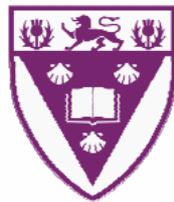


# **Analysis of SQL injection prevention using a filtering proxy server**

**Submitted in partial fulfilment of the degree of  
Bachelor of Science with Honours  
of**



**RHODES UNIVERSITY**

**By David Brodie Rowe**

Project Supervisor: Barry Irwin

Department of Computer Science

Rhodes University

November 2005

# Abstract

**This paper details an analysis of SQL injection prevention. This work is separated into two parts. The first highlights methods that should be adopted in order to reduce the risk of an SQL injection attack. The second details the creation of a filtering proxy server used to prevent a SQL injection attack and analyses the performance impact of the filtering process on web transactions. The test implementation focuses on Microsoft SQL Server 2000 although the guidelines are applicable to all database management systems. This is because SQL is a standard used by most databases.**

# Keywords

SQL injection, fault injection, SQL poisoning, prevention, preventing.

## **ACM classification categories relevant to this project**

C.2.0 Security and protection (e.g., firewalls) (REVISED)

D.4.6 Security and Protection

K.4.4 Security

K.4.2 Abuse and crime involving computers

K.6.5 Security and Protection

# Acknowledgements

I would like to thank Seven Fountains Digital for their support. Many thanks go to Professor Peter Wentworth for his patience, understanding and guidance.

I thank my supervisor, Mr Barry Irwin for his invaluable guidance and support throughout this project. I would also like to thank the Rhodes University Computer Science Department lecturers and my peers for their constructive feedback regarding the project.

Lastly, I would like to thank my family, honours lab peers and friends for their constant encouragement.

# Table of Contents

ABSTRACT .....	I
ACKNOWLEDGEMENTS .....	III
TABLE OF CONTENTS .....	IV
LIST OF FIGURES.....	VI
LIST OF CODE BOXES .....	VIII
LIST OF TABLES.....	VIII
LIST OF TEXT BOXES .....	IX
CHAPTER 1 - INTRODUCTION .....	1
1.1    INTRODUCTION .....	1
1.2    EXAMPLE OF SQL INJECTION.....	3
1.3    PROBLEM STATEMENT .....	4
1.4    DOCUMENT STRUCTURE .....	5
1.5    INTRODUCTION SUMMARY .....	5
CHAPTER 2 - RESEARCH.....	6
2.1    INTRODUCTION .....	6
2.2    LITERATURE REVIEW.....	6
2.3    PREVENTION METHODS .....	12
2.3.1 <i>Database Privileges</i> .....	12
2.3.2 <i>Error Trace</i> .....	13
2.3.3 <i>Suppressing Error Messages</i> .....	14
2.3.4 <i>Sanitising</i> .....	15
2.3.5 <i>SQL Signatures - Filtering SQL Injection</i> .....	17
2.4    EXISTING PRODUCTS .....	20
2.5    CONCLUSION.....	21
2.6    RESEARCH SUMMARY .....	22
CHAPTER 3 – SYSTEM DESIGN .....	24

3.1	INTRODUCTION .....	24
3.3	DESIGN CONCLUSIONS .....	30
3.4	SYSTEM DESIGN SUMMARY .....	31
<b>CHAPTER 4 – SYSTEM IMPLEMENTATION .....</b>		<b>32</b>
4.1	INTRODUCTION .....	32
4.2	IMPLEMENTATION .....	32
	<i>4.2.1 Design Decisions</i> .....	33
	<i>4.2.2 Methodology</i> .....	34
	<i>4.2.3 Testing and validation</i> .....	34
	<i>4.2.5 Problems Encountered</i> .....	47
	<i>4.2.6 Web transaction tests</i> .....	51
	<i>4.2.7 Conclusions</i> .....	54
4.3	SYSTEM IMPLEMENTATION SUMMARY .....	54
<b>CHAPTER 5 - CONCLUSION .....</b>		<b>56</b>
5.1	CONCLUSION .....	56
5.2	FUTURE WORK .....	57
<b>REFERENCES .....</b>		<b>59</b>
<b>APPENDIX A – PROJECT POSTER .....</b>		<b>65</b>
<b>APPENDIX B – CD CONTENTS .....</b>		<b>67</b>
<b>APPENDIX C – CODE OVERVIEW .....</b>		<b>69</b>
<b>APPENDIX D – TIMING TESTS.....</b>		<b>74</b>

# List of Figures

Figure 1: The OSI stack [Davis, 2005].....	7
Figure 2: Major Vulnerabilities in a multi-tier system [Microsoft, 2003a].....	8
Figure 3: Example of an error message returned by the database.....	15
Figure 4: Information flow diagram.....	25
Figure 5: High level design view.....	26
Figure 6: XML file containing the settings of TDSProxy.....	27
Figure 7: Flowchart of the TDSProxy server.....	28
Figure 8: State Change Diagram for Client Query.....	30
Figure 9: Proxy server connecting NetCat clients to a NetCat server.....	35
Figure 10: Proxy server listening on port 4444.....	36
Figure 11: NetCat simulating a server listening on port 5555.....	36
Figure 12: NetCat simulating one client connected to the proxy.....	37
Figure 13: NetCat simulating a second client connected to the proxy.....	37
Figure 14: Proxy server code setting up a view before connecting to the database.....	37
Figure 15: SQL server database view created via the proxy server.....	38
Figure 16: Use of OSQL and error message on shutdown of proxy server.....	39
Figure 17: Ethereal packet capture compared to proxy server packet analysis.....	40
Figure 18: Typical Usage sequences for TDS [FreeTDS, 2005].....	41
Figure 19: Packet format of all TDS packets [FreeTDS, 2005].....	41
Figure 20: Code showing the extraction of the query from TDS query packet.....	42
Figure 21: Successful SQL injection in the database.....	43
Figure 22: Dropping a table using SQL injection.....	43
Figure 23: Database view of the dropped table.....	45
Figure 24: Login error – not a trusted SQL server connection.....	47
Figure 25: Packet capture showing a successful login directly to the database.....	49

Figure 26: Attempted login from data access page through proxy server.....	50
Figure 27: The average web transaction processing time on hons08.....	52
Figure 28: The average web transaction processing time on Netserv .....	52
Figure 29: Graph showing the average web throughput for the client or server .....	53
Figure 30: Graph showing the average processing time of TDSProxy .....	54
Figure A.1: Project Poster .....	66
Figure B.1: Contents of project CD.....	68
Figure C.1: TCPSock Class .....	70
Figure C.2: ProxyBackend Class.....	71
Figure C.3: Logger Class.....	71
Figure C.4: Filter Cass.....	72
Figure C. 5: UDPServer Class .....	73
Figure D.1: Graph showing hons08 select page – direct to database .....	76
Figure D.2: Graph showing hons08 insert page – direct to database .....	77
Figure D.3: Graph showing hons08 select page, TDSProxy, no filter .....	78
Figure D.4: Graph showing hons08 select page, TDSProxy, filter .....	79
Figure D.5: Graph showing hons08 insert page - TDSProxy, no filter.....	80
Figure D.6: Graph showing hons08 insert page – TDSProxy, filter .....	81
Figure D.7: Graph showing netserv select page – direct to database .....	83
Figure D.8: Graph showing netserv insert page – direct to database .....	84
Figure D.9: Graph showing netserv select page, TDSProxy, no filter .....	85
Figure D.10: Graph showing netserv select page, TDSProxy, filter .....	86
Figure D.11: Graph showing netserv insert page, TDSProxy, no filter .....	87
Figure D.12: Graph showing netserv insert page TDSProxy, filter .....	88

# List of code boxes

Code Box 1: A typical SQL statement .....	3
Code Box 2: Resultant query .....	4
Code Box 3: Code showing a SQL injection vulnerability .....	4
Code Box 4: Code to replace a single quote with two single quotes.....	16
Code Box 5: Stored procedure code to produce a denial of service attack .....	18
Code Box 6: A stored procedure that is vulnerable to SQL injection .....	18
Code Box 7: Code showing a vulnerable stored procedure.....	19
Code Box 8: The resulting SQL .....	19
Code Box 9: Editing the connection string to stipulate a trusted connection.....	48

# List of tables

Table 1: Google search.....	3
Table 2: TDS Packet Types and their descriptions [FreeTDS, 2005] .....	41

# List of text boxes

Text Box 1: User input .....	4
Text Box 2: User input to delete the users table.....	4
Text Box 3: The parameter passed to @custname .....	19
Text Box 4: SQL injected queries executed by the database .....	44
Text Box 5: Login error.....	47

# Chapter 1 - Introduction

## 1.1 Introduction

According to [Kline, 2004], in the early 1970s, the seminal work of IBM research fellow Dr. E. F. Codd led to the development of a relational data model product called SEQUEL, or Structured English Query Language. SEQUEL ultimately became SQL, or Structured Query Language. American National Standards Institute (ANSI) has released standards for SQL in 1986, 1989, 1992, 1999, and 2003.

Structured Query Language (SQL) is a textual relational database language. Its command set has a basic vocabulary of less than 100 words. According to [Anley, 2002a], there are many varieties of SQL; however, the differences among the various dialects are minor. Most dialects are loosely based around SQL-92. According to [Kline, 2004], some of the popular dialects of SQL include:

- PL/SQL, found in Oracle.
- Transact-SQL, used by both Microsoft SQL Server and Sybase Adaptive Server.
- PL/pgSQL, implemented in PostgreSQL.
- SQLPL (SQLProcedural Language) is the newest dialect by DB2 .

SQL functions fit into two categories:

- Data definition language (DDL): used to create tables and define access rights.
- Data manipulation language (DML): SQL commands that allow one to insert, update, delete, and retrieve data within database tables [Rob and Coronel, 2002].

The typical unit of execution of SQL is the query, which is a collection of statements that typically return a single result set [Anley, 2002a].

SQL injection is a method by which users take advantage of dynamic SQL through which parameters of a web-based application are chained together to create the query to the backend database [Finnigan, 2002] [Anley, 2002a]. [Chuvakin and Peikari, 2004] add that SQL injection is not only an attack on the database but an attack on the database-driven application. There are many measures that can be taken to prevent SQL injection including ensuring that database privileges are to a minimum, using input validation programming techniques, suppressing error messages returned to the client, checking error logs and filtering malicious SQL statements [Finnigan, 2003].

Doing a Google search on "sql injection flaw vulnerability" returns over 171 000 hits. According to [OWASP, 2004] and [WebCohort Inc., 2004], injection flaws has been sixth in the top ten vulnerabilities for the past two years and that 62% of web applications are vulnerable to SQL injection attacks. Many web applications have been developed and deployed with SQL injection vulnerabilities. The problem is that most of the application owners do not even know that their applications are vulnerable to SQL injection. At least 92% of web applications are vulnerable to some form of hacker attacks [WebCohort Inc., 2004].

According to Google and the United States Computer Emergency Readiness Team (US-CERT), there has been an increase in the number of SQL injection vulnerabilities reported over the past five years. The number of vulnerabilities on the US-CERT website has almost doubled in the last year (see Table 1).

<b>Google Search String</b>	<b>Hits</b>
sql injection vulnerability US-CERT 2005	<b>85,400</b>
sql injection vulnerability US-CERT 2004	<b>38,800</b>
sql injection vulnerability US-CERT 2003	<b>31,100</b>
sql injection vulnerability US-CERT 2002	<b>16,600</b>
sql injection vulnerability US-CERT 2001	<b>524</b>

Table 1: Google search

According to [Hoglund and McGraw 2004], fault injection tools can be used to inject malformed or improperly formatted input to a target software process to cause failures. With the advent of automated tools such as Acunetix Web Vulnerability Scanner [Acunetix Ltd., 2005], Absinthe (which used to be known as SQueal) [Nummish and Xeron, 2005] and Whitehat Sentinel [WhiteHat Security, Inc., 2005] the risk of SQL injection exploits has risen. This was limited in the past by exploits having to be carried out manually. This was tedious and time consuming.

In research and commercial products, there is evidence proving SQL injection can be prevented using means not so closely related to the database and web application [Ristic, 2005] [Seclutions, 2003]. These methods of approach have been developed to produce a more generic solution to a problem that requires a lot of attention to detail at the root of the problem - the application code and database deployment. Traditional means of protecting against attacks include source code auditing, protecting dynamic input and limiting database privileges granted to users. However, auditing all of the source code and protecting dynamic input is not trivial, neither is reducing the permissions of all applications users in the database itself [Finnigan, 2003]. Therefore, developing a filter seems to be the best solution to preventing SQL injection. This project will provide an independent application that will sit between the web application and the database in order to provide security against SQL injection attacks.

## 1.2 Example of SQL Injection

A typical SQL statement is shown in Code Box 1.

```
select id, forename, surname from authors where forename = 'Joe' and  
surname = 'Bloggs'
```

**Code Box 1: A typical SQL statement**

An important point to note is that the string literals are delimited by single quotes. The user may be able to inject some SQL if the user provides the input shown in Text Box 2.

Forename: Jo'e	Surname: Bloggs
----------------	-----------------

**Text Box 1: User input**

The query string formed from the input shown in Text Box 1 is shown in Code Box 2.

```
select id, forename, surname from authors where forename = 'Jo'e' and
surname = 'Bloggs'
```

**Code Box 2: Resultant query**

In this case, the database engine will return an error due to incorrect syntax in the SQL query that it received.

In many web languages, a critical vulnerability is the way in which the query string is created. An example is shown in Code Box 3.

```
var SQL = "select * from users where username = '\" + username + \"'
and password = '\" + password + \"'";
```

**Code Box 3: Code showing a SQL injection vulnerability**

If the user specifies the input shown in Text Box 2, the 'users' table will be deleted, denying access to the application for all users [Anley, 2002a].

<b>Username:</b> `'; drop table users-
--

**Text Box 2: User input to delete the users table**

### 1.3 Problem Statement

The aim of this project is to produce software that will prevent SQL injection by filtering SQL query strings through a filtering proxy server. This will be done by analysing the structure of SQL query commands and investigating common SQL injection techniques. This will then be followed by building a filtering proxy server which will use attack signatures to prevent SQL injection. This aims to allow protection of vulnerable applications or complex applications that are difficult to audit for vulnerability bugs.

In an effort to reduce the deployment of vulnerable applications an extension of this project is to produce a list of best practices for Database Administrators and Software Developers with respect to preventing SQL injection.

#### **1.4 Document Structure**

- Chapter 1 presents an introduction to the project and outlines the work being presented in this document.
- Chapter 2 outlines some of the literature that was used to aid the design and implementation of the project.
- Chapter 3 details the design of the project software
- Chapter 4 runs through the implementation and results of the project.
- The final chapter, chapter 5, draws conclusions on the work done and provides possible project extensions.

#### **1.5 Introduction Summary**

SQL injection makes use of dynamic SQL. Dynamic SQL happens when parameters are chained together to create the database query. Most web applications are vulnerable to SQL injection or some form of hacker attack. There are many measures that can be implemented to reduce the chance of an attack. However, filtering malicious SQL statements seems to be the best solution in preventing SQL injection. Therefore, this project aims to produce a filtering proxy server to prevent SQL injection. In addition, a list of best practices that will provide a reference point to reduce the chance of deploying vulnerable applications.

The next chapter outlines the literature relevant to the project in an attempt to analyse the background and seek a direction for the design and implementation of the project. This is then followed by chapters showing the results and conclusion of the project.

# Chapter 2 - Research

## 2.1 Introduction

This chapter will present some of the literature related to this project as well as some suggested prevention methods. The chapter aims to put SQL injection into perspective by outlining some of the material and research that has already been completed. The section on suggested methods of mitigating SQL injection aims to clarify some misconceptions about SQL injection prevention and provides some useful tips to software developers and database administrators. A brief review of existing products concludes the chapter.

