weaknesses and therefore can not effectively cope with SQL injection?

While our classification can be used as a consistent terminology when discussing SQL injection, we also use it as general criteria for the construction of our security model, presented in section 5.3.

5.2.1 Security Services

The objectives of SQL injection attacks can be expressed in terms of compromising security services discussed in section 4.2 on page 35. We consider the security services below relevant to maintain asset security in respect to SQL injection. We have taken the liberty of altering their definitions slightly:

s1. Access control: Access control involves ensuring that users can only access and manipulate data according to their privileges.

s2. Availability: The services offered by a web application must be available to users when they request them.

s3. Authenticity: Ensuring that users who log in to a web application are who they claim to be.

s4. Confidentiality: Ensuring that information is kept secretly. This security service can be divided into privacy and secrecy.

s4.1. Privacy: Personal information, concerning employees and customers must be kept secret.

s4.2. Secrecy: Sensitive business-related information must be kept secret.

s5. Integrity: Information consistency must be maintained.

5.2.2 Means

Attackers can use different means to perform an attack:

m1. Web page form manipulation: An attacker can use forms to enter parts of SQL statements such as SQL keywords, control characters or data in order to manipulate underlying application server-side scripts or programs.

m2. URL header manipulation: In a similar way as m1, parts of SQL queries can be entered into a page's URL, sending manipulated arguments to the server-side.

m3. Cross-site scripting: By viewing the source code of web pages and examining existing client-side scripts, an attacker can write a fabricated script and use it to send information to the underlying server-side scripts and programs instead of using the original script [62].

m4. Error message interpretation: By examining error messages generated by either the RDBMS or server-side scripts, an attacker can retrieve information about the database structure: table and column names, number of columns and column data types. Furthermore, error messages can contain information about SQL query syntax and how scripts are formed.

5.2.3 Attack Methods

References made in this section refer to which authors that explicitly mention them as attack methods.

There are numerous ways to conduct SQL injection attacks, and the chosen methods depend on what the attacker will accomplish, i.e. which security services to endanger, and what vulnerabilities the web application contains. These methods constitute threats, as discussed in section 4.3 on page 4.3 and they can be grouped into two main categories:

a1. Data manipulation: Using data manipulation, an attacker can bypass authentication as well as retrieve, change, fabricate or delete data in a database.

a2. Command execution: Command execution is a method that enables an attacker to execute SQL specific system commands through the RDBMS and may even allow the attacker to take control over other host computers in the network. This method also takes advantage of static SQL from within dynamic SQL, i.e. tampering with and terminating an existing SQL query and adding a new statement that calls a stored procedure. In case the attacker is calling system procedures that come with the RDBMS, he may for example copy and email database tables to a foreign account. The other approach is to try to call stored procedures that have been tailored by system developers, database administrators or application programmers for the web application. [2, 3, 4, 5, 8, 13, 14, 21, 37, 44, 53, 55, 63, 65].

Category **a1** can further be roughly divided into distinct methods:

a1.1. Authentication bypass: An attacker may use this method to pretend to be a legitimate user [3, 5, 14, 21, 24, 27, 28, 31, 32, 34, 36, 37, 39, 42, 44, 43, 53, 55, 63, 65].

a1.2. Information retrieval: Attackers can try to manipulate or execute SELECT statements in order to get access to information beyond their privileges. This could be achieved by e.g. manipulating the WHERE clause. One example of this is that more rows than intended can be retrieved from the table specified in the original query using the example given in section 5.4. Another example is by using UNION, causing rows from more tables to be returned than specified in the original query [4, 5, 8, 12, 13, 14, 21, 28, 31, 34, 42, 43, 44, 55, 60, 63].

a1.3. Information manipulation: Attackers can try to manipulate or execute UPDATE statements in order to alter information beyond their privileges [3, 12, 28, 31, 44, 60].

a1.4. Information fabrication: Attackers can try to manipulate or execute INSERT statements in order to alter information beyond their privileges. [8, 28, 31, 36, 37, 60, 63]

a1.5. Information deletion: Attackers can try to manipulate or execute DELETE or DROP statements in order to alter information beyond their privileges [3, 5, 13, 27, 36, 43, 44, 60].

Few authors mention or discuss methods a1.3 through a1.5 and among those not referred to in the list above, few supply concrete examples. Rather, these methods are implicitly included in authors documentation. We think that one reason for this might be that their papers and reports are not intended to fully cover all available methods. Nevertheless, our test system mentioned in section 1.8, has proven that the methods a1.3 through a1.5 are valid SQL injection attack methods.

We point out that all means, discussed in section 5.2.2, can be used by methods falling into both category a1 and a2.

5.2.4 Prerequisites

We have found that different SQL injection attack methods need different prerequisites in order to be carried out. These prerequisites are related to both query execution properties and other features that RDBMS support as well as properties of programming languages used for implementing scripts and programs. What is important here is not which properties are offered by which RDBMS and programming languages. Rather, the question concerns which properties are supported by the RDBMS and programming languages chosen in a given web application. These prerequisites are not necessary to conduct SQL injection attacks. Rather, they should be viewed as components that make attacks easier to conduct. For example, prerequisite p4 may enable an attacker to add an INSERT query after an intended SELECT query. However, the attacker could also try to find a field in a form where an INSERT query is expected.

References made in this section refer to which authors that explicitly mention them as prerequisites.

p1. **Sub-selects:** Sub-selects are multiple SELECT statements used together. A top-level SELECT statement is using other lower-level statements to retrieve values to be used in a WHERE clause [14, 63].

p2. **JOIN clause:** JOIN clause can be used when multiple SELECT queries are combined in the same query.

p3. **UNION clause:** UNION clause can be used when multiple SELECT queries are combined in the same query [3, 14, 28, 31, 36, 37, 42, 63, 65].

p4. **Multiple statements:** Refers to the ability to allow execution of multiple SQL statements, where each statement is separated by a delimiter, e.g. a semicolon [3, 14, 21, 27, 34, 44].

p5. **End-of-line comments:** the ability to comment out parts of a SQL statement, meaning that the RDBMS will not take notice of the SQL syntax followed by a comment symbol. For example, some RDBMSs uses '–' as a comment symbol. symbol [2, 3, 21, 27, 28, 31, 32, 36, 37, 39, 55, 60, 63, 65].

p6. **Privileged accounts:** Accounts defined in the database are used by database connections to access the database. An attacker could only use attack methods that execute SQL statements associated with defined privileges in the account used by the web application. For example, if the account does not specify DELETE as a privilege, the attacker can not use **m4** as attack method.[36, 37]

p7. **Error messages:** Errors that occur in the RDBMS or in any server-side script or program can produce an error message that can be sent to the client and printed in the web browser. [3, 21, 15, 28, 31, 36, 37, 39, 55, 63].

p8. Weak data types: Several script and programming languages used in web application development support variables of weak type[9], i.e. variables that can store data of arbitrary type. [2, 12, 13, 34]

p9. Data type conversion: Several RDBMSs support variable type conversion, e.g. allowing numeric values to be converted automatically into a string type. [28]

p10. Stored procedures: Such procedures, supported by some RDBMSs, allow execution of system or database commands and SQL sub-routines in the RDBMS [3, 13, 14, 15, 21, 31, 36, 37, 44, 63].

p11. Dynamic SQL: In order to embed user input into SQL queries, server-side scripts and programs can use dynamically built SQL queries where SQL statements are combined with user input and then sent into the RDBMS for execution. Another approach is to let dynamically built SQL queries call stored procedures. [3, 13, 14, 21, 24, 27, 31, 32, 43, 52, 55, 63, 65]

p12. INTO OUTFILE support: If INTO OUTFILE is supported by the RDBMS, users may print query results into a text file on the host computer [12].