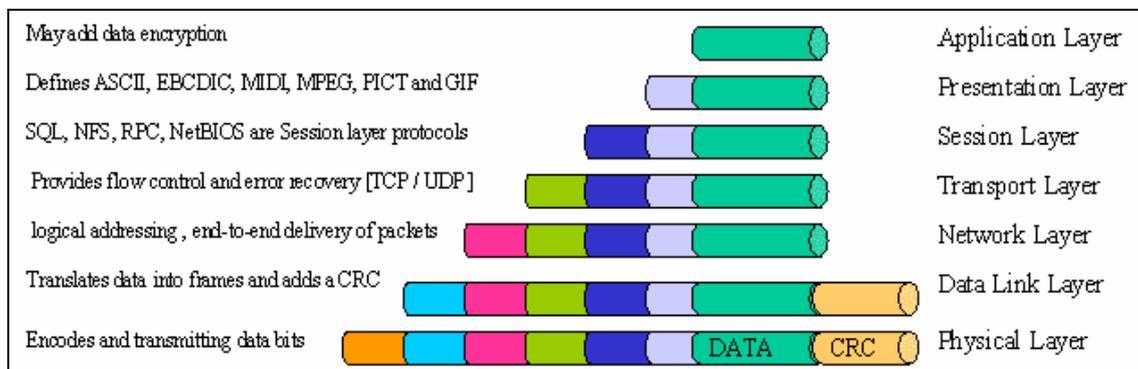
## 2.2 Literature Review

According to David Litchfield [Litchfield, 2005], one of the first publications revealing SQL injection was a 1998 Phrack 54 issue [Phrack, 2005], which was published on Christmas Day that year. The article by rain forest puppy in that publication did not use the term SQL injection but contained details of exploits.

SQL injection is a way to attack a database through a firewall by taking advantage of non-validated SQL vulnerabilities. It is a method by which the parameters of a Web-based application are modified in order to change the SQL statements that are passed to a backend database. An attacker is able to insert a series of SQL statements into a query by manipulating the data input, for example, by adding a single quote (') to the parameters. It is possible to cause a second query to be executed with the first.

In his paper entitled “*Advanced SQL injection*”, Chris Anley [Anley, 2002a] demonstrated that application developers often chain together SQL commands with user-provided parameters, and can therefore embed SQL commands inside these parameters. This is known as dynamic SQL. According to Peter Finnigan [Finnigan, 2002], dynamic SQL must be used in the application otherwise SQL injection is not possible. SQL injection has been described [Overstreet, 2004] as a “code hole” that is as serious as any IIS hole.

According to Pete Finnigan [Finnigan, 2002], existing SQL can be short-circuited to bring back all data. This technique is often used to gain access via third party-implemented authentication schemes.



**Figure 1: The OSI stack [Davis, 2005]**

It must be noted that firewalls, which traditionally operate at the network layer or layer 3 or the OSI stack, cannot protect against SQL injection as it takes place at layer 5 or the session layer of the OSI stack [Vicomsoft Ltd., 2003]. This can be clearly seen in the OSI stack diagram in Figure 1 above and is supported by [Imperva Inc., 2004]. Figure 2 shows the major points of vulnerability in a simple multi tier system. SQL injection takes place at the client.

Because the coding hole has created a direct tunnel for SQL injection from the client to the database, an attack is possible via a web server when the user has legitimate database access. According to Pete Finnigan [Finnigan, 2002], an attack against a database using SQL Injection could be motivated by three primary objectives:

1. To steal data from a database from which the data should not normally be available.

2. To obtain system configuration data that would allow an attack profile to be built. One example of this would be obtaining all of the database password hashes so that passwords can be brute-forced.
3. To gain access to an organisation's host computers via the machine hosting the database.

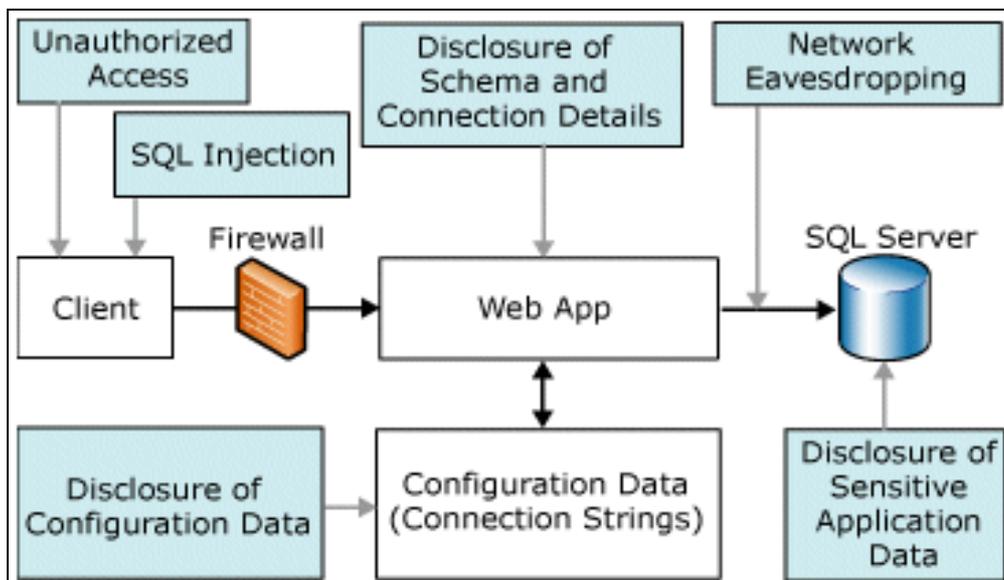


Figure 2: Major Vulnerabilities in a multi-tier system [Microsoft, 2003a]

According to David Litchfield [Litchfield, 2001] and Pete Finnigan [Finnigan, 2002], the following are some programming languages, APIs and tools that can access databases and be part of a Web-based application.

- Java Server Pages (JSP)
- Active Server Pages (ASP)
- XML, XSL and XSQL
- JavaScript and Asynchronous JavaScript and XML (Ajax)
- Visual Basic, C sharp, Java and other ODBC-based tools and APIs
- 3- and 4GL-based languages such as C, PHP and COBOL
- Perl, Python, Ruby and CGI scripts.

None of the above languages implicitly provide ways to protect against SQL injection.

According to Greg Hoglund and Gary McGraw [Hoglund and McGraw 2004] in their book entitled “*Exploiting software: how to break code*”, deep operating system integration leads to a security risk because integration runs counter to the principle of

compartmentalization. They also mention that one common assumption made by developers is that users of their software will never be hostile. Unfortunately this is a bad assumption to make as there are malicious users who will try to break software.

[Hoglund and McGraw 2004] also state that accepting anything blindly from the client and trusting it is a bad idea, and yet this is often the case with server side design. A potential hacker should not be implicitly trusted by a software system. Yet, most software happily accepts raw input from the user to perform database queries, file operations and system calls.

According to Maor and Shulman [Maor and Shulman, 2003], SQL injection attacks have been on the rise in the last few years. They outline research that has proved that suppressing error messages - going back to the “security by obscurity” approach [Finnigan, 2003] - cannot provide a real solution to application level risk but can add a measurement of protection. This is because no information can be inferred from error messages sent back to the client by the database.

[Microsoft, 2003b] offers the following tips for preventing SQL injection:

- Validate all user input before transmitting it to the web or database server. Authentication on the client side is vulnerable to SQL injection. According to [Hotchkies, 2004], it is possible to bypass the authentication on the client side. Therefore, the server must validate the input received in order to protect the database from unauthorised access.
- Permit only minimally privileged accounts to send user input to the server.
- Run SQL Server itself with the least necessary privileges.

Chuvakin and Peikari, in their book entitled “*Security Warrior*” [Chuvakin and Peikari, 2004], state that there are four types of SQL injection:-

- Unauthorised data access permits the attacker to trick the application into returning data that the attacker should not be able to see.
- Authentication bypass allows unauthorised access to data-driven applications without proper authentication credentials. The attacker is then allowed to observe data from the database.

- Database modification lets the attacker insert, modify or destroy database content without authorisation.
- Escape from a database allows the attacker to compromise the database host and possibly even attack other systems.

The novel presentation by Hotchkies [Hotchkies, 2004], at a Black Hat USA 2004 convention outlines automated blind SQL injection techniques. Because the automation tools are able to ask the database as many yes/no questions as they like, it is possible to use a binary search, for example, to discover an 8 character long username with 62 requests. Discovering the full database schema would take a few days, depending on the size of the database. Blind holes give the user a false sense of security. He mentions that string comparison is suitable for error based SQL injection but not blind SQL injection. He also mentions that there are three kinds of SQL injection:-

- Redirecting and reshaping a query involves inserting SQL commands into the query being sent to the database. The commands allow a direct attack on the database.
- Error message based SQL injection makes use of the database error messages returned to the client. The messages provide clues as to the database type and structure as well as the query structure.
- Blind SQL injection which involves a certain amount of guesswork and thus requires a larger investment in time. The attacker tries many combinations of attack and makes the next attack attempt based on their interpretation of the resulting html page output received from the target website. They are then able to infer the database type and structure. It should be noted that SQL injection can still occur if there is no feedback to the client. So, one could create a new valid user in a database without receiving errors and then log on.

David Litchfield [Litchfield, D 2005], in his recent paper titled “*Data-mining with SQL Injection and Inference*” defines three classes of SQL injection, namely inband, out of band and inference. They are outlined below:-

- Inband uses the existing connection to the database to manipulate the database. An example of this would be to use the data returned in a well formed web page or an error message.

- Out of band requires a new channel to be opened between the client and the application. This usually requires the database to connect out to the client using email, http or a database connection.
- Inference does not require any data transfer at all but uses properties such as web server response time or web server response codes. This allows the attacker to infer the value of the data they are enquiring about. Inference can be done at the bit level and the core of the attack is a simple question. If the answer is A, do Y; if the answer is B, do Z. David Litchfield has termed this very slow data-mining process “data chipping”. In the appendix to the paper there are advanced methods to avoid using single quotes, spaces, angle brackets, the ampersand and the equals’ character.

[Microsoft, 2003a] provides a good background into the problem of SQL injection by providing explanations of the components of SQL injection strings and the syntax choices. The examples include SQL injection attacks and show the creation of a secure data access component using Java’s regular expressions.

[Beyond Security Ltd., 2002] provides concise examples of SQL injection and database error messages as well as methods on how to prevent SQL injection.

[Spett, 2002] of SPI Dynamics presented a paper that describes SQL injection in general. It goes through some common SQL injection techniques and proposes data sanitizing and better coding as some of the solutions to the problem. The paper provides a list of database tables that are useful to SQL injection in MS SQL Server, MS Access and Oracle. It also provides examples of SQL injection using select, insert, union, stored procedures. The examples work with a web service that returns information to the user. The paper deals primarily with the structure of the SQL injection commands and guidelines to reducing errors returned by the database.

Mr. Grossman, CEO of White Hat Security, Inc., in his presentation at the Black Hat Windows Security 2004 convention, outlines the challenges of scanning web application code for vulnerabilities. He points out that the scanner is restricted to looking for classes of vulnerabilities such as SQL injection or cross site scripting. The reason for this being that the benefit of known security issues is lost because the remote scanner does not have access to the source code. Without the source code, knowledge of

the programming language or even what platform the application resides on, it is virtually impossible for a remote vulnerability scanner to pick up known critical vulnerabilities. He states that the problem with automated web application scanning is in detecting "known security issues in unknown code" [Grossman, 2004].

[Kc, Keromytis, and Prevelakis, 2003] presented their paper on "*Countering code-injection attacks with instruction-set randomization*" in Proceedings of the 10th ACM conference on Computer and communication security in Washington D.C. in 2003. This intriguing work describes a new, general approach for safeguarding systems against *any* type of code-injection attack. This is done by creating process-specific randomized instruction sets (*e.g.*, machine instructions) of the system executing potentially vulnerable software. An attacker who does not know the key to the randomization algorithm will inject code that is invalid for that randomized processor, causing a runtime exception. This method of protection can be used to protect scripting and interpreted languages from code injection attacks.

## **2.3 Prevention Methods**

There are many preventative measures that can be implemented by the administrator of the database and web application interfaces. These include ensuring that the users have the minimum database privileges possible, using input validation programming techniques, suppressing error messages returned to the client, checking error logs and filtering malicious SQL statements. These are explained below in more detail.

### **2.3.1 Database Privileges**

According to [Howard and LeBlanc, 2003], when developers use 'sa' accounts to ensure that everything works so that no extra configuration is required at the back are also ensuring that everything works for the attackers too.

Prevention is better than cure. One should adopt the principle of least privilege by ensuring that the users created for the applications have the privileges needed and all extra privileges (such as PUBLIC ones) are not available. According to [Microsoft, 2003a], this principle can be extended by permitting only minimally privileged accounts

to send user input to the server and running the database server itself with the least necessary privileges. The application generally does not need 'dbo' or 'sa' permissions. By limiting the permission granted to the database, one is able to limit the vulnerability of the database. Generally, users should not be allowed to delete records from a database. They should only be granted the minimum privileges required for the tables that they need whilst not having any rights to access tables that they do not require. Read only access is far safer than read write access. Both read only and read write are far safer than full control.

With Microsoft SQL Server, if the database connection string uses the security context of 'dbo', it is possible to use Data Definition Language (DDL) SQL commands such as *drop* and *create*. If the database connection uses the security context of 'sa', it is possible to control the entire SQL Server, and under the correct configuration even create user accounts to take control of the Windows server hosting the database [Overstreet, 2004]. [Overstreet, 2004] suggests that one should consider using a separate account for each component with data access capabilities to isolate vulnerabilities. For instance, a front-end public interface to one's Web site needs more restricted database access than an internal content management system.

[Finnigan, 2003] is an extension of a two-part paper on investigating the possibilities for an Oracle database administrator to detect SQL injection. The paper provides many scripts on SQL injection and extracting logs and goes through worked examples of SQL injection attacks. The paper focuses on detecting SQL injection by auditing the error message log files and attempts to highlight the fact that during a hacking attempt, the error messages leave a trail that can help expose the vulnerabilities of the database being attacked. According to [Finnigan, 2003], there is no way to provide everyone with the minimum privileges necessary and thus his paper explores some simple techniques in extracting the logging and trace data that could be used for monitoring. This will be discussed in the next section.