5.2.5 Vulnerabilities

In section 4.5 on page 37 we presented vulnerabilities. In this section, we present vulnerabilities that might be inherent in web applications and that can be exploited by SQL injection attacks. References made in this section refer to which authors that explicitly mention them as vulnerabilities.

v1. Invalidated input: Unchecked parameters to SQL queries that are dynamically built can be used in SQL injection attacks. These parameters may contain SQL keywords, e.g. INSERT or SQL control characters such as quotation marks and semicolons. [2, 3, 5, 12, 13, 27, 28, 31, 32, 36, 37, 39, 42, 43, 44, 53, 55, 60, 63, 65]

v2. Error message feedback: Error messages that are generated by the RDBMS, as defined in section 3.4, 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. [3, 8, 15, 21, 27, 37, 39, 43, 55]

[9] Also referred to by authors as loose data types, soft variables or variables that have no clear type.

v3. 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 for other attacks, such as *buffer overrun* [3, 36, 43, 53, 63, 65].

v4. Variable morphism: If a variable can contain any data, it is possible for an attacker to store other data than expected. Such variables are either of weak type, e.g. variables in PHP, or are automatically converted from one type to another by the RDBMS, e.g. numeric values converted into a string type. For example, SQL keywords can be stored in a variable that should contain numeric. values. [2, 12, 13, 28, 31, 34, 55]

v5. 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 are 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 a database connections using a specific account for accessing the database. An attacker who bypasses authentication gains privileges equal to the account's. 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. [2, 8, 12, 13, 14, 27, 28, 36, 44, 52, 55]

v6. Dynamic SQL: As defined in section 3.3.1 on page 30, dynamic SQL refers to 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 e.g. WHERE clauses. The problem with this approach is that query building components can also receive SQL keywords and control characters, creating a completely different query than the intended [2, 4, 14, 24, 26, 32, 34, 43, 52, 65].

v7. Stored procedures: As discussed in section 3.3.2 on page 30, stored procedures are statements stored in RDBMSs. The main problem using these procedures is that an attacker may be able to execute them, causing damage to the RDBMS as well as the operating system and even other network components. Another risk is that stored procedures may be subject to buffer overrun attacks. System stored procedures that comes with different RDBMS are well-known by attackers and fairly easy to execute. [3, 4, 12, 21, 28, 36, 44, 53, 63, 65]

v8. 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. [37, 39, 50, 60, 62]

v9. 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 information in it in order to e.g. bypass authentication. [12]

v10. Sub-selects: If the 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. [14, 12]

v11. JOIN/UNION: If the RDBMS supports JOIN or UNION, the variations of attack methods used by an SQL injection attacker increases. For example, an original SELECT class can be modified with a JOIN SELECT or UNION SELECT clause. [14, 12]

v12. Multiple statements: If the RDBMS supports JOIN or UNION, the variations of attack methods used by an SQL injection attacker increases. For example, 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 himself to the table of users. [12, 14, 21, 34]

5.2.6 Countermeasures

In this section, we present technically oriented countermeasures found during the part of our survey that concerns prevention techniques against SQL injection attacks. References made in this section refer to which authors that explicitly mention them as countermeasures.

c1. Different accounts: Default accounts that come with some RDBMS, such as the account used by the system administrator, should never be used for web application access. Instead, different accounts should be created and used for different client profiles. Moreover, in case that the RDBMS chosen has a default system administrator password, change it immediately after installation. [2, 3, 13, 21, 26, 36, 37, 43, 44].

c2. Limited privileges: Permissions granted to database accounts, used by web applications, should be given according to the principle of least privilege. The actions made by web applications users on stored procedures should be limited too, i.e. if the intention is to not use or have unused stored procedures they should be removed or moved to an isolated server. This minimizes the damage an attacker can cause in case of authentication bypass. Normally, SELECT is a privilege that in almost any case will be given to an account. This is used for logging in to the system and retrieve information. But when logging in, a user should not have any other privileges such as INSERT or DELETE [2, 3, 4, 13, 14, 15, 21, 28, 36, 43, 44, 52, 55, 63, 65].

c3. Static SQL: As defined in section 3.3.2 on page 30, static SQL refers to SQL statements that can not be altered by inserting SQL keywords where values are expected. Examples of static SQL include taking advantage of *prepared statements* and the use of stored procedures. This countermeasure implies that static SQL should be used in favor of dynamic SQL, which should be avoided. [2, 3, 8, 15, 24, 26, 32, 34, 36, 52, 65]

c4. Error handling: Error messages generated by RDBMSs or web servers should never be passed back to the client-side since they may contain information about database and script structure. Instead, handle error messages at the server-side and send back messages to the client-side that do not contain information that could be used for SQL injection attacks. [2, 3, 8, 43]

c5. Input validation: All data sent by the client-side should be considered potentially harmful. While input validation could be implemented in client-side scripts for performance factors, one should never rely on client-side scripts for security. Therefore, every parameter sent from the client-side should always be examined and validated by server-side scripts and programs. This could be done by for example using comparison and replacing functions if the development platform enables it. Developers can otherwise write such functions themselves. One can also take advantage of regular expressions. If concatenation is necessary, then use numeric values for the concatenation part or check the input for malicious character strings and sequences, e.g. SQL keywords, such as UNION, or meta characters and SQL control characters such as single and double quotation marks and semicolons. [2, 3, 5, 12, 13, 15, 21, 24, 26, 28, 31, 32, 34, 36, 37, 39, 42, 43, 44, 52, 53, 55, 60, 63, 65]

c6. Character escaping: In case characters such as quotation marks or semicolons must be allowed, for example if arbitrary text should be accepted, those characters should be escaped using a scheme, e.g. ASCII code or using bind variables. [3, 12, 21, 24, 26, 31, 32, 36, 42, 53, 55, 60, 63]

c7. Stored procedure limitation: Limit the use of system stored procedures that follow with an RDBMS. Stored procedures that are not intended to be used should be removed or made inaccessible. [15, 26, 28, 44]

c8. Variable size: Control variable length and size [3, 4, 21, 26, 36, 43, 53, 65].

c9. Strong typing: try to avoid weak variables that have either no clear type, e.g. variables in PHP, or are automatically converted from one type to another by components such as the RDBMS. Instead, make sure that variables are explicitly typed. In case variables of weak type need to be used, check their content. [2, 12, 13, 15, 28, 31, 34, 55]

5.3 SQL Injection Security Model

One objective given in section 1.5.1 on page 8 was to create a general security model for SQL injection. We have compiled how different components, expressed as general criteria and defined in previous sections, relate to each other into such a model, shown in tables 2, 3 and 4. In this model, we start by choosing an attack method and define at which security services it aims. Then, we list the prerequisites needed to commit attacks using that method, continuing with presenting the vulnerabilities needed. Finally, countermeasures used for protecting web applications against the chosen method are presented.

Most of the lists found under services, prerequisites, vulnerabilities and countermeasures contain several items. Not all items contained in such lists are always necessary conditions. Our model merely represent an attackers tool box when planning and committing SQL injection attacks. Further relationships between items in the list will be presented in sections 5.4 section 6.

We will use this model in the process of evaluating existing prevention techniques, covered in section 6.

5.4 SQL Injection Attack Examples

We have studied various attack examples as well as tested them in our system, mentioned in section 1.8 on page 12. Our intention is not to describe every type of attack method and their variations, since they can be studied in our referenced research material. However, for issues of clarity, we present examples that both give information on how SQL injection attacks could be performed and how we processed attack examples during the creation of our model, shown in tables 2, 3 and 4.

Attacks on the database can be made in order to gain access to the web application, and thus threatening the application's authentication security

Attacks:	a1.1. Authentication bypass	a1.2. Information retrieval
Services:	s3. Authenticity	s4. Confidentiality
Prerequisites:	p5. End-of-line comments p7. Error messages p8. Weak data types p9. Data type conversion p11. Dynamic SQL p12. INTO OUTFILE support	p1. Sub-selects p2. JOIN clause p3. UNION clause p4. Multiple statements p5. End-of-line comments p6. Privileged accounts p7. Error messages p8. Weak data types p9. Data type conversion p10. Stored procedures p11. Dynamic SQL p12. INTO OUTFILE support
Vulnerabilities:	v1. Invalidated input v2. Error message feedback v3. Uncontrolled variable size v4. Variable morphism v6. Dynamic SQL v8. Client-side-only control v9. INTO OUTFILE support	v1. Invalidated input v2. Error message feedback v3. Uncontrolled variable size v4. Variable morphism v5. Generous privileges v6. Dynamic SQL v7. Stored procedures v8. Client-side-only control v9. INTO OUTFILE support v10. Sub-selects v11. JOIN/UNION v12. Multiple statements
Countermeasures:	c3. Static SQL c4. Error handling c5. Input validation c8. Variable size c9. Variable morphism	c1. Different accounts c2. Limited privileges c3. Static SQL c4. Error handling c5. Input validation c6. Character escaping c7. Stored procedure limitation c8. Variable size c9. Variable morphism

Table 2: SQL injection security model, attack methods a1.1 and a1.2

Attacks:	a1.3 Information manipulation	a1.4 Information fabrication
Services:	s1. Access control s5. Integrity	s1. Access control s5. Integrity
Prerequisites:	p4. Multiple statements p5. End-of-line comments p6. Privileged accounts p7. Error messages p8. Weak data types p9. Data type conversion p10. Stored procedures p11. Dynamic SQL	p4. Multiple statements p5. End-of-line comments p6. Privileged accounts p7. Error messages p8. Weak data types p9. Data type conversion p10. Stored procedures p11. Dynamic SQL
Vulnerabilities:	v1. Invalidated input v2. Error message feedback v3. Uncontrolled variable size v4. Variable morphism v5. Generous privileges v6. Dynamic SQL v7. Stored procedures v8. Client-side-only control v12. Multiple statements	v1. Invalidated input v2. Error message feedback v3. Uncontrolled variable size v4. Variable morphism v5. Generous privileges v6. Dynamic SQL v7. Stored procedures v8. Client-side-only control v12. Multiple statements
Countermeasures:	c1. Different accounts c2. Limited privileges c3. Static SQL c4. Error handling c5. Input validation c6. Character escaping c7. Stored procedure limitation c8. Variable size c9. Variable morphism	c1. Different accounts c2. Limited privileges c3. Static SQL c4. Error handling c5. Input validation c6. Character escaping c7. Stored procedure limitation c8. Variable size c9. Variable morphism

Table 3: SQL injection security model, attack methods a1.3 and a1.4

Attacks:	a1.5. Information deletion	a2. Command execution
Services:	s2. Availability s5. Integrity	s1. Access control s2. Availability s3. Authenticity s4. Confidentiality s5. Integrity
Prerequisites:	p4. Multiple statements p5. End-of-line comments p6. Privileged accounts p7. Error messages p8. Weak data types p9. Data type conversion p10. Stored procedures p11. Dynamic SQL	p4. Multiple statements p5. End-of-line comments p6. Privileged accounts p7. Error messages p8. Weak data types p10. Stored procedures p11. Dynamic SQL
Vulnerabilities:	v1. Invalidated input v2. Error message feedback v3. Uncontrolled variable size v4. Variable morphism v5. Generous privileges v6. Dynamic SQL v7. Stored procedures v8. Client-side-only control v12. Multiple statements	v1. Invalidated input v2. Error message feedback v3. Uncontrolled variable size v4. Variable morphism v5. Generous privileges v6. Dynamic SQL v7. Stored procedures v8. Client-side-only control v12. Multiple statements
Countermeasures:	c1. Different accounts c2. Limited privileges c3. Static SQL c4. Error handling c5. Input validation c6. Character escaping c7. Stored procedure limitation c8. Variable size c9. Variable morphism	c1. Different accounts c2. Limited privileges c3. Static SQL c4. Error handling c5. Input validation c6. Character escaping c7. Stored procedure limitation c8. Variable size c9. Variable morphism

Table 4: SQL injection security model, attack methods a1.5 and a2

service (s3)[10]. Suppose that an attacker would like to try and log in into a system that uses a web interface which has not been protected from SQL injection attacks. This might be the first step an attacker takes in order to be able to commit further attacks on an application. In order to do this, an attacker might try to analyze how a login page, like the one in figure 10 below might be exploited.

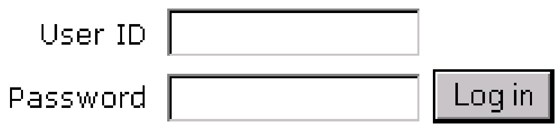

Figure 4: A login form

In order for that to work, a few prerequisites are needed, and some related vulnerabilities should be left non-handled. To begin with, the application must support dynamic SQL queries (p11 and v6), and the application should not validate the user's input (v1) or control the input's type and size (p8, v3 and v4). Furthermore, if it also displays server-side-generated error messages in the user's web browser (p7 and v2), it would give helpful feedback. Finally, it would be useful to the attacker if end-of-line comments are supported (p5). The attacker may begin the attack by entering some chosen symbols and SQL keywords in the login page fields and analyzing the error messages. Following Anley's [3] example, suppose that the attacker prints the following in the login field:

```
Username: ' having 1=1--
```

The error message returned might be:

```
Microsoft OLE DB Provider for ODBC Drivers error '840e14'

[Microsoft][ODBC SQL Server Driver][SQL Server]Column
'users.id' is invalid in the select list because it is
not contained in an aggregate function and there is no
GROUP BY clause.

/process_login.asp, line 35
```

This error message contains information about the SQL query: it concerns a table named `users` with a column named `id`. The attacker continues investigating the query by writing:

[10]We list in parenthesis the relevant attack method, security services, means, prerequisites, vulnerabilities or countermeasures by referring to the symbols denoted in section 5.2

```
Username: ' group by users.id having 1=1--
```

Which results in the error message:

```
Microsoft OLE DB Provider for ODBC Drivers error '840e14'

[Microsoft][ODBC SQL Server Driver][SQL Server]Column
'users.username' is invalid in the select list because
it is not contained in either an aggregate function or
the GROUP BY clause.

/process_login.asp, line 35
```

So now the attacker knows that the table also contains the column name `username`. The attacker may continue to acquire information about the query (the exact technique is explained Anley's article) until he recovers the entire syntax used by the script to build the dynamic SQL query:

```
"SELECT * FROM users WHERE username = '" + username + "'
 AND password = '" + password + "'"
```

The attacker could now bypass the authentication control by entering the following fragment into the username field:

```
Username: ' OR 1=1--
```

Upon concatenating that input into the SQL query, the following query will be send to the database:

```
SELECT * FROM users WHERE username = '' OR 1=1--
(the rest of the statement is ignored)
```

The query is sent to a SQL Server database, which returns the rows that match the query. The script checks whether any rows were returned. If so, it returns true, indicating that the user is identified. Otherwise, the script returns false meaning no such user is registered in the database. The system considers the user authenticated or not based on the return value received from the script or program that requested the execution of the query, and allows the user to use it if confirmed.

Since the conditions in the WHERE clause will be evaluated to true, the query will return a result containing all users in the database. As a consequence, the script will also return true, and the query yields a valid user who's identity the attacker would assume while using the system. The attacker will have the privileges specified in the account used for accessing the RDBMS. Different accounts may have been defined in the RDBMS and application logic may choose different accounts for different users. In case all rows are returned, they might have been sorted by the RDBMS according to id, name or some other criteria. The application logic may determine which user is logging in due to the result set by looking at the first row. The system administrator may be the first user defined in the table of users, hence having the lowest id. Therefore, if the application logic chooses the first row, it is likely that the attacker will log in as the system administrator and normally, that account will be given all privileges [21, 31, 55].