### **2.3.2 Error Trace**

Some detection is better than none at all. It is easier to detect if SQL injection has occurred by auditing the errors generated when the hacker is trying to gain access to the

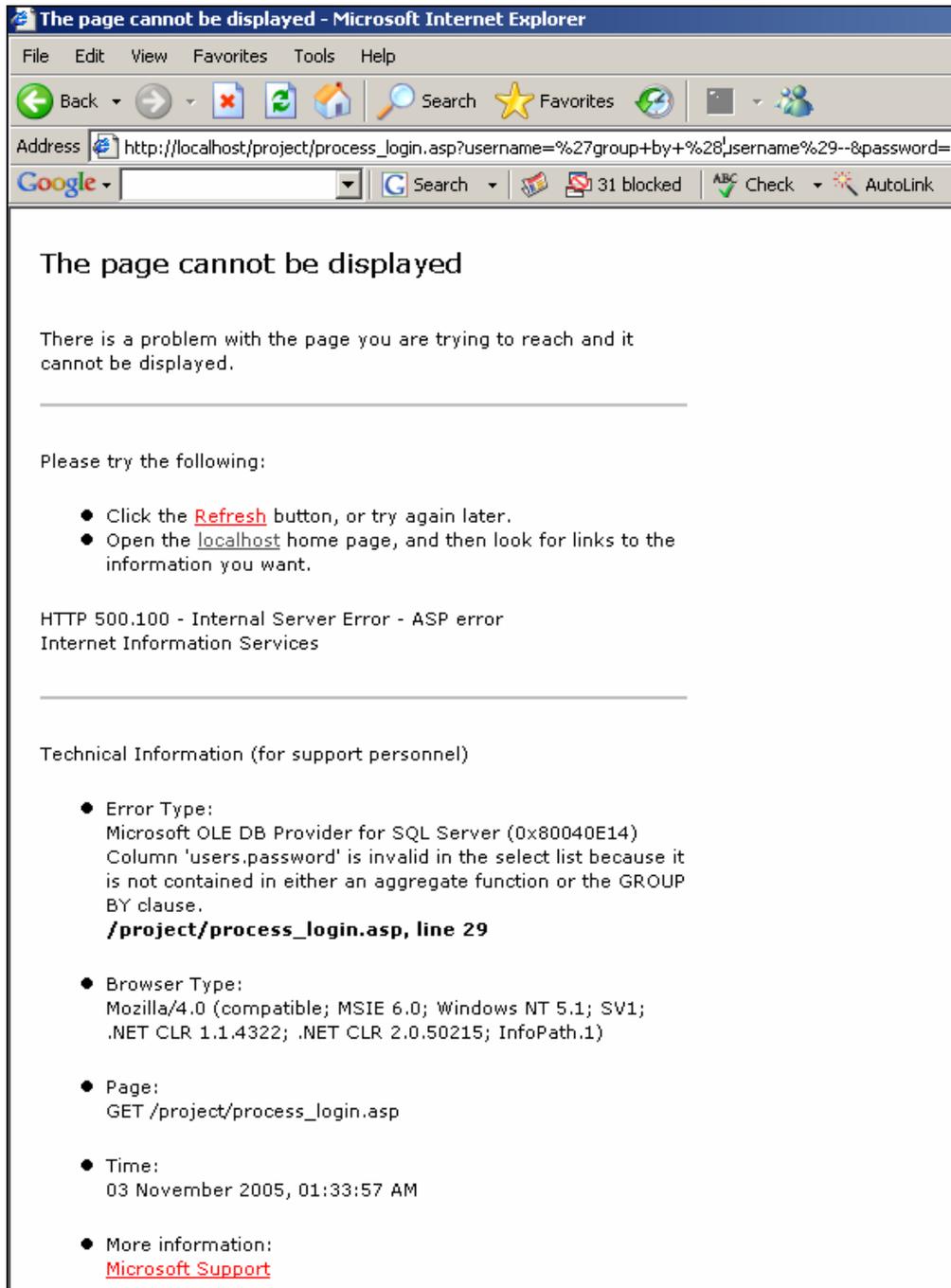
database as opposed to auditing of the SQL commands executed. These error messages can be as useful to the hacker as they are to the database administrator building up database queries and stored procedures [Finnigan, 2003]. According to [Finnigan, 2003], SQL injection detection is possible but not in real time. One should use the log files of traced data to scan for irregular SQL statements. A few of the disadvantages of this are that it requires a large computational overhead, a large amount of disk space may be required to store the logs and by the time one has found that a table name or a view has been changed, it is too late - the damage has already been done. One other point that [Finnigan, 2003] makes is that an attacker can steal the admin account, making it hard to distinguish normal administration from an attack on the database.

### **2.3.3 Suppressing Error Messages**

Error messages typically contain information that can be used to make informed decisions on the next attack method. Security by obscurity tries to reduce the unnecessary information from being sent back to the client. According to [Cerrudo, 2004], [Anley, 2002a] and [Litchfield, 2001] error messages can be used to determine information such as the database type and table structure.

An example of a useful error message is shown in Figure 3. By entering `'group by (username) --` into the input box, the error message returned gives information about the type of database as well as the table and column name.

[Microsoft, 2003a] and [Overstreet, 2004] advise removing any technical information from client-delivered error messages. [Maor and Shulman, 2003] makes the point that the absence of error messages makes it harder but not impossible to use SQL injection if the application is vulnerable. They also outline research that has proved that suppressing error messages - the “security by obscurity” approach - cannot provide a real solution to application level risk. Applications have still proven to be vulnerable despite all efforts to limit information returned to the client. According to [Chuvakin and Peikari, 2004], the obfuscation method of defence is a poor means of defence and should be coupled with good coding.



**Figure 3: Example of an error message returned by the database**

### 2.3.4 Sanitising

[Microsoft, 2003a] says that web applications should validate all user input before transmitting it to the server. Programmers should not make use of dynamic SQL that uses concatenation anywhere in the application [Finnigan, 2002] [Finnigan, 2003]. If concatenation is necessary then the input should be checked for malicious code, i.e. unions in the string passed in or meta-characters such as quotes. Using numeric values

for the concatenation part is a way to ensure that SQL strings cannot be injected into the query [Finnigan, 2002].

Fortify Software Inc. [Fortify Software Inc., 2004] aims to give companies an automated way to discover flaws in code that could lead to threats such as buffer overflows, format string errors and SQL injection exploits. When reviewing the source code for dynamic SQL where concatenation is used, one should find the call that parses the SQL or executes it. Input values must be validated, quotes must be matched and meta-characters must be checked [Finnigan, 2002]. Kavado's ScanDo web application scanner [Kavado, 2005] creates a report showing how to eliminate web application vulnerabilities after scanning the entire website.

Web Application Vulnerability and Error Scanner (WAVES, an open-source project available at <http://waves.sourceforge.net>) is presented by [Huang, Huang, Lin, and Tsai, 2003] at the 12th international conference on World Wide Web table of contents in Budapest, Hungary. This security assessment tool has the ability to learn from previous experience and build up its injection knowledgebase. WAVES detects SQL injection by using a black-box "complete crawling" mechanism and reverse engineering the web application to determine all data entry points and then applying fault injection techniques. This intriguing project is presented in their paper "*Web Application Security Assessment by Fault Injection and Behavior Monitoring*" [Huang, Huang, Lin, and Tsai, 2003].

When using Active Server Pages (ASP) technology with Active Data Objects (ADO), [Litchfield, 2001] suggests that the fix for the SQL injection problem is better coding which includes the *replace()* function for strings is a way to escape single quotes in SQL. [Overstreet, 2004] reaffirms these sanitization techniques, adding that one should check all input received from any ASP request object e.g. *Request.QueryString*. An example of this is shown in Code Box 4. However, this will not work on fields that do not require string inputs or if the user uses a hex equivalent of the word [Howard and LeBlanc, 2003].

```
Replace(Request.QueryString("foobar"), "'", "' ' ")
```

**Code Box 4: Code to replace a single quote with two single quotes**

[Chuvakin and Peikari, 2004] advise regular penetration testing and web application scanning. These tests will provide reports on how to patch vulnerabilities in the application. A web shield should be used for additional layered security.

### **2.3.5 SQL Signatures - Filtering SQL Injection**

Data Definition Language (DDL) can be injected if DDL is used in a dynamic SQL string. Other databases can be injected through the first by using database links [Finnigan, 2002]. According to [Microsoft, 2005], Microsoft® SQL Server™ 2000 uses reserved keywords for defining, manipulating and accessing databases. Reserved keywords are part of the grammar of the Transact-SQL language used by SQL Server to parse and understand Transact-SQL statements and batches. Although it is syntactically possible to use SQL Server reserved keywords as identifiers and object names in Transact-SQL scripts, this can only be done using delimited identifiers. In addition, the SQL-92 standard defines a list of reserved keywords. Avoid using SQL-92 reserved keywords for object names and identifiers. The ODBC reserved keyword list is the same as the SQL-92 reserved keyword list. The SQL-92 reserved keywords list sometimes can be more restrictive than SQL Server and at other times less restrictive. For example, the SQL-92 reserved keywords list contains INT, which SQL Server does not need to distinguish as a reserved keyword.

[Microsoft, 2005] also adds that transact-SQL reserved keywords can be used as identifiers or names of databases or database objects, such as tables, columns, views, and so on. Use either quoted identifiers or delimited identifiers. The use of reserved keywords as the names of variables and stored procedure parameters is not restricted [Microsoft, 2005]. The effect of this possible overlap with keywords being used is that filtering out false negatives is more likely. The filter will not be able to determine if the words are legitimate or are part of an attack query. By the same token, allowable words based on the database schema may lead to harmful code passing through the filter undetected.

Existing SQL can be short-circuited to bring back all data. This technique is often used to gain access via third party-implemented authentication schemes. A large selection of installed packages and procedures are available on Microsoft SQL Server 2000. These include packages to read and write O/S files. [Beyond Security Ltd., 2002] suggests that stored procedures such as `xp_cmdshell`, `xp_startmail`, `xp_sendmail` and `sp_makewebtask` in the master database should be deleted if they are not going to be used.

Executing the stored procedure shown in Code Box 4 will prevent the SA account from logging onto the server, a powerful denial of service attack [Lawson, 2005].

```
XP_REVOKELOGIN{[@LOGINAME=] 'SA' }
```

**Code Box 5: Stored procedure code to produce a denial of service attack**

Users should not be able to perform direct CRUD (Create, Read [Select], Update and Delete) statements. Erland Sommarskog [Sommarskog, 2005], SQL Server MVP (Microsoft Valued Professional) advises that stored procedures should be used because they increase performance. For example, with a long select statement that relies only on the where clause being changed, using a stored procedure can limit the amount of data transferred. Another reason for using stored procedures is because SQL server caches the first execution of the stored procedure. Consequent calls to the procedure will be executed in less time.

According to [Anley, C 2002b], it is possible for stored procedures to be vulnerable to SQL injection. In his example shown in Code Box 5, by default, the 'sp\_msdropretry' system stored procedure is accessible to 'public' and allows SQL injection.

```
sp_msdropretry [foo drop table logs select * from sysobjects], [bar].
```

**Code Box 6: A stored procedure that is vulnerable to SQL injection**

Consider the procedure in Code Box 4, as illustrated by [Sommarskog, 2005]:

```
CREATE PROCEDURE search_orders @custname varchar(60) = NULL,
```

```

        @prodname varchar(60) = NULL AS
    DECLARE @sql nvarchar(4000)
    SELECT @sql = 'SELECT * FROM orders WHERE 1 = 1 '
    IF @custname IS NOT NULL
        SELECT @sql = @sql + ' AND custname LIKE "' + @custname + '"'
    IF @prodname IS NOT NULL
        SELECT @sql = @sql + ' AND prodname LIKE "' + @prodname + '"'
    EXEC(@sql)

```

**Code Box 7: Code showing a vulnerable stored procedure**

Assume that the input for the parameters @custname and @prodname comes directly from user-input fields. Assume further that a malicious user passes the value in Text Box 3 to @custname, the resulting query is shown in Code Box 6.

```
' DROP TABLE orders --
```

**Text Box 3: The parameter passed to @custname**

```
SELECT * FROM orders WHERE 1 = 1 AND custname LIKE '' DROP TABLE
orders --'
```

**Code Box 8: The resulting SQL**

According to [Anley, 2002b], the vulnerability is caused by the 'exec' statement. Any stored procedure that uses the 'exec' statement to execute a query string that contains user - supplied data should be carefully checked for SQL injection. [Howard and LeBlanc, 2003] suggest using the *quotename()* function for object names and using *sp\_executesql* to execute dynamically built SQL statements.

According to Litwin [Litwin, 2005], using parameterized SQL greatly reduces the hacker's ability to inject SQL into your code. Paul Litwin is a lead programmer with Fred Hutchinson Cancer Research Center in Seattle. He is the chair of the Microsoft ASP.NET Connections conference and the owner of Deep Training, a .NET training company. [Howard and LeBlanc, 2003] advise the use of strongly typed parameters in the web application because parameterised queries are faster and more secure.

According to [Finnigan, 2002] and [Finnigan, 2003], common attack techniques include the use of:

- UNIONS that can be added to an existing statement to execute a second statement;
- SUBSELECTS which can be added to existing statements;
- A large selection of installed packages and stored procedures which include packages to read and write O/S files;
- Data Definition Language (DDL) can be injected if DDL is used in a dynamic SQL string;
- INSERTS, UPDATES and DELETES; and,
- Other databases can be injected through the first by using database links.

According to [Chuvakin and Peikari, 2004], there are two types of external filtering that only allow legitimate requests to pass through the system. SQL shielding protects the database and web shielding protects the web application itself. Patiently trying various innovative injection types may result in the attacker bypassing this method of defence.

## **2.4 Existing Products**

Applications have still proven to be vulnerable despite all efforts to limit information returned to the client. There are a few applications that have been developed by companies in an effort to provide a solution to this problem. Some have been outlined below:

- SecureSphere [Imperva Inc., 2005] uses advanced anomaly detection, event correlation, and a broad set of signature dictionaries to protect web applications and databases. It also uses error responses from the same user to identify an attack.
- ModSecurity is an open source intrusion detection engine for web applications, which may provide helpful tips on how to detect SQL injection. [Ristic, 2005] has developed ModSecurity for Java which is a Servlet 2.3 filter that stands between a browser and the application, monitors requests and responses as they are passing by, and intervenes when appropriate in order to prevent attacks.
- AirLock combines secure reverse proxy with intrusion prevention, content filtering, user authentication enforcement, and application-level load balancing and failover [Seclutions, 2003]. (Seclutions' AirLock was awarded the Swiss Technology Award 2003)

- McAfee® Enterscept® Database Edition [Networks Associates Technology, Inc., 2005] provides many sophisticated proactive database protection techniques. The SQL interception engine screens all incoming database queries and blocks any that would cause malicious activity; database shielding blocks both outside penetration and malicious use
- Amongst many features, Connectra Web Security Gateway [Check Point Software Technologies Ltd., 2004] can prevent users from accessing confidential data using directory traversal or SQL injection attacks whilst providing connectivity at the same time.

## 2.5 Conclusion

"A true SQL injection tool would involve writing a parser or filter to analyse the SQL statements" [Finnigan, 2003]. The ideal solution is to build a filter that checks for all cases of SQL injection possible. The problem with this is that a list of all possible injection strings is not possible to define [Finnigan, 2003]. This is suggested by [Maor and Shulman, 2004] in their paper on "*SQL Injection Signatures Evasion*". However, going back to the principle of least privilege, by using a white list, it is possible to define what is allowed and thus prevent invalid signatures.

The filtering application should sit as close to the database as possible. Ideally it should sit on the same machine as the database; however, this may have a performance impact due to the filtering process of the filtering proxy server. If the filtering application and the database are on different machines, there is a security risk as the network traffic passes from one machine to the other. With the filtering application sitting on the same machine as the database, there are several advantages.

- There is an additional security as network traffic is limited.
- Processing time is reduced as network latency has no additional effect on the transaction round trip time.
- The filtering application provides a last means of defence for the database.

There are some advantages to using SQL signature filtering as a preventative measure to SQL injection. These are:

- Real time analysis does not impact the database [Finnigan, 2003].
- Any flaws in the configuration of database privileges or coding of the application would not affect the database security.

However, there are also several disadvantages.

- False positives may also be filtered out in the filtering process [Imperva Inc., 2004].
- Packet filtering does not show internal dynamic SQL execution [Finnigan, 2003].
- This method will not work if the data is encrypted because the strings cannot be viewed in plain text without decryption [Finnigan, 2002] [Linux Journal, 2004].
- Filtering all incoming http packets may turn out to be resource intensive. A large amount of traffic may need to be handled at the webserver [Finnigan, 2003].

To be a useful intrusion detection system, the filter should be able to find the attackers. Finding the user or attacker means logging login information for inspection. The filter would need a timestamp as well as the source and destination IP address [Finnigan, 2003].

Most of the suggestions above apply to future deploying of web applications. The (open web application security project) OWASP guide with its many precautions is now becoming an accepted standard [OWASP, 2004].

## **2.6 Research Summary**

There are many vulnerable applications whose code will not be reviewed or patched and it is common knowledge that programmers will continue to produce vulnerable applications. According to [Finnigan, 2003], there are no commercial solutions to SQL injection. However, several post 2003 software packages have been found that claim to prevent SQL injection attacks.