Suppose that the attacker would now want to continue the attack, but instead of bypassing authentication control, he would attempt to insert a fabricated user identity into the database table, which he could later use for further attacks. The attacks would thus attempt to fabricate information (a1.4) and would endanger the application's access control security service (s1) as well as its integrity (s5). The attacker would now need a way to manipulate the application into entering the user identity into the `users` table in the database. This could be done in a few ways: the attacker could try to call a stored procedure (if such exists in the RDBMS used by the application) that would enter a row into the `users` table (a2). A second way would be to try and find a field in the application that is used for entering information into the database, and tamper with it (a1.4). A third way would be to manipulate the above mentioned login field query and add to it an INSERT statement (a1.4). This would be possible if the RDBMS supports multiple statements (p4). Regardless of the approach taken, the account chosen by the web application when the attacker circumvented authentication would need to have the corresponding privileges. For example, either one of the latter two choices world require the privilege INSERT on the table `users`. This could be expressed in terms of privileged accounts p6 and v5.

All the attacker needs to do is to find out exactly which fields the `users` table contains and which types each column requires (as explained above and in Anley's article), and then enter the manipulating data into the login field. That data could look like the example below:

```
'; INSERT INTO users(username, password)
  VALUES('hacker', '666');
```

This would yield the following query:

```
SELECT * FROM users WHERE username = '';
INSERT INTO users(username, password)
VALUES('hacker', '666');
```

Once the manipulated query executes in the database, it would result in a new row in the `users` table containing the attacker's fabricated identity.

6 Model Analysis

In this section, we comment some attributes of the model covered in section 5: we reflect on some important aspects of the model, explain our view on how it should be used, compare the countermeasures in our model with other authors' suggestions, and propose an outline for how the model should be implemented.

6.1 Aspects of the Model

We discuss in this section those aspects of the model that we consider central for our results.

6.1.1 Static vs. Dynamic SQL

Some authors suggest that the usage of dynamic SQL in web applications is one of the major reasons for that successful SQL injection attacks can be committed [2, 4, 14, 24, 26, 34, 43, 65]. The usage of static SQL may dramatically reduce that risk, since the original SQL query may not be changed, as explained in section 3.3.2 on page 30. Other authors point out, that some applications could not be implemented solely with static SQL, since it would make the implementation too complex [65]. We concur that static SQL is a very effective countermeasure, but not fail-proof [2, 3, 8, 15, 24, 26, 32, 34, 36, 52, 65]. Defending an application against SQL injection attacks using a single countermeasure might be dangerous, due to the following factors:

- SQL injection attacks can still be committed, according to Anley [4], though the application only uses static SQL. That could occur when for example the `exec` stored procedure is used in SQL server.
- Even if the risk of SQL injection is not so large, other attack techniques, such as buffer overrun, may still exploit the same vulnerabilities to perform an attack.
- Using static SQL, and especially stored procedures, could enable an attacker to perform harmful attacks on the application and even the host or network, as explained in section 5.2.5 on page 46. Naturally, execution of such stored procedures through web applications requires that dynamic SQL is used, though one should still be careful and only allow the most necessary stored procedures to be available, and keep user access to them according to least-privilege principle.

6.1.2 RDBMS Support

Some prerequisites in our model are only supported by some RDBMSs and not by others. For example, stored procedures are supported by MS SQL

Server and Oracle RDBMSs, but not by MySQL[11]. Among others, SQL injection-related differences may include variations in support of sub-selects, multiple statements, end-of-line comments, UNION and JOIN clauses, INTO OUTFILE clause and default administrator accounts [12]. These prerequisites are related to the following vulnerabilities:

- Sub-selects, UNION and JOIN clauses, and multiple statements may allow execution of concatenated queries in the RDBMS.
- End-of-line comments may allow manipulation of the query.
- The INTO OUTFILE clause may allow attackers to output database information into a file that might be available in some way, exposing personal or corporation-sensitive information.
- Default administrator accounts might carelessly be used and give users privileges beyond their need. If such accounts are used as web application accounts, an attacker who succeeds in bypassing authentication may cause more substantial damage.

We would like to stress that the choice of RDBMS, made when a web application is to be developed, should among other factors even consider these prerequisites and related vulnerabilities. For example, if the Oracle RDBMS is used, one must take into consideration that a few countermeasures must be taken, such as stored procedures and privilege limitation. On the other hand, the multiple statements vulnerability does not concern Oracle RDBMS users, since it is not supported. We would further like to suggest that our model could be used as a base for decision-making when choosing an RDBMS to be used in a web application.

6.1.3 Input Validation

Many authors consider input validation to be a vital countermeasure against SQL injection [2, 3, 5, 12, 13, 15, 21, 24, 26, 28, 31, 32, 34, 36, 37, 39, 42, 43, 44, 53, 55, 60, 63, 65]. This, in combination with related countermeasures, such as character escaping and variable type and size control, can be used to control user input and allow only valid data to be used. There are two approaches concerning input validation. The first one is to actively search for unwanted SQL fragments or symbols in the user input, and to replace them with legitimate characters or symbols. The other approach is to define which symbols are allowed and throw away all input that is different. The problem with the first approach is that new symbol or character combinations which can be used for attacks may be found in the future, causing the application

[11]We refer here to the list of specific RDBMS supported prerequisites made by Eizner [12], and we have not researched in depth whether new versions of these RDBMSs supports other prerequisites.

to be vulnerable. The problem with the second approach is that it is difficult to know in advance which characters or symbols will be used throughout the application lifetime.

We consider input validation to be an important countermeasure, but would like to emphasize that it should be combined with other countermeasures such as static SQL, privilege limitation, error handling etc. The reason for this is to defend the application in different ways and levels: if an attacker decides to commit an attack on an organization's web application, there should as many attempts to stop him from damaging the application as possible, both in the application's logic components and in the RDBMS.

6.2 How to use the Model

Our intentions in this thesis were originally to review as much SQL injection-related documentation as possible and to summarize it in a way that would allow insight into the problem. We hope that our security model, and the tables presenting it, contribute to that goal. We have collected much material about SQL injection and have tried to present it in a way that would help understanding it. Since our discussion concerns SQL injection attack methods and related components, we decided that the best way to present the information was by starting out at the attacks, allowing us to think the way an attacker might do. Then, we tried mapping other related components to the attack, thus listing what the attacker is trying to gain, which prerequisites must exist for the attack to succeed, which vulnerabilities are being exploited, and finally, what can be done to prevent them. We hope that anyone who is interested in an overview of SQL injection could find it here.

By listing all model components together with a reference to the authors who describe them, and particulary by presenting which authors recommend different countermeasures, as listed in 6.4, we hope to help anyone who is interested in further information research to find it easily. Since different authors concentrated on different aspects of SQL injection, our model could also be used as a tool for evaluation of related articles. We have, during our research, encountered numerous authors who claim that SQL injection is a problem which one could easily solve, using static SQL or input validation as countermeasures. We believe that, as Anley claims, "It is extremely dangerous to place your faith in a single defence" [4]. In order to make an application safe from SQL injection attacks, we recommend that all countermeasures should be considered and applied.

We would also like to consider this model a decision-making help tool which, for example, could be used when choosing between RDBMSs. Thus, one could use the model to see which prerequisites and vulnerabilities one should be aware of, as explained in section 6.1.2. Furthermore, the model could be of help when conducting a risk analysis, or when evaluating a web application for how well it is protected against SQL injection attacks. Ultimately, we suggest that this model could be used as a guide for web application designers as an aid for design-related decision making during the process.