Auditing all of the source code and protecting dynamic input is not trivial, neither is reducing the permissions of all applications users in the database itself. Checking through log files and relying on the least privileges principle does not seem sufficient. Passively detecting SQL injection is not as useful as preventing it in real time. The use

of packet sniffers does not allow for the SQL injection prevention as the removal of malicious SQL query statements from the packets is not possible.

This chapter has introduced SQL and SQL injection, outlined background research, discussed methods of protecting against SQL injection and presented existing software on the market today. Given the fact that there is a finite set of words in the SQL vocabulary, it seems possible to develop a filter to prevent SQL injection.

The following chapter outlines the design of a filtering proxy server. The design process makes use of unified modelling language (UML) in an iterative process with the implementation of the design. Following the design chapter are chapters on implementation, the results and conclusions of this project.

# Chapter 3 – System Design

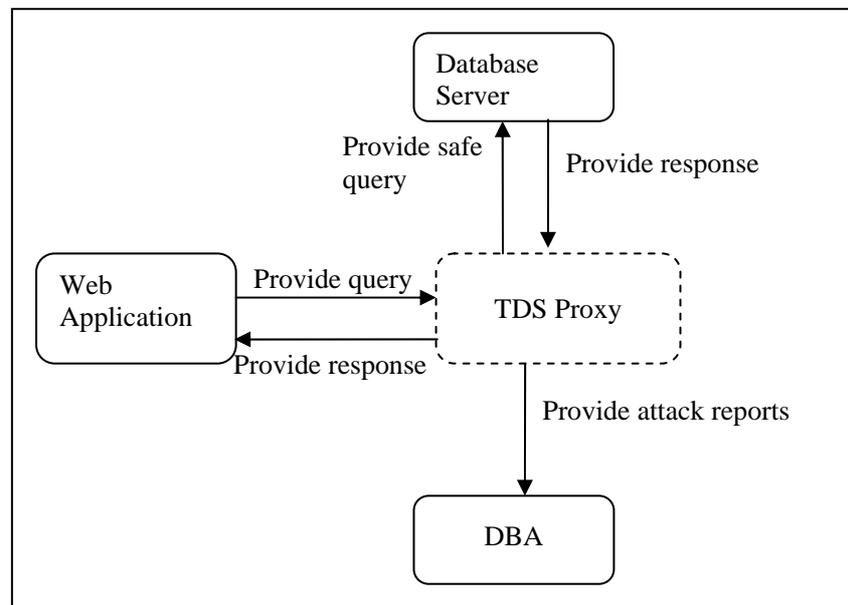
## 3.1 Introduction

This chapter outlines the design of the filtering proxy server. The project aims to eliminate the possibility of SQL injection by the use of a filtering proxy server, which will be placed in between the two communicating devices, namely the web application or client and the database. This added layer of protection will allow for the filtering of possible SQL injection attempts and provide the database with a last means of defence. Protecting the database can be best achieved by keeping the proxy server and database close together. This is achieved by having them run on the same machine. It is possible for the filtering proxy server to run as a standalone application on a separate machine sitting between the two communicating devices. It is also possible to run the application on the web server or database server. The intended outcome is to set up an environment that provides protection against SQL injection by filtering bad SQL queries and only allowing good queries to be executed by the database.

## 3.2 Design

The information flow diagram in Figure 4 shows the flow of information between a TDSProxy server within the domain of this project and the other entities and abstractions with which it communicates. The diagram helps to discover the scope of the system and identify the system boundaries. The system under investigation (TDSProxy) is represented as a single process interacting with various data and resource flow entities via an interface. As can be seen from the diagram, the web application

provides the query to TDSProxy which in turn provides safe queries to the database and attack reports to the Database Administrator. The response from the database is routed back to the web application through TDSProxy. Should the need arise, log files in the database application provide information for auditing purposes at a later stage.



**Figure 4: Information flow diagram**

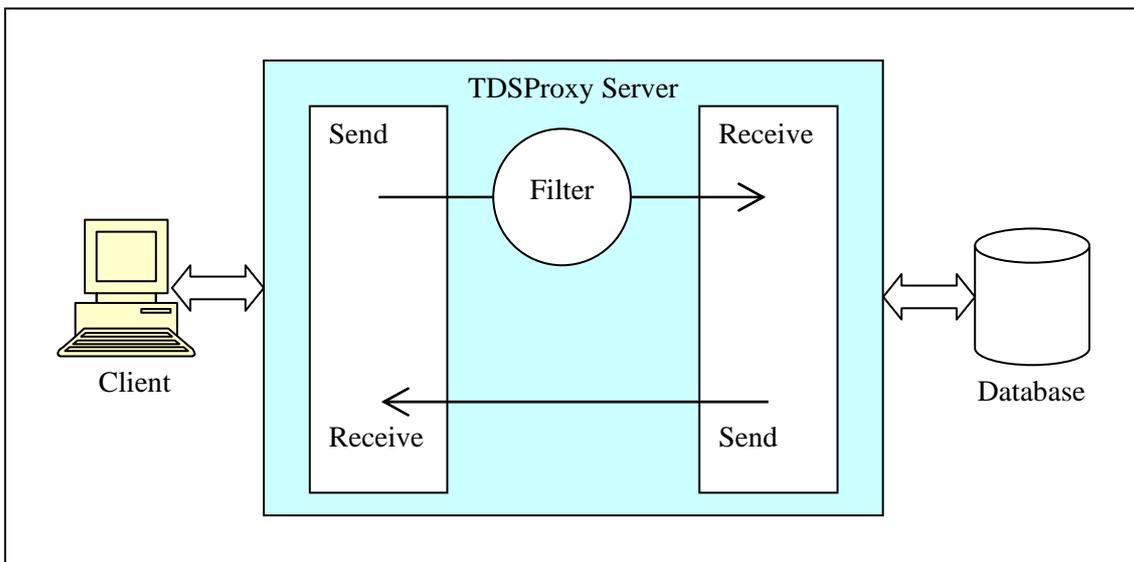
The reason for naming the application TDSProxy is because Microsoft SQL Server 2000 uses the Tabular Data Stream (TDS) protocol to communicate with its clients. The application being developed is a proxy server that filters queries being carried by the TDS protocol. Therefore, the name came about by a combination of the acronym TDS and the word Proxy to form TDSProxy.

The design and implementation steps of the project used the Rational Unified Process (RUP) with the aid of UML (Unified Modelling Language). The process was iterative, started off with a simple application and developing into a more complex system in subsequent iterations. This methodology was chosen to overcome problem areas in segments.

One of the problems was not being able to create a connection from a client application – a Microsoft Access data access page - through the proxy server to the database. The approach to solving this problem was to break it down into its components, making sure that the proxy server could create a connection to the database, ensuring that the data

access page was connecting to the proxy server and finally ensuring that the data was passed through the transparent proxy server without being changed. Once the basic concept was conceived and implemented, more advanced features were added to flesh out the software used for this proof of concept project.

The web application is where the queries are formed from the input parameters. These queries are sent to the database through TDSProxy. The bulk of the system operations take place at the TDSProxy. When the TDSProxy has filtered the query, the clean query is sent to the database server. Incoming requests are filtered and only clean queries are passed on to the database for processing (Figure 3). For security reasons, the proxy server will sit on the same machine as the database.



**Figure 5: High level design view**

The diagram in Figure 5 shows all the components in the high level view of the system. The web interface is the tool used by the client to send requests to the database. The web application is pointing to TDSProxy so that all requests and responses must go through TDSProxy.

The client's web application request triggers the formation of the SQL statement which uses the input parameters of the web form to create the correct SQL statement. This SQL statement is then sent to TDSProxy. When the SQL statement is received, it is first filtered. Only clean SQL statements are then sent to the database. The database processes the request and sends its response through TDSProxy. TDSProxy in turn

sends the response to the web application for processing to produce the correct view for the client.

At start-up, TDSProxy loads a configuration file, shown in Figure 6, by extracting the parameters. This is an xml file that contains, filter settings and options as well as the settings required for the passing of data to the correct destination. The filter signatures are also loaded from text files. These text files are easily updatable.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<appSettings>
  <!-- external config file to control some of the settings -->

  <!-- For this proxy server database port was 2222 for MSSQL-->
  <!-- For this proxy server database port was 3306 for mysql-->
  <add key="ListeningPort" value="4444" />
  <add key="DatabasePort" value="2222" />

  <add key="LogFile" value="SQL Injection 2005.txt" />

  <!-- For the udp server sending halted SQL injection strings to DBA -->
  <add key="UDPListenerPort" value="8080" />
  <add key="UDPSenderPort" value="8080" />
  <add key="MulticastGroupAddress" value="224.0.0.1" />

  <!-- For the sql filter -->
  <add key="black" value="True" />
  <add key="white" value="True" />
  <add key="gray" value="True" />
  <add key="regex" value="True" />

  <add key="SearchOrder1" value="regex" />
  <add key="SearchOrder2" value="white" />
  <add key="SearchOrder3" value="black" />
  <add key="SearchOrder4" value="gray" />