Finally, we would like to suggest this model as an example for how security problems could be surveyed. We believe that by collecting much information about a specific area such as SQL injection, and by grouping and mapping its components, we achieved a better understanding of the problem and presented it in a way that would help dealing with it. We believe that other issues, whether more complex or at the same level of complexity, should also be surveyed in a comprehensive way, and we suggest that one could follow our example.

6.3 Aspects of Existing Prevention Techniques

In section 1.6 on page 10 we listed a set of aspects and hoped that a general SQL injection security model could help us to determine if those aspects are taken into account by the contributors of existing prevention techniques. We will now try to test whether our model complies to that objective by giving one example, where we examine one of the existing techniques. We choose the technique given by Spett [62, 63] and start with extracting items in groups, given in section 5.2, that the author has taken into account[12].

Spett covers attack methods authentication bypass, information retrieval, information fabrication and command execution (a1.1, a1.2, a1.4 and a2)[13]. Furthermore, prerequisites sub-selects, UNION clause, end-of-line comments, error-messages, stored procedures and dynamic SQL is covered (p1, p3, p5, p7, p10 and p11). Additionally, vulnerabilities invalidated input, uncontrolled variable size, stored procedures and client-side-only control (v1, v3, v7 and v8) are mentioned. Finally, Spett discusses countermeasures limited privileges, input validation and character escaping (c2, c5 and c6).

By using our security model, we find out that the attack methods covered by Spett together may compromise security services s1, s2, s3, s4 and s5. This indicates that the contributor has tried to implement all our defined security services. However, this prevention technique does not conform to our general criteria, since all attack methods, prerequisites, vulnerabilities and countermeasures are not covered. Moreover, since all vulnerabilities are not covered, the technique has left out some vulnerabilities that SQL injection attacks may exploit. This technique considers stored procedures as a vulnerability and explicitly mentions uncontrolled variable size as a

[12]We want to point out that our intention has not been to reduce the value of any article. Rather, we intend to use this article as an example where the author's suggested countermeasures might not cover all vulnerabilities

[13]We list in parenthesis the relevant attack methods, security services, means, prerequisites, vulnerabilities or countermeasures by referring to the symbols denoted in section 5.2.

vulnerability too. But no information has been given by Spett that mentions uncontrolled variable size (v3) as a problem of concern when using stored procedures (v7). Furthermore, the countermeasure for v7, variable size (c8), is not given. This indicates that the web application may be subject for buffer overrun attacks in case stored procedures are used, and that other countermeasures are needed. In summary, we conclude that this technique does not overcome all threats associated with SQL injection and that weaknesses have been revealed.

We have come to the conclusion that our model comply with our objectives, since answers to our stated aspects could be given for an example prevention technique. However, this does not necessarily mean that an attempt to reveal all answers for all prevention techniques would be successful.

6.4 Countermeasure Comparison

In section 1.5.1 on page 8, we set out an objective to perform a comparison between required countermeasures and countermeasures taken into account in existing prevention techniques. We have constructed a matrix, shown in table 5, which lists which countermeasures are accounted for in each prevention technique. Instead of naming the prevention techniques, their contributors are given.

In section 1.6 on page 10, we hoped that the matrix would reveal, with respect to proposed countermeasures, weaknesses in existing prevention techniques.

After inspecting the matrix, we might assume that the required countermeasures would constitute all countermeasures stated in our security model and to discard the suggested prevention techniques since none of them live up to that requirement. However, the task of identifying the required countermeasures has not been as easy as we thought. In order to fully understand what conclusions could be drawn from this matrix, we need to discuss underlying assumptions and circumstances inherent in the prevention techniques that we have encountered during our survey.

The intentions with a prevention technique differ among authors. This also holds regarding the scope of SQL injection and the degree of generalization in authors' documents. Some authors have, deliberately or not, chosen to only cover a few aspects of the problem. Examples of this include considering only a subset of the available attack methods, looking more carefully at vulnerabilities and focus less on countermeasures and only cover a few vulnerabilities. Another factor is that some authors have been thorough in there attempts to cover the problem. Others, on the other hand, either reference the work of their predecessors or assume that readers already are experienced with SQL injection, only adding one or more pieces to the puzzle.

Author	**c1. Different Accounts**	**c2. Limited Privileges**	**c3. Static SQL**	**c4. Error Handling**	**c5. Input Validation**	**c6. Character Escaping**	**c7. Stored Procedure**	**c8. Variable Size Limitation**	**c9. Strong Typing**
Andrews [2]	x	x	x	x	x				x
Anley [3, 4]	x	x	x	x	x	x		x	
Atkins [5]					x				
Cerrudo [8]			x	x					
Eizner [12]					x	x			x
Farrow [13]	x	x			x				x
Finnigan [14, 15]		x	x		x		x		x
Harper [21]	x	x			x	x		x	
Jepson [24]			x		x	x			
Kiely [26]	x		x		x	x	x	x	
Kok [28]		x			x		x		x
Litchfield [31]					x	x			x
Litwin[32]			x		x	x			
Macromedia [34]			x		x				x
McDonald [36]	x	x	x		x	x		x	
Meer [37]	x				x				
Memonix & MrJade [39]					x				
Newman [42]					x	x			
Overstreet [43]	x	x		x	x			x	
Peikari and Fogie [44]	x	x			x		x		
Qualls [53]					x	x		x	
Robertson [55]		x			x	x			x
Sonnemans [60]					x	x			
Spett [63]		x			x	x			
Talmage [65]		x	x		x			x	
Yvel [52]		x	x		x				

Table 5: Countermeasure comparison

We have found one concrete example of ambiguous reasoning that concerns vulnerabilities and countermeasures: a majority of the authors contemplate dynamic and static SQL differently in that they either reject dynamic SQL and favor static SQL, or vice versa. Some authors even suggest that both dynamic and static SQL are subjects for vulnerabilities. The standpoint taken here will ultimately have an impact on suggested countermeasures. Those who state that dynamic SQL is a vulnerability suggest static SQL as countermeasure and devote less attention to input validation. The other group conclude that invalidated input is the problem, not dynamic SQL itself, and therefore heavily stress the importance of input validation. Some authors in that group have even stated that static SQL is also a vulnerability. Hence, if the first approach is followed, static SQL is not proposed as countermeasure. Of course, we have a minority of techniques that suggest countermeasures that combine the two approaches. The problem here is to claim who is right. Nevertheless, this clearly indicates that it is difficult to claim a prevention technique that does not propose static SQL as countermeasure to be less effective, if it at the same time mentions input validation.

Another implication is that the majority of authors have been discussing SQL injection in the context of a particular RDBMS. As showed in our security model, different attack methods require different prerequisites. While some of the prerequisites concern programming languages, most of them refer to properties of RDBMS. Therefore, authors may be influenced of the properties in the type of RDBMS used in their discussion when suggesting countermeasures.

Due to implications discussed above, we can not in general give a comprehensive opinion that claims to what extent existing prevention techniques are effective. However, in our method given in section 1.5 on page 8 we state that we are not measuring the effects of SQL injection attack methods and the effectiveness of common prevention techniques empirically. Instead, we theorize about effects and effectiveness. The inconsistencies and weaknesses introduced above and in section 6.3 thus support this survey's purpose.

Furthermore, our matrix shows differences among authors with respect to proposed countermeasures, and that no single contributor has taken every countermeasure stated in our security model into account. In section 6.1.3 on page 58, we stress that if dynamic SQL is prohibited in a web application and replaced with static SQL in conjunction with input validation, an attacker's arsenal will be considerably limited. As a consequence, if prevention techniques do not mention static SQL and input validation as countermeasures, we consider them to be less effective and should be used as complementary techniques. Such techniques are identified in our matrix.