</appSettings>
</configuration>
```

**Figure 6: XML file containing the settings of TDSProxy**

The flowchart in Figure 7 focuses on the internally driven processes as opposed to external events, capturing the actions performed at system start-up and run time. The action states in the diagram represent the decisions and behaviour of the processing. TDSProxy is not a passive application but analyses the TCP payload for TDS query packets. For simplicity of testing and real time analysis, one client application can connect TDSProxy.

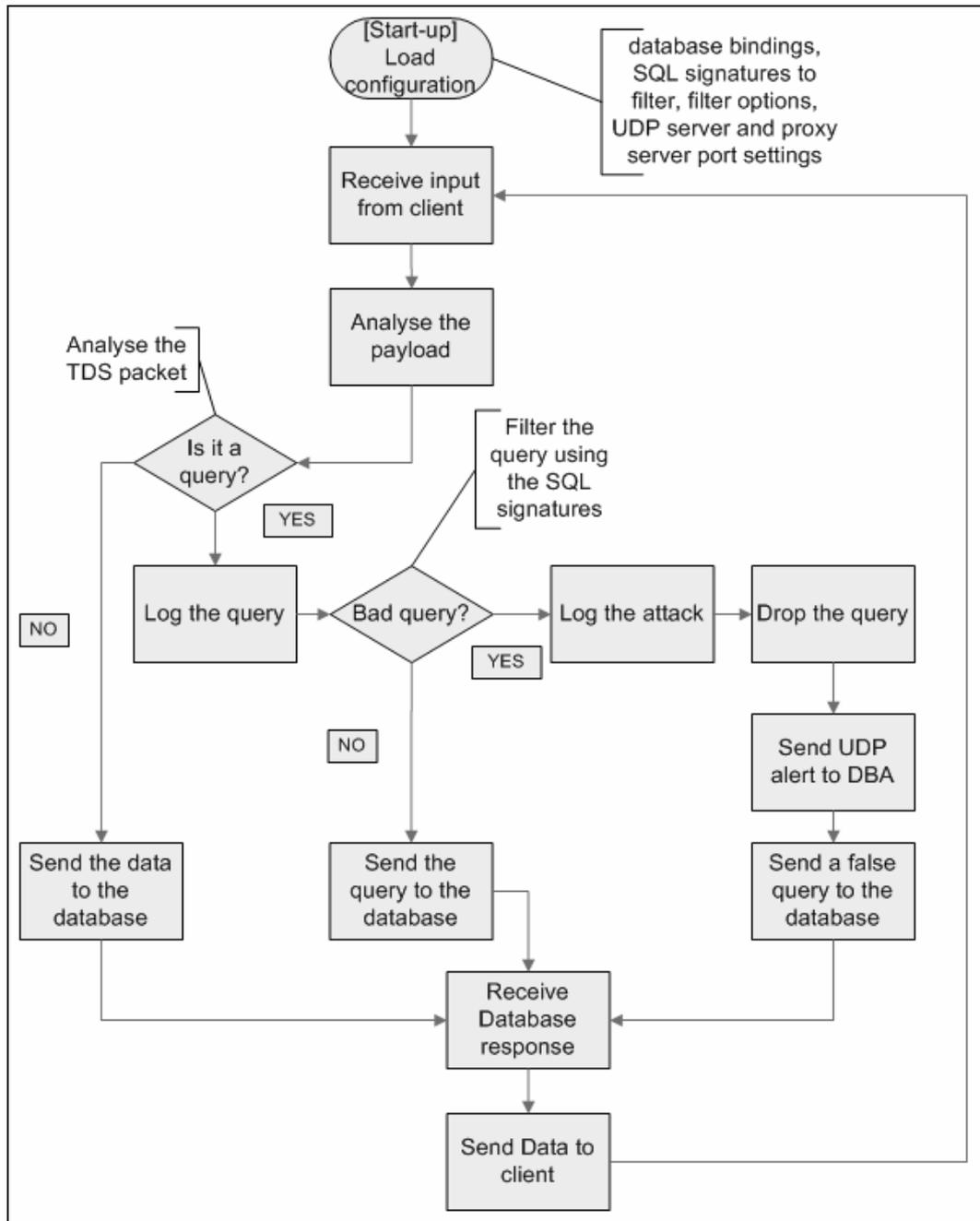


Figure 7: Flowchart of the TDSProxy server

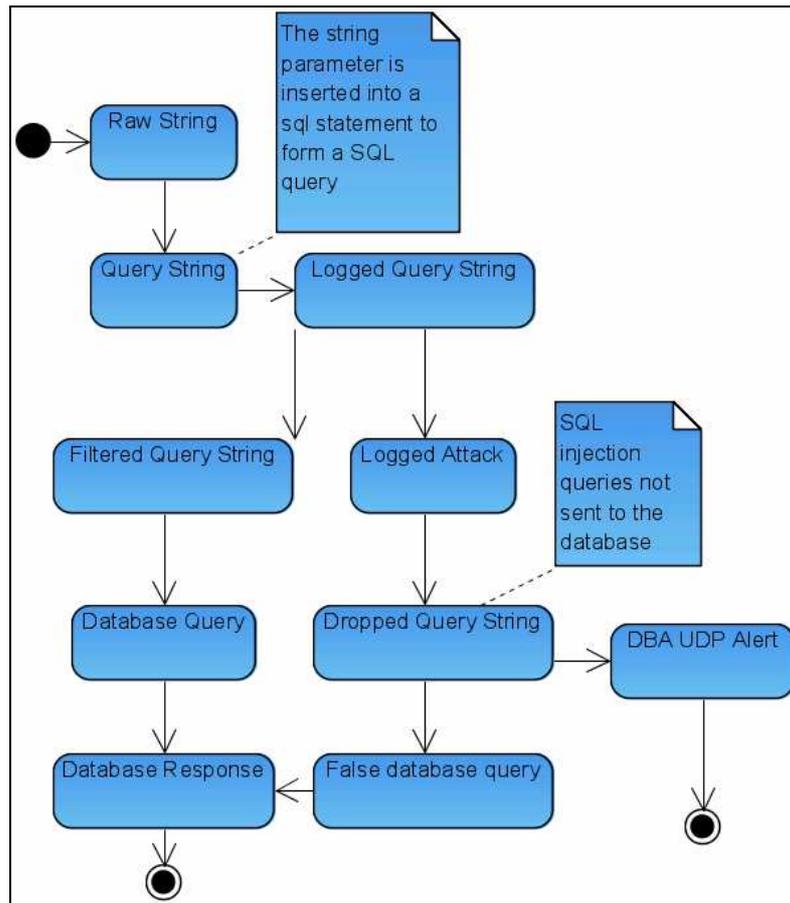
By following the flowchart in Figure 7, once the system has started, it is able to start receiving data from the client. When data is received from the client, the TCP payload is analysed for a TDS query packet. If the payload contains a SQL query, the query is extracted, logged and then filtered for any bad SQL commands. If the filter process finds that there is a potential attack, the attack is logged. After logging the attack, the attack information is sent via UDP to the DBA. The original query is discarded and a

false query is sent to the database. The response from the database is then returned to the client through TDSProxy.

If the filter process did not pick up an attack, the query is sent to the database and the database response is returned to the client. If the payload does not contain a TDS query packet, for example, it contains a login packet; the data is simply passed on to the database. The responses are then forwarded to the client.

The operations and methods of the system transform the query from one state to another depending on what route the information is flowing. These changes are shown in Figure 8. The various states depend on the route taken as illustrated in the flowchart in Figure 7.

The raw string becomes part of the query string through processing at the client interface. This happens when the input parameters are selected from the client interface and inserted into the hard coded query. The query may be formed at the client side or the parameters may be passed to the web application server. Once the SQL query has been formed, it is sent to TDSProxy where it is analysed for SQL injection. The query is logged and then filtered for SQL injection. If the query contains SQL injection, the attack is logged, the dangerous SQL is discarded, the DBA is notified via a UDP alert and a false query is sent to the database. The database response is then relayed to the client. If the filtered query does not contain SQL injection, the query becomes a database query and is sent to the database. The database response is then relayed the client interface through TDSProxy.



**Figure 8: State Change Diagram for Client Query**

### 3.3 Design conclusions

There are several advantages and disadvantages to producing TDSProxy.

➤ The advantages are:

- It is a standalone application independent of flaws in the client application coding and database privileges.
- TDSProxy can be deployed on a separate server, the database server or web application server.
- TDSProxy adds an extra layer of protection with real time analysis and prevention against SQL injection.

➤ Disadvantages include:

- False positives may be passed on to the database for execution if a hacker enters a valid malicious query that is not detected by the filter.
- False negatives may be filtered out when words from a valid clean query match words that are in the signature lists that block SQL injection.

- TDSProxy will not work if the data is encrypted. This is because of the use of a matching method with plain text signatures. This method checks if the query sent by the client matches signatures in the signature files.
- As the signature files increase, the filtering process may turn out to be resource intensive.

### **3.4 System Design Summary**

When a TDS query packet is received, the query string is analysed for possible a SQL injection attack. This is done by regular expressions checking for matches with SQL injection signatures. Bad queries are logged and an alert is sent to the Database Administrator. Only when the query has successfully passed through the filter is it sent to the database for processing. This means that there are no words in the query that are in the list of blocking signatures.

# Chapter 4 – System Implementation

## 4.1 Introduction

This chapter presents the implementation of TDSProxy by outlining the design decisions and methodology used. An important part of the implementation was the TDS protocol analysis which lead to the development of the filter used in TDSProxy. This part of the implementation is outlined in the section on testing and validation. The end of the chapter highlights a few of the problems encountered. After the development of the filter and creation of the signature lists used by the filter, some testing was done on the impact of the filtering process on web transactions. The results are presented in the form of graphs showing the average processing times and average throughput available during the transaction processing. The detailed data is available in Appendix D. The chapter concludes with conclusions regarding the implementation stage are drawn at the end of the chapter.

## 4.2 Implementation

The development languages were initially Java and Perl: Java is platform independent and Perl has powerful regular expressions capabilities.

In accordance with the guidelines by [Spolsky, 2000], the development language was changed to C# to allow for better code management and integration with Microsoft

Visual Source Safe and to link in with other Microsoft products; Microsoft SQL server 2000 and Microsoft Windows XP professional.

The operating system, development language and database platforms are all Microsoft based in order to prevent compatibility issues. Thus Windows XP Professional Edition, SQL Server 2000 Developer Edition and Visual Studio .NET 2003 along with Visual Source Safe version 6d [Spolsky, 2000] were used in the design and implementation phase. UML modelling made use of Visual Paradigm for UML 5.0 Enterprise Edition.

#### **4.2.1 Design Decisions**

The server will listen on the specified port number and the proxy server can be set to send the TDS packets to that port. According to the rfc on assigned numbers [Postel, 2004], port numbers ranging from 0 - 1023 because are restricted for well-known services such as HTTP and FTP. Therefore, the port numbers chosen for the development were 2222, 4444 and 5555.

Packets are not encrypted by default, but encryption can be enforced by the database. If secure socket layer (SSL) is used, the filtering process on TDSProxy will not work as SSL ensures that the data is encrypted. Filtering will work on plain text at this stage. Since SQL injection cannot be stopped using firewalls, intrusion detection systems or intrusion prevention systems, one should search for the following in all strings inputs from users in order to prevent SQL injection:

- ‘ “ / \; Strings in many programming languages are delimited by the double quote whilst in SQL, strings are delimited with a single quote.
- extended characters like NULL, carry return, new line;
- UNIONS which can be added to an existing statement to execute a second statement;
- SUBSELECTS which can be added to existing statements;
- INSERTS, UPDATES and DELETES
- Public system stored procedures

Since Microsoft SQL Server 2000 uses the Tabular Data Stream (TDS) protocol to communicate with its clients and TDS is carried by TCP, TDSProxy needs to handle TCP sockets and connections. This allows the proxy server to analyse the payload of TCP packets.

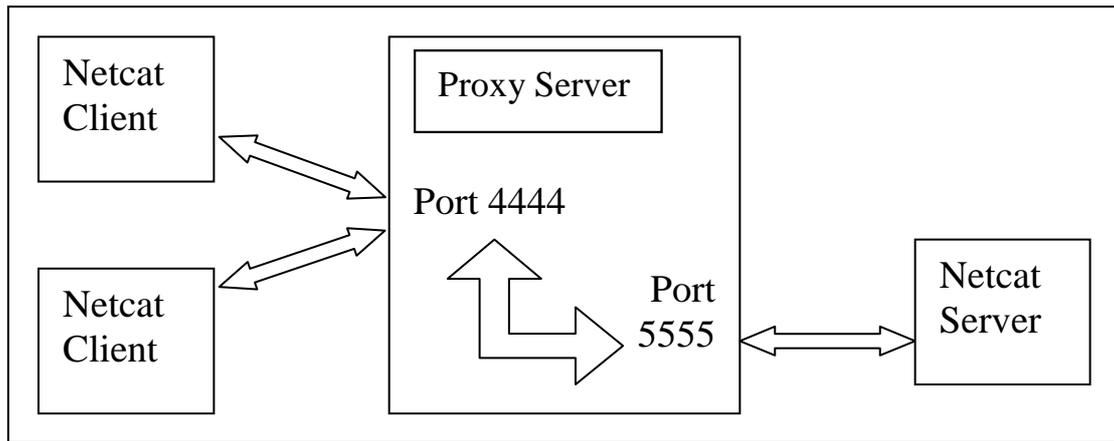
### **4.2.2 Methodology**

The implementation was done iteratively, starting off with an application that piped text (the TCP payload) through a proxy server. This was tested using a powerful networking tool called NetCat [Giacobbi, 2004]. This testing phase was aimed at creating the proxy server and ensuring it was transparent to the client application and database. After being able to pipe text, plain text queries were sent to the database from a NetCat client through the proxy server. This was followed by an attempt at using a data access page as a client. This was abandoned when a tool call OSQL was able to connect to the database through the proxy server. Once the proxy server was able to route information effectively, TDS query extraction and filtering was implemented.

### **4.2.3 Testing and validation**

Creating an echo client-server setup would aid the creation of a proxy server that would sit between the client and server. The initial application consisted of a knock-knock client-server that sent automated responses to each other based on the reply. This idea is an implementation of a common verbal game played by school children. Next, the client was extended to become the beginning of a proxy server that connected to a NetCat server. The server responses were typed in manually and the client application returned automated responses.

The next enabled a NetCat client to connect to the proxy server and send requests to the NetCat server through the proxy server. One NetCat client instance sent plain text to the port that the proxy server was listening on. The proxy server then rerouted the text to the port that the NetCat server it had connected to. The NetCat server was listening on. This is illustrated in Figure 9 below.



**Figure 9: Proxy server connecting NetCat clients to a NetCat server**

The purpose of this was to learn about the workings of a proxy server. One NetCat instance was set up as the client sending the query to the database whilst another NetCat instance simulated the database server receiving requests and sending responses to the client. The proxy server was placed between the NetCat client and server. All TCP traffic was logged to a text file. The payload of the TCP packets was plain text ASCII characters. This was a good exercise in learning about ports, TCP routing and the basic client server architecture.

Figure 10 to Figure 13 are screenshots of the proxy server connected to a NetCat server. The proxy server has two clients connected to it and the server is sending responses to each client. The order of start-up of the applications is important. The order should be database server, proxy server and finally the querying client. Once there is a link from the client to the server, transmission can begin. The value of this exercise was being able to create a proxy server that routed information from one port to another.

Figure 10 shows the proxy server connecting to the NetCat server. The proxy server was set to try and connect to port 5555. Port 5555 is the port that the NetCat client was going to listen on. Figure 10 also shows the connection of Client 0. Client 0 then transmits a message and receives a response. Client 1 connects, sends a message and receives a response. Both clients then disconnect, Client 1 first and then Client 0.

```

C:\ D:\project work\proxyServer01\TcpSock\bin\Debug\TcpSock.exe
This is the SQL proxy server filter: version 0.1
-----
Connecting.....
Connected to server
hons09: listening to port 4444
Client 0 connected.
Transmitting.....
Receiving.....
Client 1 connected.
Transmitting.....
Receiving.....
Client 1 disconnected.
Client 0 disconnected.

```

Figure 10: Proxy server listening on port 4444

Figure 11 shows the NetCat server being created. It listens on port 5555 and waits for connections. The proxy server connects once the NetCat server is ready. Once the proxy server is connected to the NetCat server, it listens on port 4444 for clients to connect to it. The NetCat server receiving a connection from the proxy server is also shown in Figure 11. Once connected to the proxy server, the proxy server routes messages sent to it from clients that have connected to it. Responses to the messages from the clients, received from the proxy server, are sent by typing responses in the NetCat server console. The responses are sent back to the client through the proxy server.

```

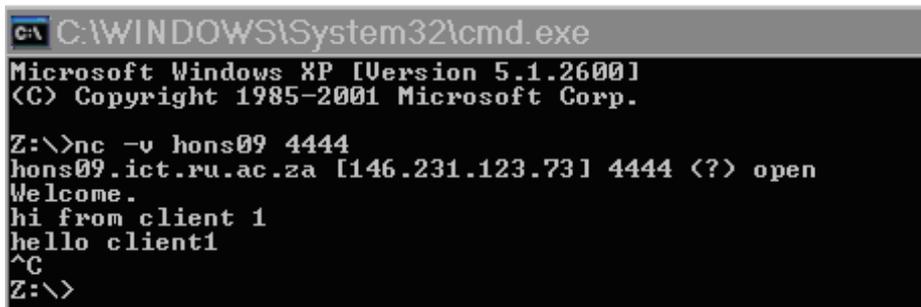
C:\WINDOWS\System32\cmd.exe - nc -v -L -p 5555
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

Z:\>nc -v -L -p 5555
listening on [any] 5555 ...
connect to [146.231.123.73] from hons09.ict.ru.ac.za [146.231.123.73] 1219
hi from client 1
hello client1
this is another client
hello another client

```

Figure 11: NetCat simulating a server listening on port 5555

Figures 12 and 13 show clients connecting to the proxy server. On connection they receive an automatic welcome message. The proxy server is now waiting for a response from the client that has just connected to it. Client details are kept in an ArrayList to keep track of which client sent which message. This information is used when sending replies to the client. The client sends a message to the proxy server. The message is routed to the NetCat server. A message is typed and sent in response to the client through the proxy server.



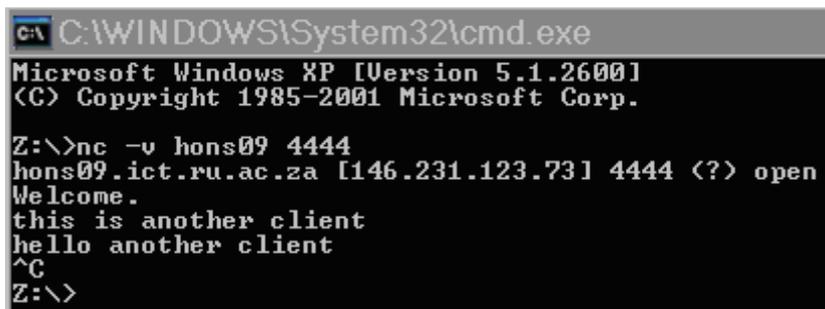
```

C:\WINDOWS\System32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

Z:\>nc -v hons09 4444
hons09.ict.ru.ac.za [146.231.123.73] 4444 (?) open
Welcome.
hi from client 1
hello client1
^C
Z:\>

```

Figure 12: NetCat simulating one client connected to the proxy



```

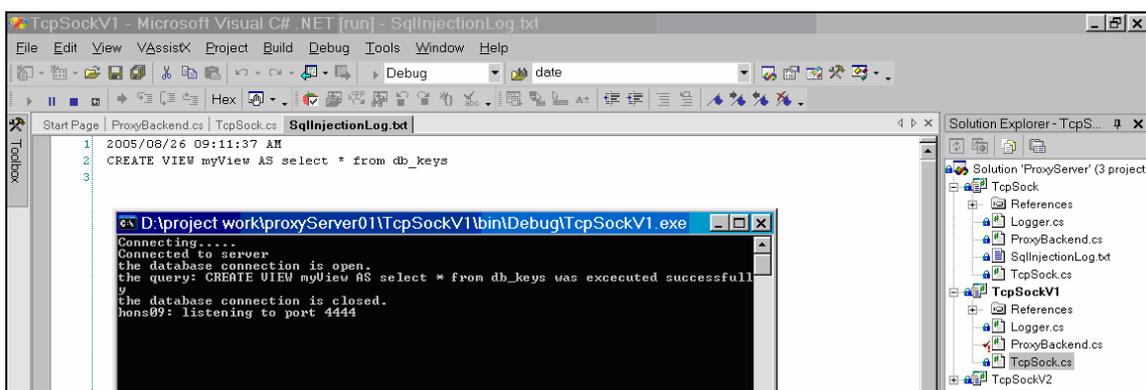
C:\WINDOWS\System32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

Z:\>nc -v hons09 4444
hons09.ict.ru.ac.za [146.231.123.73] 4444 (?) open
Welcome.
this is another client
hello another client
^C
Z:\>

```

Figure 13: NetCat simulating a second client connected to the proxy

The proxy server was then improved to connect to the database using a connection string. There was a problem at this stage of the development. Initially the problem was thought to be in the code, however on further investigation it was found that a database setting had been causing the problems. With the correct username, password and rights, the database was manipulated by entering the SQL text on a NetCat client instance.