Taken the aspects mentioned above into account, our matrix reveals weaknesses in existing prevention techniques. However, since we could not define the complete set of countermeasures required, the matrix can not give enough information to enable a thorough comparison. Therefore, we fail in achieving the second statement listed under our second objective, given in section 1.6. The matrix merely validates that our security model can serve as a valuable tool for evaluating material of SQL injection, evaluating web applications with respect to SQL injection vulnerability, decision support when choosing web application components, and guidelines during web application development.

7 Epilogue

7.1 Discussion

Our choice of confronting SQL injection from a general perspective enabled us to classify SQL injection and compile all of the aspects we found into a single, coherent security model. We have speculated and theorized how this model can contribute to the problem of SQL injection, as stated in section 1.2 on page 6. By summarizing the aspects involved, concerned readers may easier gain a better understanding of the overall picture of SQL injection. Moreover, we believe this model to be a useful checklist when evaluating to what degree web applications are vulnerable to SQL injection attacks. This model also presents what aspects to look for when reading and evaluating articles about SQL injection. Furthermore, the model gives valuable clues on how other web application security problems could be confronted.

We have also stated that our security model can be used for revealing weaknesses in existing prevention techniques against SQL injection. This has been further supported by the matrix constructed from our model. Though, we could not determine the complete set of countermeasures needed by any given existing prevention technique for a completely accurate comparison.

While an attempt to present new prevention techniques is outside the scope of this thesis, one question arises: does our model and matrix together reveal a combination of countermeasures into a more effective technique for dealing with SQL injection then existing ones? The answer to this question is no, mainly because we could not define the optimal combination of countermeasures to be used in our comparison of existing prevention techniques, with respect to countermeasures, as discussed in section 6.4. Consequently, we have no support for our recommendation made in section 6.2 on page 59. Our second objective, stated in section 1.6 on page 10, could not be completely fulfilled. Nevertheless, the information given in the matrix, combined with our security model and deeper insight of SQL injection indicates that one should be suspicious of material on SQL injection in general and suggested prevention techniques in particular.

The inductive approach taken in this thesis is subject for various problems related to its reliability. The material found regarding SQL injection may not be that exhaustive that we assumed. There might exist material that we have not found, foremost due to incomplete search criteria. Our delimitations may also contribute to reliability problems, especially the choice of not including direct attacks in the area of SQL injection. Another factor is that the aspects and characteristics included in our security model are based upon our own insights and assumptions during the survey.

These implications might very well be subject to at least three sources of errors. First, we might not have extracted all characteristics regarding

SQL injection, such as vulnerabilities and countermeasures. The other thing concerns the risk that we may have missed some aspects regarding characteristics, e.g. there might exist more countermeasures than we have found. Finally, we could have misinterpreted the contributors intentions and results.

However, in our defence we will point out the lack of comprehensive material available for us regarding SQL injection. This, of course, further contributes to our validity problems. Information found in documents and reports have often been gathered and written by authors with several years of technical experience. But the structure and language used in the material presented us with differences in extracting knowledge from the documents.

7.2 Future Work

In this thesis, we have concentrated on the specific area of SQL injection. Though this is a narrow subject, we consider the processes of collecting documentation and analyzing all documentation we have found to be a complex task. We believe that this area is in need of further investigation, mainly because of two reasons: first, we can not be certain that we have compiled a definite list of all components that could be taken into consideration. Secondly, SQL injection attacks are most likely to evolve and new vulnerabilities will be found, together with new countermeasures to deal with them. Since many hacking sites are available on the web, and since attack methods are well described and distributed between hackers, we believe that information about new attack methods should continuously be surveyed and new countermeasures should be developed.

According to OWASP's Ten Most Critical Web Application Security Vulnerabilities [49], many SQL injection-related issues are among the most harmful threats to web applications. Since we have in this thesis only covered SQL injection aspects, we would like to suggest that further studies should be made on other threats to related security issues, especially such that relate to application security. The reason is that several authors have mentioned that organizations spend most security resources on operating system and network level security [29, 49, 62], and not enough on application layer security. If further studies will be made on application layer security issues, and particulary on web application, it would be possible to compile results from all of these into general security guidelines, which could be used in developing more secure web applications.

One of our goals in this thesis was to increase the level of security awareness among organizations regarding web applications, especially towards SQL injection threats. We hope that further surveys in this area and in related web application subjects will help achieving that goal, so that hopefully security standards will be implemented and countermeasures built into applications during development. Ultimately, organizations will use a proactive approach towards application layer security, which will then be an indispensable part of web applications.

7.3 Concluding Remarks

Besides the problems described in section 7.1, we think that our results are promising in that as far as we are concerned, this is the first attempt to analyze SQL injection from a general perspective and to gather many of its characteristics. Our research material constitutes resources that supply readers with more technical details, including particular programming environments, RDBMSs, and information about how to program and configure web application components. In contrast, our security model gives insights into what aspects to look for in such material. Furthermore, for evaluation purposes, our classification also presents which attack methods, prerequisites, vulnerabilities and countermeasures different authors cover. We have used our general criteria and security model to confront one existing technique. We also presented gathered techniques and proposed countermeasures into a matrix. The results of those processes revealed weaknesses: not all prevention techniques proposed by authors can be claimed effective, since different techniques present different countermeasures and the authors' underlying discussions do not cover all attack methods, prerequisites, vulnerabilities and countermeasures presented in our model.

References

[1] Ross J. Anderson. *Security Engineering.* John Wiley & Sons, Inc., 2001.

[2] Chip Andrews. Sql injection faq. Web advisory, apr 2003. `http://www.sqlsecurity.com/DesktopDefault.aspx?tabindex=2&tabid=3`.

[3] Chris Anley. Advanced sql injection in sql server application. Technical report, NGSSoftware Insight Security Research (NISR), 2002. `http://www.nextgenss.com/papers/advanced_sql_injection.pdf`.

[4] Chris Anley. (more) advanced sql injection in sql server application. Technical report, NGSSoftware Insight Security Research (NISR), jun 2002. `http://www.nextgenss.com/papers/more_advanced_sql_injection.pdf`.

[5] Craig Atkins. Data sanitization - reducing security holes in an asp web site. Web advisory, 2003. `http://www.4guysfromrolla.com/webtech/112702-1.shtml`.

[6] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice.* Addison-Wesley Longman Inc., 1999.

[7] CERT Coordination Center, DoD-CERT, the DoD Joint Task Force for Computer Network Defense (JTF-CND), the Federal Computer Incident Response Capability (FedCIRC), and the National Infrastructure Protection Center (NIPC). Cert® advisory ca-2000-02 malicious html tags embedded in client web requests. Technical report, Carnegie Mellon Software Engineering Institute, feb 2002. `http://www.cert.org/advisories/CA-2000-02.html`.

[8] Cesar Cerrudo. Manipulating microsoft sql server using sql injection. Technical report, Application Security, Inc. `http://www.appsecinc.com/presentations/Manipulating_SQL_Server_Using_SQL_Injection.pdf`.

[9] Robert Chartier. Application architecture: An n-tier approach - part 1. Online Documentation, 2001. `http://www.15seconds.com/issue/011023.htm`.

[10] Thomas Connolly, Carolyn Begg, and Ann Strachan. *Database Systems - A Practical Approach to Design, Implementation, and Management.* Addison - Wesley, 1999.

[11] Microsoft Corporation. https protocol. Online Documentation, 2003. `http://msdn.microsoft.com/workshop/networking/predefined/https.asp`.

[12] Martin Eizner. Direct sql command injection. Technical report, The Open Web Application Security Project, 2001. `http://qb0x.net/papers/MalformedSQL/sqlinjection.html`.

[13] Rik Farrow. Databases under fire. Web advisory, may 2002. `http://www.airscanner.com/pubs/sql.pdf`.

[14] Pete Finnigan. Sql injection and oracle, part one. Technical report, Security Focus, nov 2002. `http://www.securityfocus.com/infocus/1644`.