```

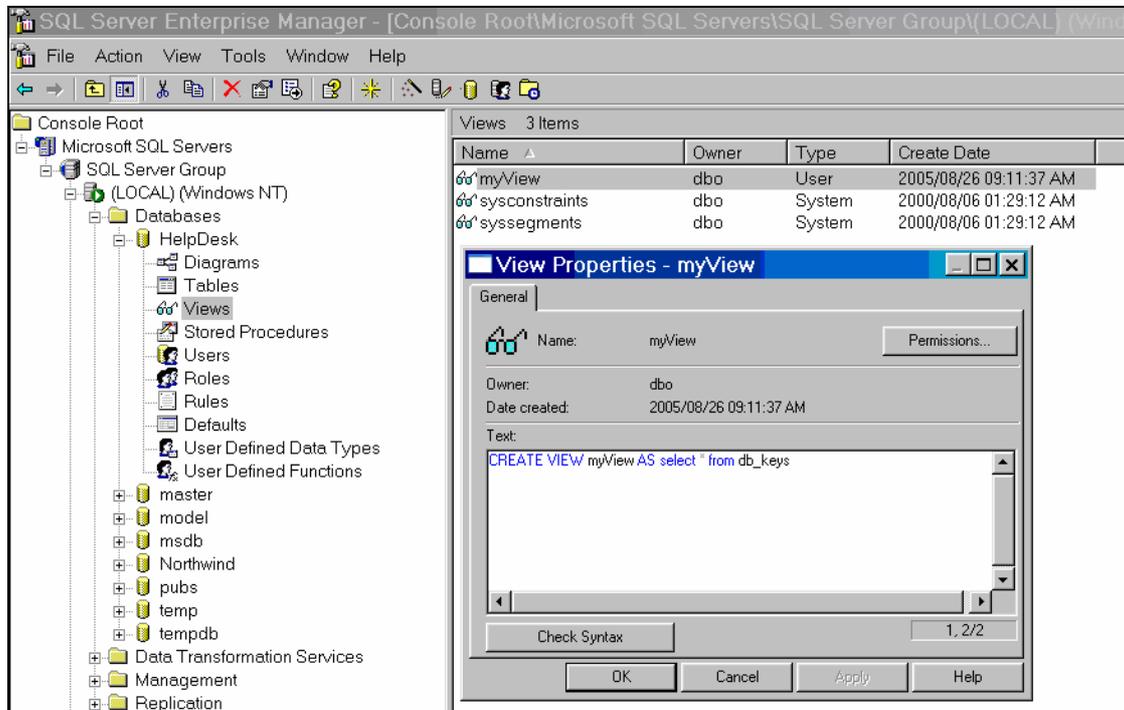
D:\project\work\proxyServer01\TcpSockV1\bin\Debug\TcpSockV1.exe
Connecting...
Connected to server
the database connection is open.
the query: CREATE VIEW myView AS select * from db_keys was executed successfully
the database connection is closed.
hons09: listening to port 4444

```

Figure 14: Proxy server code setting up a view before connecting to the database

The next step involved sending hard coded SQL queries to the database at start-up of the application. This confirmed that the username, password and privileges were correct. This is illustrated in Figure 14 where a view was created in the database by

sending `CREATE VIEW myView AS select * from db_keys` as the query string to the database. The proxy server is shown in the foreground and the background shows the log file that was generated. Figure 15 shows the result of the query as shown in the database.



**Figure 15: SQL server database view created via the proxy server**

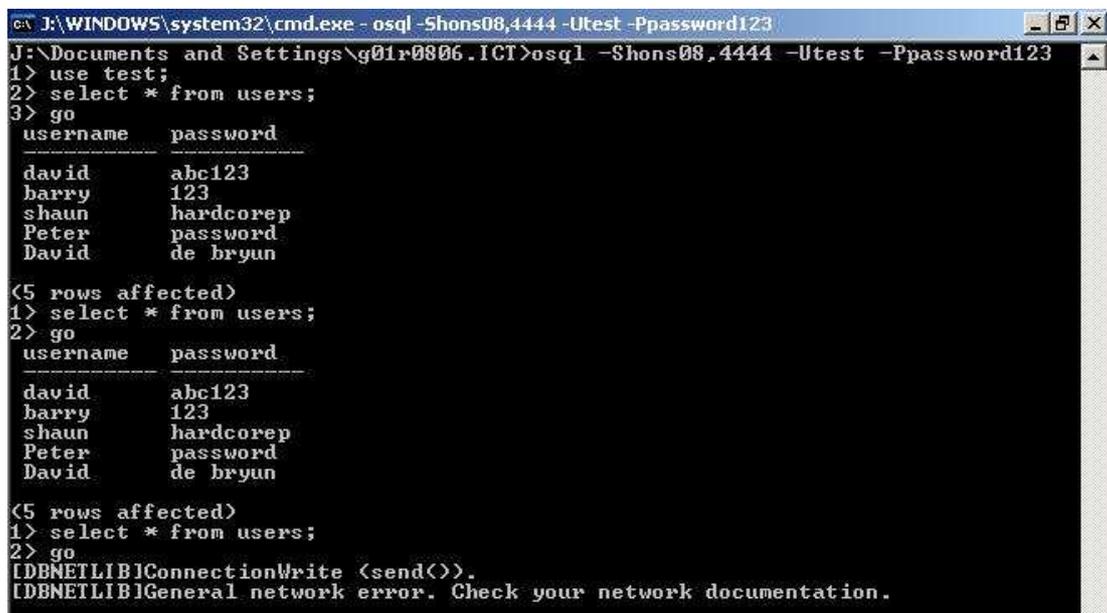
#### 4.2.4 Protocol Analysis

The querying client made use of OSQL, a tool that comes with MS SQL Server 2000. Figure 13 shows the use of OSQL to query the database through the proxy server listening on port 4444. OSQL is able to connect to the database by specifying the machine name, port number, username and password.

OSQL was used to successfully query the test database and select all information from the users table. When the proxy server was shut down, there was no link to the database and a network error was returned as shown in Figure 16. The proxy server, OSQL tool and the database are all sitting on the same machine. The result of Figure 13 shows that the connection to the database on the local machine was made through the database. It should be noted that without specifying `'-S hons08, 4444'`, OSQL would connect to

the local database directly using the default port. This is undesirable as the queries must be sent through the proxy server in order for the queries to be filtered.

The use of OSQL, along with packet sniffers Ethereal [Ethereal, 2005a] and Packetyser [Network Chemistry, 2005] allowed for the development of the SQL extracting method. This was done by analysis of the TDS protocol [FreeTDS, 2005] and lead to the extraction of the query in the query packet sent to the database after the login challenge [Bruns, Wheeler, Schaal, Ziglio et al., 2005]. Figure 16 shows the use of OSQL to query the database through TDSProxy. After the second query has been sent, TDSProxy was stopped. With the third query attempt using OSQL, there are two errors (shown in Figure 16) due to no connectivity between OSQL and the database.



```

cmd J:\WINDOWS\system32\cmd.exe - osql -Shons08,4444 -Utest -Ppassword123
J:\Documents and Settings\g01r0806.ICT>osql -Shons08,4444 -Utest -Ppassword123
1> use test;
2> select * from users;
3> go
username      password
-----
david         abc123
barry        123
shaun       hardcorep
Peter       password
David      de bryun

(5 rows affected)
1> select * from users;
2> go
username      password
-----
david         abc123
barry        123
shaun       hardcorep
Peter       password
David      de bryun

(5 rows affected)
1> select * from users;
2> go
[DBNETLIB]ConnectionWrite (send()).
[DBNETLIB]General network error. Check your network documentation.

```

Figure 16: Use of OSQL and error message on shutdown of proxy server

Figure 17 shows a comparison of the extracted TDS packet data using TDSProxy and a packet capture using Ethereal. The packet sizes are the same and so are the source and destination names (Figure 17). The captured data shows a connection from hons09 to a database sitting on Netserv. Netserv's IP address is 146.231.131.136 and the database is listening on the default port 1433. The logon request comes from hons09.ict.ru.ac.za port 2222 and this is shown in the packet capture with Ethereal and TDSProxy. There is no difference between the packet capture information on both Ethereal and TDSProxy, showing that TDSProxy is capturing all the data effectively.

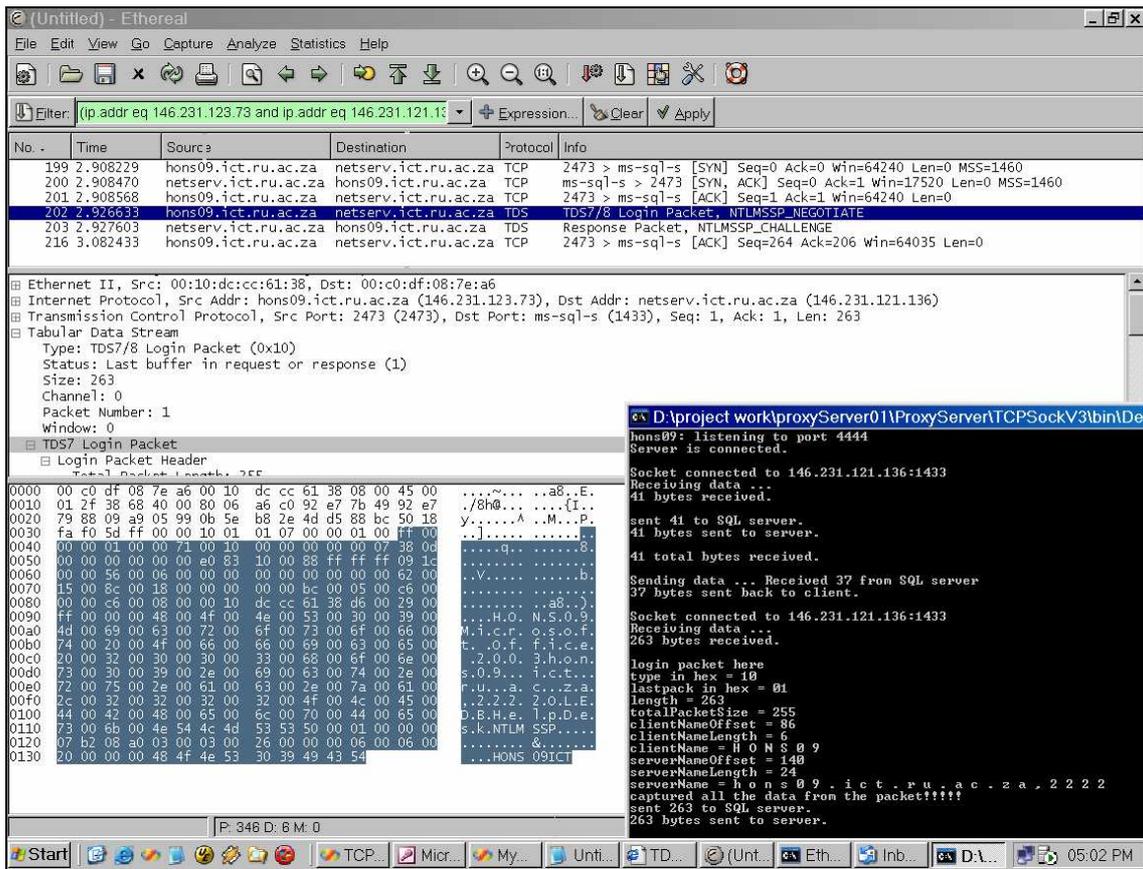


Figure 17: Ethereal packet capture compared to proxy server packet analysis

Ethereal does not support packet capture on the loop back on Windows [Ethereal, 2005b]. Since the Microsoft loop back adaptor on Windows XP is not very effective in capturing the data sent between OSQL and MS SQL server 2000 for example, the request was sent to Netserv, a remote machine, making an Ethereal packet capture possible. By sending the data to another machine, a comparison could be made between an Ethereal packet capture and the payload extracted by TDSProxy. This is illustrated in Figure 18 which shows the packet capture using Ethereal as well as the payload capture using TDSProxy. This was helpful during the creation of the SQL extraction methods. The proxy server logs the traffic and is able to manipulate the packets sent to the database as well as capture the SQL statements send to the database.

Figure 18 shows the typical usage sequences for the TDS protocol as used by Sybase SQL Server and Microsoft SQL Server. Figure 19 and Table 2 show the structure of TDS packets.

```

--> Login
<-- Login acknowledgement

--> INSERT SQL statement
<-- Result Set Done

--> SELECT SQL statement
<-- Column Names
<-- Column Info
<-- Row Result
<-- Row Result
<-- Result Set Done
    
```

Figure 18: Typical Usage sequences for TDS [FreeTDS, 2005]

Packet type	Description
0x01	TDS 4.2 or 7.0 query
0x02	TDS 4.2 or 5.0 login packet
0x03	RPC
0x04	responses from server
0x06	Cancels
0x07	Used in Bulk
0x0F TDS	5.0 query
0x10 TDS 7.0	login packet

Table 2: TDS Packet Types and their descriptions [FreeTDS, 2005]

INT8	INT8	INT16	4 Bytes
Packet type	Last packet indicator	Packet size	Unknown

Figure 19: Packet format of all TDS packets [FreeTDS, 2005]

During the development stages, TDSProxy made use of some of the code by [Kocak, 2004] in his application called Pacanal. Pacanal is a C sharp attempt at producing Ethereal-like capabilities and currently supports up to fifteen protocols. The `PacketSQL` code in Pacanal [Kocak, 2004] and the capture shown in Figure 19 proved to be very useful in understanding the TDS protocol and how to extract the query string from the TDS query packet. Figure 20 shows code used to extract the query string from the TDS query packet. The variable `msg` is a byte array containing the TCP payload. If the packet

being captured is a TDS query packet as indicated by the first byte of the packet being 0x01.