[15] Pete Finnigan. Sql injection and oracle, part two. Technical report, Security Focus, nov 2002. `http://www.securityfocus.com/infocus/1646`.

[16] JD Glaser. One-way sql hacking: Futility of firewalls in web hacking. In *Windows Security 2002*, Las Vegas, USA, jul 2002. Black Hat. `http://www.blackhat.com/presentations/win-usa-02/glaser-winsec02.ppt`.

[17] Distributed Technologies GmbH. 3- and n-tier architectures. Online documentation, 1998. `http://www.corba.ch/e/3tier.html`.

[18] Dieter Gollmann. *Computer Security*. John Wiley & Sons, 2001.

[19] Network Working Group. Hypertext transfer protocol – http/1.0, request for comments: 1945. Online Documentation, may 1996. `http://www.w3.org/Protocols/rfc1945/rfc1945`.

[20] Knut Halvorsen. *Samhällsvetenskaplig metod*. Studentlitteratur, 1992.

[21] Mitchell Harper. Sql injection attacks - are you safe? Technical report, DevArticles, may 2002. `http://www.devarticles.com/content.php?articleId=138&page=2`.

[22] Advosys Consulting Inc. Writing secure web applications. Online Documentation, jun 2002. `http://advosys.ca/papers/web-security.html`.

[23] Andrew Jaquith. The security of applications: Not all are created equal. Technical report, @stake, feb 2002. `http://www.atstake.com/research/reports/acrobat/atstake_app_unequal.pdf`.

[24] Brian Jepson. Beware sql injection in web applications. Web advisory, jun 2002. `http://www.oreillynet.com/pub/wlg/1595`.

[25] Himanshu Khatri. Sql server stored procedures 101. Web advisory, jun 2002. `http://www.devarticles.com/printpage.php?articleId=142`.

[26] Don Kiely. Guarding against sql injection attacks. Web advisory, mar 2002. http://www.itworld.com/nl/windows_sec/03252002/.

[27] Don Kiely. Sql injection attacks. Web advisory, mar 2002. http://www.itworld.com/nl/windows_sec/03182002/.

[28] Chong Siew Kok. Sql injection walkthrough. Web advisory, may 2002. http://www.scan-associates.net/papers/sql_injection_walkthrough.txt.

[29] Matthew Levine. The importance of application security. Technical report, @stake, jan 2003. http://www.atstake.com/research/reports/acrobat/atstake_application_security.pdf.

[30] David Litchfield. Exploiting windows nt 4 buffer overruns. Web advisory, may 1999. http://www.nextgenss.com/papers/ntbufferoverflow.html.

[31] David Litchfield. Web application disassembly with odbc error messages. In *Windows Security 2001*, Las Vegas, USA, jul 2001. Black Hat. http://www.blackhat.com/presentations/win-usa-01/Litchfield/bh-win-01-litchfield.doc.

[32] Paul Litwin. Guard against sql injection attacks. Web advisory, aug 2002. http://www.aspnetpro.com/opinion/2002/08/asp200208pl_o/asp200208pl_o.asp.

[33] Peng Liu. Dais: A real-time data attack isolation system for commercial database applications. In *17th Annual Computer Security Applications Conference*, New Orleans, Louisiana, dec 2001. Annual Computer Security Applications Conference (ACSAC). http://www.acsac.org/2001/papers/44.pdf.

[34] Macromedia. Multiple sql statements in dynamic queries. Security bulletin, feb 1999. http://www.macromedia.com/devnet/security/security_zone/asb99-04.html.

[35] Tony Marston. The 3-tier architecture - is it hardware or software? Web advisory, apr 2002. http://www.marston-home.demon.co.uk/Tony/3tierhardsoft.html.

[36] Stuart McDonald. Sql injection: Modes of attack, defence, and why it matters. Technical report, The SANS Institute, jul 2002. http://www.sans.org/rr/appsec/SQL_injection.php.

[37] Haaron Meer. Sql insertion. Web advisory, 2002. http://www.cgisecurity.com/lib/SQLinsertion.htm.

[38] J.D. Meier, Alex Mackman, Michael Dunner, and Srinath Vasireddy. Building secure asp.net applications: Authentication, authorization, and secure communication. Online Documentation, nov 2002. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/SecNetch12.asp.

[39] Memonix and MrJade. Abusing poor programming techniques in webserver scripts v 1.0. Web advisory, jul 2002. http://www.infosecuritymag.com/2002/jul/faster.shtml.

[40] Sun Microsystems. Using prepared statements. Online documentation. http://java.sun.com/docs/books/tutorial/jdbc/basics/prepared.html.

[41] Timothy M. Mullen. Web vulnerability and sql injection countermeasures. In *Windows Security 2002*, New Orleans, USA, feb 2002. Black Hat. http://www.blackhat.com/presentations/win-usa-02/mullen-winsec02.ppt.

[42] Aaron C. Newman. Protecting oracle databases. Technical report, Application Security, Inc., 2001. http://www.appsecinc.com/presentations/Protecting_Oracle_Databases_White_Paper.pdf.

[43] Ross Overstreet. Protecting yourself from sql injection attacks. Web advisory, 2003. http://www.4guysfromrolla.com/webtech/061902-1.shtml.

[44] Cyrus Peikari and Seth Fogie. Guarding against sql server attacks: Hacking, cracking, and protection techniques. Web advisory, 2003. http://www.airscanner.com/pubs/sql.pdf.

[45] Razvan Peteanu. Best practices for secure development, v4.03. White Paper, oct 2001. http://www.owasp.org/whitepapers/best_prac_for_sec_dev4.pdf.

[46] The International Common Criteria Project. Common criteria for information technology security evaluation - part 1: Introduction and general model, Version 2.1, CCIMB-99-031. Online Documentation, aug 1999. http://commoncriteria.org/docs/PDF/CCPART1V21.PDF.

[47] The International Common Criteria Project. Common criteria for information technology security evaluation - part 3: Security assurance requirements, Version 2.1, CCIMB-99-033. Online Documentation, aug 1999. http://commoncriteria.org/docs/PDF/CCPART3V21.PDF.

[48] The International Common Criteria Project. Common criteria for information technology security evaluation part 2: Security functional

requirements, Version 2.1, CCIMB-99-032. Online Documentation, aug 1999. http://commoncriteria.org/docs/PDF/CCPART2V21.PDF.

[49] The Open Web Application Security Project. A guide to building secure web applications, Version 1.1.1. Online Documentation, sep 2002. http://www.owasp.org/.

[50] The Open Web Application Security Project. The ten most critical web application security vulnerabilities, Version 1. Online Documentation, jan 2003. http://www.owasp.org/.

[51] Rain Forest Puppy (pseudonym). A quick look at web vulnerabilities and a small demo of sql tampering. In *Hack-Expo 2002*, Melbourne/Sydney, Australia, mar 2002. Wire-trip. http://www.wiretrip.net/rfp/talks/hackexpo-2002/hackexpo-sql-web/slide1.html.

[52] Yvel (pseudonym). Sql injection attack! Web advisory, sep 2002. http://www.roymoon.com/Articles.aspx?ArtNum=66.

[53] William A. Qualls. Exploit in action: A beginners view of incident handling for sql injection techniques. Technical report, SANS Institute, 2003. http://www.giac.org/practical/GCIH/William_Qualls_GCIH.pdf.

[54] Jason Rafail. Cross-site scripting vulnerabilities. Technical report, Carnegie Mellon Software Engineering Institute, 2001. http://www.cert.org/archive/pdf/cross_site_scripting.pdf.

[55] Mike Robertson. Understanding and preventing sql injection attacks. Web advisory, 2002. http://www.silksoft.co.za/data/sqlinjectionattack.htm.

[56] W3 Schools. W3 schools. Online documentation. http://www.w3schools.org/.

[57] SearchDatabase. Sql. Online documentation, mar 2003. http://searchdatabase.techtarget.com/sDefinition/0,,sid13_gci214230,00.html.

[58] Edward Skoudis. Cracker tools and techniques. Web advisory, jul 2002. http://www.infosecuritymag.com/2002/jul/faster.shtml.

[59] Ian Sommerville. *Software Engineering.* Addison-Welsey, 2001.

[60] Fons Sonnemans. Sql-strings considered harmful. Web advisory, mar 2003. http://www.reflectionit.nl/SqlInsert.aspx.

[61] Mark Spenik and Orryn Sledge. Microsoft sql server 2000 error messages. Web advisory, 2002. http://www.developer.com/db/article.php/10920_724711_1.

[62] Kevin Spett. Security at the next level - are your web applications vulnerable? Technical report, SPI Dynamics, 2002. http://www.spidynamics.com/whitepapers/webappwhitepaper.pdf.

[63] Kevin Spett. Sql injection - are your web applications vulnerable? Technical report, SPI Dynamics, 2002. http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf.

[64] William Stallings. *Network Security Essentials: Applications and Standards*. Prentice-Hall, Inc., 1999.

[65] Ron Talmage. Securing your sql server. White paper, dec 2002. http://www.devx.com/codemag/Article/10290/0/page/1.

[66] W3C. Hypertext markup language (html) home page. Online documentation. http://www.w3.org/MarkUp/.

[67] James Woodger. General web architecture. Web advisory, jan 2002. http://www.woodger.ca/archweb.htm.

[68] Joseph Yoder and Jeffrey Barcalow. Architectural patterns for enabling application security. In *PLoP 97*, Monticello, Illinois, USA, sep 1997. CiteSeer. http://citeseer.nj.nec.com/yoder98architectural.html.

A Glossary

We note that this is not a tutorial of concepts, but only a consistent set of definitions (even if they are not universally accepted) that we use throughout this thesis.

Access Control "...the ability to limit and control the access to host systems and applications via communications links. To achieve this control, each entity trying to gain access must first be identified, or authenticated, so that access rights can be tailored to the individual." [64]

Authentication "A mechanism that determines whether a user is, who he or she claims to be." [10]

Authorization "The granting of a right or a privilege, which enables a subject to have legitimate access to a system's object". [10]

Availability "The property that a procedure's services are accessible when needed and without undue delay." [18]

Buffer Overrun "A buffer overrun is when a program allocates a block of memory of a certain length and then tries to stuff too much data into the buffer, with the extra overflowing and overwriting possibly critical information crucial to the normal execution of the program." [30]

Confidentiality "Prevention of unauthorized disclosure of information." [18]

Cross-site scripting "malicious script [that] runs with the privileges of a legitimate script originating from the legitimate web server." [54]

Database Server A server on which a DBMS runs. [10]

Decryption The process of decoding encrypted data.

DBMS "A software system that enables users to define, create and maintain the database and provides controlled access to this database." [10]

Encryption "The encoding of the data by a special algorithm that renders the data unreadable by any program without the decryption key." [10]

Extranet "An intranet that is partially accessible to authorized outsiders." [18]

Firewall "...any security system protecting the boundary of an internal network." [18]

HTML "HTML is the lingua franca for publishing hypertext on the World Wide Web. It is a non-proprietary format based upon SGML..." [66]

HTTP "The Hypertext Transfer Protocol (HTTP) is an application-level protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems. HTTP has been in use by the World-Wide Web global information initiative since 1990" [19]

HTTPS "The secure hypertext transfer protocol (HTTPS) is a communications protocol designed to transfer encrypted information between computers over the World Wide Web. HTTPS is http using a Secure Socket Layer (SSL). A secure socket layer is an encryption protocol invoked on a Web server that uses HTTPS."[11]

Integrity "Prevention of unauthorised modification of information." [18]

Intranet "A Web site or group of sites belonging to an organization, accessible only by the members of the organization." [18]

OWASP "The Open Web Application Security Project (or OWASP–pronounced OH' WASP) was started in September of 2001. [...] OWASP is an open source reference point for system architects, developers, vendors, consumers and security professionals involved in Designing, Developing, Deploying and Testing the security of web applications and Web Services. In short, the Open Web Application Security Project aims to help everyone and anyone build more secure web applications and Web Services." [49]

Prepared Statement While prepared statements can be implemented in various ways, we prefer the concept as it is defined and used by Sun Microsystems [40]. Using Java technologies, a prepared statement is an object that is given an SQL statement when it is created. In most cases, this SQL statement will be sent to the RDBMS right away, where it will be compiled. As a result, the prepared statement object contains not just an SQL statement, but an SQL statement that has been precompiled. This means that when the prepared statement is executed, the RDBMS can just run the prepared statement's SQL statement without having to compile it first. According to Sun Microsystems, the main advantages relate to performance factors. However, we have found that for prepared statements that require arguments, the parameters sent will be considered illegal if they contain SQL keywords. Therefore, prepared statements are considered more secure than dynamic SQL.

RDBMS An RDBMS which supports the relational model. [10]

SQL SQL (Structured Query Language) is a standard interactive and programming language for getting information from and updating a database. Although SQL is both an ANSI and an ISO standard, many database products support SQL with proprietary extensions to the standard language. Queries take the form of a command language that lets you select, insert, update, find out the location of data, and so forth. [57]

SQL Injection (also named Direct SQL Command Injection, Query Poisoning, Malformed SQL, SQL Tampering, SQL Insertion and One-Way SQL Hacking) A technique that enables malicious users to exploit web applications and bypass control mechanisms in order to gain access to and manipulate information assets outside their privileges [12].

Threat "Any situation or event, whether intentional or unintentional, that will adversely affect a system and consequently an organization." [10]

URL "A string of alphanumeric characters that represents the location or address of a resource on the Internet and how that resource should be accessed." [18]

Vulnerability "A weakness in computer-based system that may be exploited to cause loss or harm." [59]

Web Application Web-based business applications that use a relational database for persistent storage of data [12].

Web Services "...a collection of functions that are packaged as a single entity and published to the network for use by other programs. Web services are building blocks for creating open distributed systems, and allow companies and individuals to quickly and cheaply make their digital assets available worldwide." [49]

B Internet Links

In this section we list some Internet links which discuss SQL injection or some aspect of it.

http://www.4guysfromrolla.com/
http://www.adopenstatic.com/
http://www.advosys.ca/
http://www.anticrack.de/
http://www.acsac.org/
http://www.airsdcanner.com/
http://www.appsecinc.com/
http://www.aspnetpro.com/
http://www.atstake.com/
http://www.baselinemag.com/
http://www.blackhat.com/
http://www.cert.org/
http://www.cgisecurity.com/
http://www.devarticles.com/
http://www.devx.com/
http://www.digitaloffense.net/
http://www.eckes.com/
http://www.extropia.com/
http://www.giac.org/
http://www.hillside.net/
http://www.infosecuritymag.com/
http://www.itworld.com/
http://www.lwn.net/
http://www.macromedia.com/
http://msdn.microsoft.com/
http://www.networkmagazine.com/
http://www.nextgenss.com/
http://newdata.box.sk/
http://www.oracle.com/
http://www.oreillynet.com/
http://www.owasp.org/
http://www.pentest-limited.com/
http://www.php.net/
http://www.qb0x.net/
http://www.reflectionit.nl/
http://www.sans.org/
http://www.secunia.com/
http://www.security-patterns.de/
http://www.securityfocus.com/
http://www.securiteam.com/
http://www.silksoft.co.za/
http://www.sitepoint.com/
http://www.software-factory.ch/
http://www.sourceforge.net/projects/wpoison/
http://www.spidynamics.com/
http://www.sqlsecurity.com/
http://www.webmasterbase.com/
http://www.whitehatsec.com/