```
157 #region if (query packet), filter first
158 if(FILTER && msg[0]==1){
159
160 #region extract the sql from the packet
161 int index = 8;//the sql starts at msg[8]
162 //there are spaces in between each character representation
163 //so we need to move along to skip that out.
164 sqlString = "";//prevent concatenation to previous query
165 for( int i = 0; i < (size - 8)/2 ; i++ )
166 {
167     sqlString += (char) msg[ index++ ];
168     index++;
169 }
170
171 #endregion extract the sql from the packet
```

**Figure 20: Code showing the extraction of the query from TDS query packet**

The white paper by [Anley, 2002a] covers research into SQL injection as it applies to Microsoft Internet Information Server/Active Server Pages/ MS SQL Server platform. It addresses some of the data validation and database lockdown issues that are related to SQL injection into applications. The paper provides examples of SQL injection attacks and gives some insight into .asp login code and query error messages used to exploit databases.

With TDSProxy now able to capture the query sent in the TDS query packet, a vulnerable ASP application was developed [Anley, 2002a]. It comprises of an asp login page that does a comparison of username and password with rows in the database. If there is a match, the user is taken to an “Access Granted” page. Otherwise, the user is taken to an “Access Denied” page. The ASP page was hosted on a remote machine and connection to the database came through the proxy server by manipulating the connection string to connect to the machine and port that the proxy server was listening on. This ASP application allows the user to enter SQL injection text into the input parameters and manipulate the database.

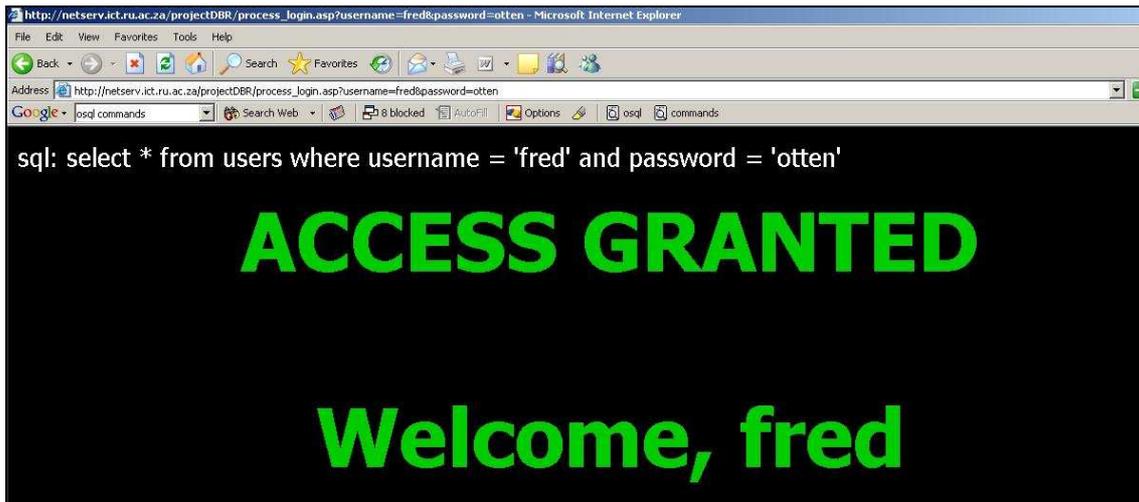


Figure 21: Successful SQL injection in the database

Entering the correct parameters present the user with an “Access granted” page (Figure 21). The query sent to the databases is shown at the top of the page as `select * from users where username = '<username parameter>' and password = '<password parameter>'`. There is a row in the database where the username is “fred” and the password is “otten”.

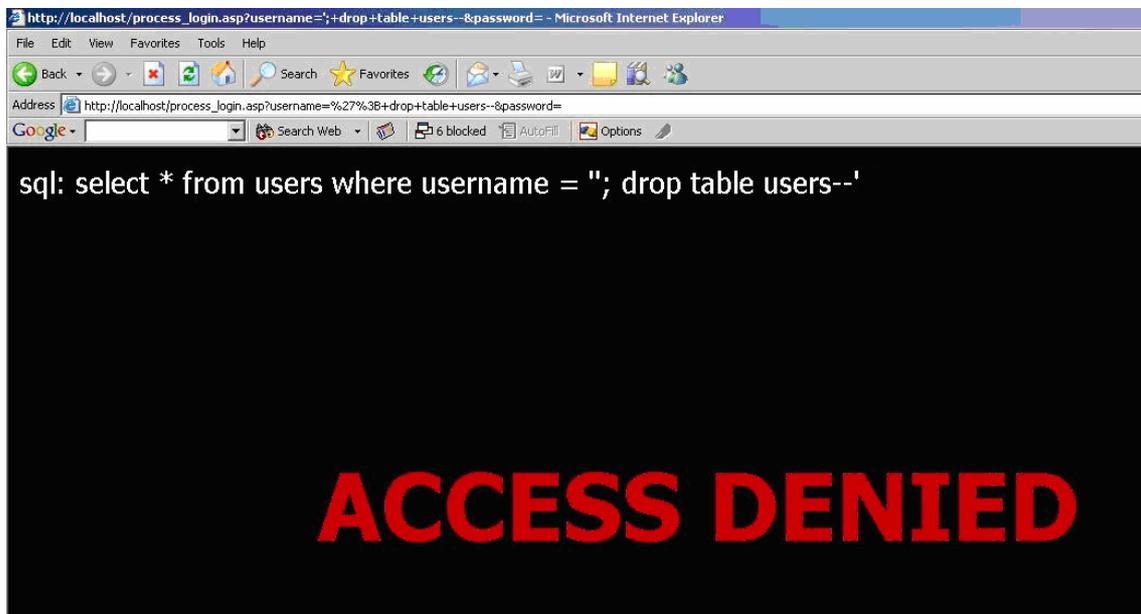


Figure 22: Dropping a table using SQL injection

Figure 22 shows the result of entering `'';drop table users--` as an input parameter for username. The user is presented with an “Access denied” page because there is no

matching username `'';drop table users--` in the database. In this example, the two SQL queries formed are shown in Text Box 4.

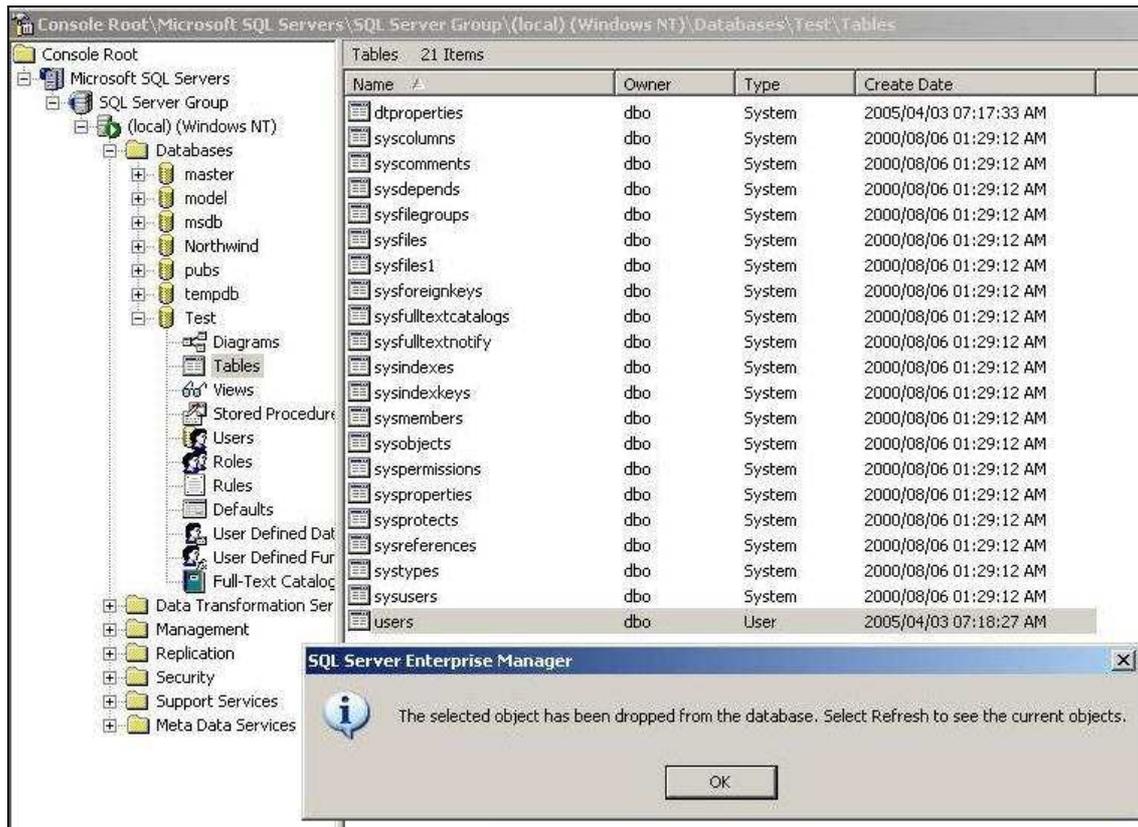
```
1. select * from users where username = ''';
2. drop table users--
```

**Text Box 4: SQL injected queries executed by the database**

The first statement looks for a row in the table where the username is a double quote (“). This statement is terminated by a semi colon. The next SQL query drops the users table. The double dash (--) indicates that the rest of the statement is a comment and can be ignored by the database. The problems with this scenario are:

- There is no input validation and so SQL injection is possible with the use of dynamic SQL.
- The user being used to connect to the database has too many privileges because they are allowed to execute a *drop* statement. A user who owns the object or has database administrator (DBA) access can execute a drop statement.

Figure 23 shows the result of this SQL injection as seen from the database. The database has executed the drop command and the users table has been dropped from the database. This action was performed while Enterprise Manager was open. On trying to access the dropped table, the database returned the error shown in Figure 19. Upon refreshing the database, the view did not contain the users table.



**Figure 23: Database view of the dropped table**

A regular expression is an expression that concisely describes a set of characters without having to list the whole set they describe. However, there are multiple patterns that can describe a single set of characters. Regular expressions are used in pattern matching thus the next step involved creating the filter by making use of powerful regular expressions.

The filter method made use of black, white, gray and pattern matching signatures. A black list contains lists of strings that are black listed. These strings are the signatures that are considered bad. When the filter comes across queries containing strings that match those in the black list, they are filtered out.

White lists, on the other hand are the exact opposite of black lists. White lists show strings that are allowed. In between white lists and black lists are gray lists. They contain strings that have the potential to be bad but may also be good. Thus the action taken when a gray list is matched is alerting the DBA. The query is not halted but is

allowed to pass through. This option can be changed to a pessimistic mode where gray lists act as black lists.

The regex list contains a list of pattern matching regular expressions. They are more generic and can be used as input validations e.g. allow only ASCII alpha characters to pass. Some of the strings contained in the regex list were built using a tool called regex coach [Weitz, E 2005].

According to [Hoglund and McGraw, 2004], the problem with black lists being used as a filter method to remove bad input is the creation and maintenance. An exhaustive list is difficult to produce at best and mistakes in the black listing make the attacker's job easier. Therefore a much better approach is to use a white list approach, specifying which input patterns should be allowed. This is a version of the principle of least privilege which gives your program only as much power as it needs and no more. In light of this, a decision was made to use a combination of lists. The filter uses SQL injection signatures which are made up of a black list, white list, gray list and pattern matching list. The filter is able to report whether the SQL query text matches any of the given signatures.

At all stages of development, there is extensive logging of the queries captured. This helps with the debugging. SQL injection attacks are logged along with the signature that caught the attack. With the aid of the log files generated by TDSProxy and the database log files, the DBA can ascertain which database is being attacked. The DBA can also determine which web server or web page the attacker is using. The value of this is that the security holes can be patched and the database protected from further attacks.

Alerts are sent via UDP to the database administrator with the SQL injection query, the name of the machine hosting the web application and a timestamp. This will allow the DBA to block further injection attacks from a particular user by checking the database log file which should contain the IP address of the person who sent the query at that time.

## 4.2.5 Problems Encountered

The default setup configuration of MS SQL Server 2000 allows Windows authentication only. This needs to be changed to Windows and SQL Server authentication; otherwise it will result in an error (TextBox5).

Login failed for 'helpdesk'. Reason: not associated with a trusted SQL server connection.

### Text Box 5: Login error

During the testing stages, getting the proxy server to just route information was not enough. There has to be some changing of the packet data variables. There seems to be a need to change things like the destination machine, port numbers and the like in order for the applications to accept the Transmission Control Protocol (TCP) and Tabular Data Stream (TDS) packets.

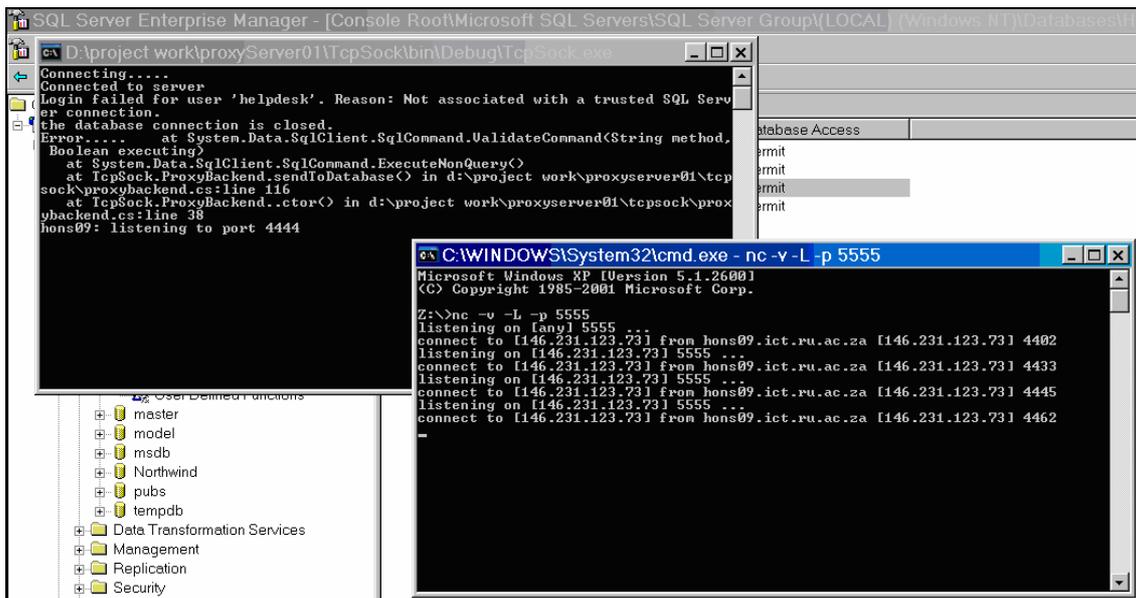


Figure 24: Login error – not a trusted SQL server connection

An attempt at using a Microsoft Access 2003 data access page as the client was unsuccessful and produced many login errors. When setting up the data access page, Access 2003 only accepted the use of an actual machine name and not its IP address. For testing purposes, a direct connection to the database was set up and querying the database through the data access page was possible.

The next step was to be able to route the login through TDSProxy so that the database would ‘think’ it was talking to an Access data access page. However, when trying to connect to the database from the data access page through the proxy server, there was a problem with the connection string. The database kept returning an error message saying that the connection was refused because it was not associated with a trusted database. This problem, shown in Figure 20, was overcome by hard coding “trusted\_server = true” into the data access page’s connection string in the htm page. This was done by editing the htm file in a text editor as shown in Code Box 9.

The login errors continued and the database kept sending back reset packets. There was no apparent reason for it not being able to log on after the data was being routed. The packet data was altered so that the source and destination ports and IP addresses made TDSProxy seem totally transparent. The first three login packets were forged from a successful login without TDSProxy. However, this made no difference and an alternative client tool was sought. The possibility of port or IP number mismatching was eliminated by continuing the development on the same machine.

```
<a:ConnectionString>
Provider=SQLOLEDB.1;trusted_server = true;Password=password123;Persist
Security Info=True;User ID=test;Initial Catalog=Test;Data
Source=hons08.ict.ru.ac.za,4444;Use Procedure for Prepare=1;Auto
Translate=True;Packet Size=4096;Workstation ID=HONS08;Use Encryption
for Data=False;Tag with column collation when possible=False
</a:ConnectionString>
```

**Code Box 9: Editing the connection string to stipulate a trusted connection**

With the proxy server still in place and routing the data to the SQL database, there was a problem with logging into the database. The database kept sending back reset packets. There was no apparent reason for it not being able to log on after the data was being routed. Figure 25 shows an Ethereal packet capture of a successful login without using the proxy server, there are two packets sent before a login packet is sent to the database. These are part of the TCP three-way handshake.

No. -	Time	Source	Destination	Protocol	Info
199	2.908229	146.231.123.73	146.231.121.136	TCP	2473 > ms-sql-s [SYN] Seq=0 Ack=0 win=64240 Len=0 MSS=1460
200	2.908470	146.231.121.136	146.231.123.73	TCP	ms-sql-s > 2473 [SYN, ACK] Seq=0 Ack=1 win=17520 Len=0 MSS=1460
201	2.908568	146.231.123.73	146.231.121.136	TCP	2473 > ms-sql-s [ACK] Seq=1 Ack=1 win=64240 Len=0
202	2.926633	146.231.123.73	146.231.121.136	TDS	TDS7/8 Login Packet, NTLMSSP_NEGOTIATE
203	2.927603	146.231.121.136	146.231.123.73	TDS	Response Packet, NTLMSSP_CHALLENGE
216	3.082433	146.231.123.73	146.231.121.136	TCP	2473 > ms-sql-s [ACK] Seq=264 Ack=206 win=64035 Len=0

**Figure 25: Packet capture showing a successful login directly to the database**

The first three packets were forged to look like the original TDS packets that were sent when there was a successful login with no proxy server in place. However, reset packets were sent back to the Access data access page at what would have ordinarily been the logon stage. Figure 26 shows the first three packets being sent to the SQL server after which, there seems to be no apparent communication between the data access page and the SQL Server. The first two are part of the synchronisation between the server and data access page. The third packet is a TDS7/8 login packet.

After extensive research into the TDS packet structure along with analysis of Ethereal packet captures, it was concluded that there may be a problem with the port numbers or source and destination mismatches. These problems were eliminated by running everything on the same machine. Thus there was no need to alter the machine address. This left only the port as the problem. The proxy server was modified to be able to change the port numbers. The source packet's destination was changed from the proxy server port to the port that the Microsoft SQL Server was listening on. This made it look like the packet was sent from Microsoft Access directly to the Microsoft SQL Server. The process was reversed on receiving a response from the server. However, this did not have the desired effect and the login packet was not accepted.

```

D:\project work\proxyServer01\ProxyServer\TCPSockV3\bin\Debug\TC...
hons09: listening to port 1433
Server is connected.

Receiving data ... Socket connected to 146.231.121.136:1433
41 bytes received.

sent 41 to SQL server.
41 bytes sent to server.

41 total bytes received.

Sending data ... Received 37 from SQL server
37 bytes sent back to client.

Receiving data ... Socket connected to 146.231.121.136:1433
1676 bytes received.

sent 1676 to SQL server.
1676 bytes sent to server.

1717 total bytes received.

Sending data ... Received 156 from SQL server
156 bytes sent back to client.

Receiving data ... Socket connected to 146.231.121.136:1433
0 bytes received.

sent 0 to SQL server.
0 bytes sent to server.

1717 total bytes received.

Sending data ... SocketException : System.Net.Sockets.SocketException: An existi
ng connection was forcibly closed by the remote host
   at System.Net.Sockets.Socket.Receive(Byte[] buffer, Int32 offset, Int32 size,
   SocketFlags socketFlags)
   at System.Net.Sockets.Socket.Receive(Byte[] buffer)
   at TCPSockU3.ProxyBackend.receive(Int32& size) in d:\project work\proxyserver
01\proxyserver\tcpsockv3\proxybackend.cs:line 60
0 bytes sent back to client.

Receiving data ... Socket connected to 146.231.121.136:1433
0 bytes received.

sent 0 to SQL server.
0 bytes sent to server.

1717 total bytes received.

Sending data ... SocketException : System.Net.Sockets.SocketException: An existi
ng connection was forcibly closed by the remote host
   at System.Net.Sockets.Socket.Receive(Byte[] buffer, Int32 offset, Int32 size,
   SocketFlags socketFlags)
   at System.Net.Sockets.Socket.Receive(Byte[] buffer)
   at TCPSockU3.ProxyBackend.receive(Int32& size) in d:\project work\proxyserver
01\proxyserver\tcpsockv3\proxybackend.cs:line 60
0 bytes sent back to client.

Receiving data ... Socket connected to 146.231.121.136:1433
0 bytes received.

sent 0 to SQL server.
0 bytes sent to server.

```

Figure 26: Attempted login from data access page through proxy server

These problems were solved by using a MS SQL console tool called OSQL. Using this tool, one is able to log into and query the remote database via the proxy server. The query was routed through the proxy server to the database and this allowed for the capturing of the SQL query. All the query information from logon to finish can be captured, logged and manipulated.

Later on in the development stage, the error was found to be the size of the buffer being used. As can be seen from Code Box 9, the size of the buffer should be at least 4096 bytes. The buffer being used was not big enough to hold all the data being sent to the database and so it kept sending reset packets back to the client.

#### **4.2.6 Web transaction tests**

Web transaction tests were carried out and the results of these indicate the effect of TDSProxy on web transactions. The tests were carried out on the local host (hons08) where the database and TDSProxy were running. The tests were then repeated on Netserv, a remote machine with TDSProxy and the database sitting on the hons08. A summary of the results available in appendix D are shown below in the graphs below.

There is no substantial increase in web transaction processing time when comparing a direct connection from the web application to the database and a connection from the web application through TDSProxy without filtering. This is true for the cases of the web application tests being done on both Netserv and hons08. Figure 27 and Figure 28 both show the first two scenarios' average time is around 5 ms per query.

There is a jump in the average web application processing time when the filter is turned on. The increase of about 20 ms per query in both Figure 27 and Figure 28 indicate that the increase in the average web application processing time is due to the filtering process. The slight increase in average web application processing time when doing the tests on Netserv is attributed to the decrease in throughput due to network traffic. This can be seen in Figure 29.

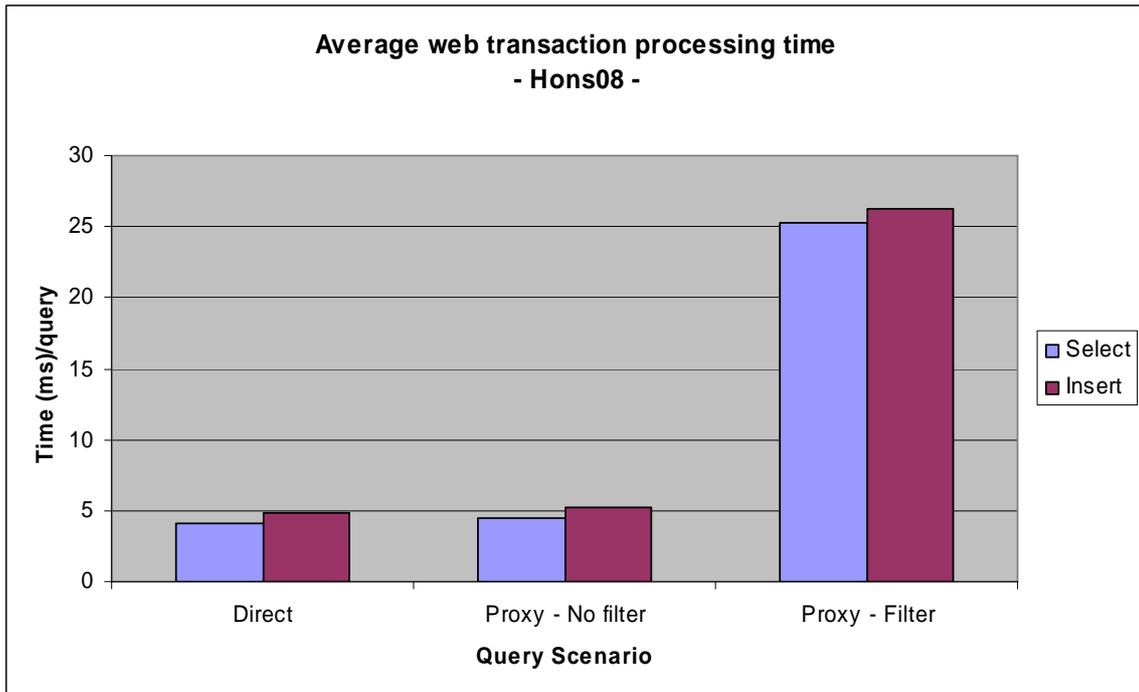


Figure 27: The average web transaction processing time on hons08

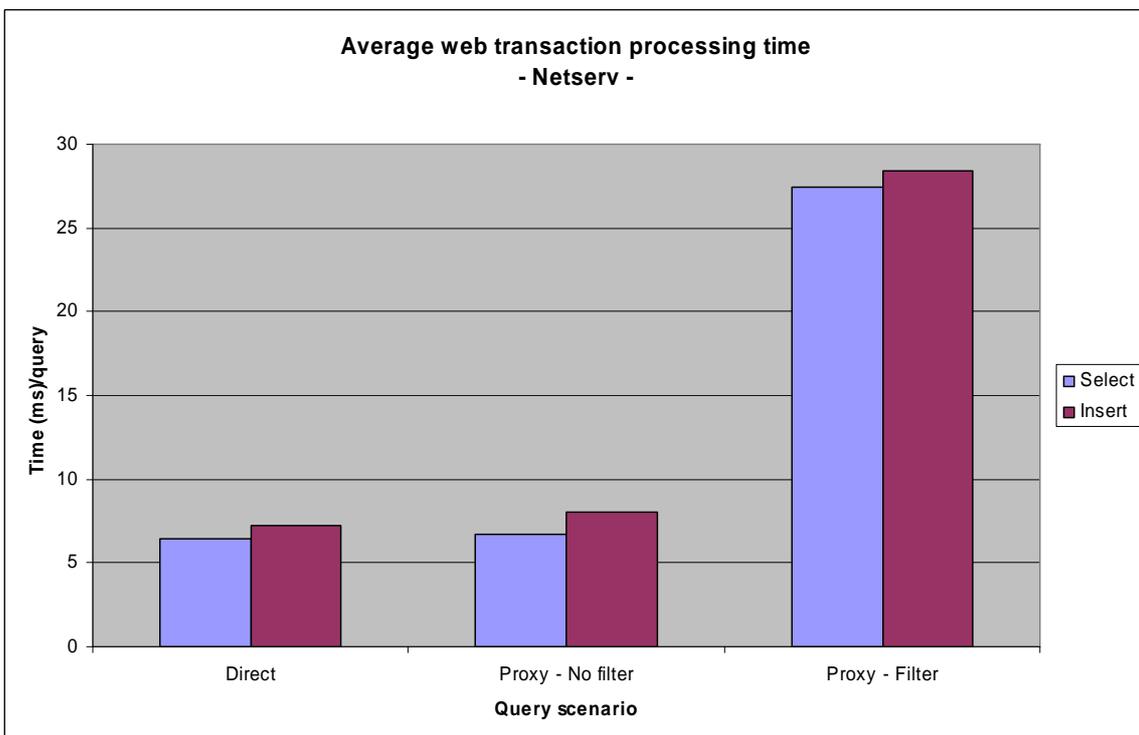


Figure 28: The average web transaction processing time on Netserv

Figure 29 shows a decrease in the throughput when the test is done on a remote server as opposed to doing the tests on the local host. The average INSERT statement seems to

take longer to execute than selecting 100 records from the database. This accounts for the decrease in throughput from selects to inserts along the same route – directly to the database, through TDSProxy without filtering or through TDSProxy and its filter.

The effect of the decrease in throughput between SELECT and INSERT statements was reduced by the network communication as the test scenario changed from Hons08 to Netserv. This is because the traffic spent more on the network whilst travelling to and from the database. The effect described is highlighted by Figure 30 which shows that there is an increase in the average processing time for TDSProxy. This is because TDSProxy may still be waiting for all the traffic to arrive before it can process the query.

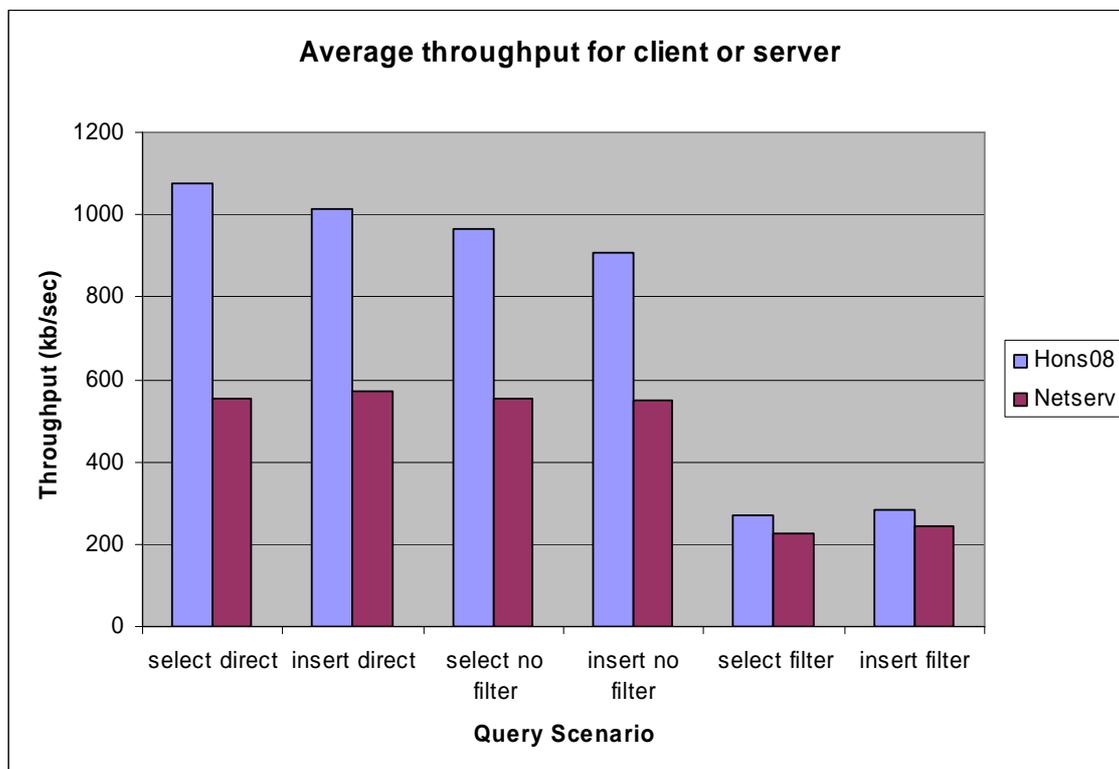


Figure 29: Graph showing the average web throughput for the client or server

Another test scenario made use of the OSQL tool's ability to run scripts from files. When filtering was turned off, the average processing time of TDSProxy was reduced from 0.256845 milliseconds to 0.002469 milliseconds for a set of 5000 queries of varied length and structure. Thus the filter process adds a 1% increase in average processing time per query for a total signature set is 190.

Timing the latency of TDSProxy was done by subtracting the time that the database spends processing the query from the roundtrip time for a client query and response. The roundtrip time was calculated as the query enters and leaves the proxy sever on the client side only. The database processing time was calculated by timing the query and response time on the database side.

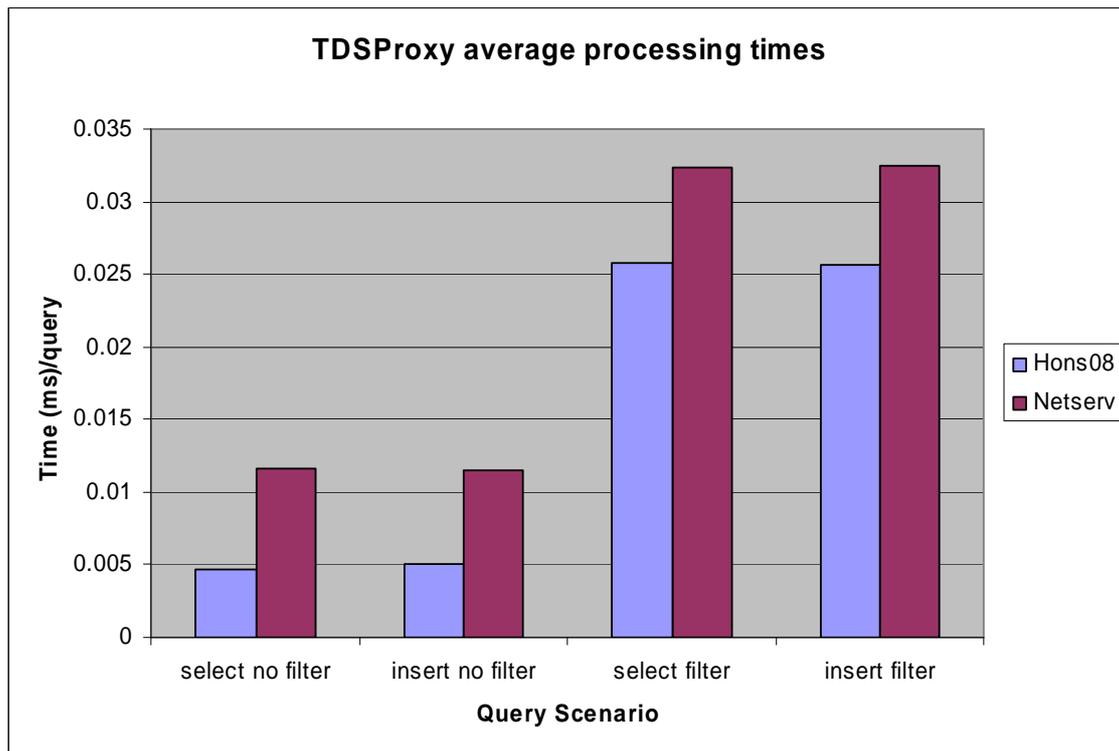


Figure 30: Graph showing the average processing time of TDSProxy

### 4.2.7 Conclusions

The time taken to process queries seems to be negligible given the default MS SQL Server 2000 login timeout time is 4 seconds and the default query timeout time is 0 seconds. However, it is predictable that as the SQL injection signature set grows, there may be an effect due to the filtering process.

### 4.3 System Implementation Summary

In this chapter, we provided an explanation of the implementation and discussed the problems encountered. The solution to the problem was developed iteratively. The final

solution was developed in a fully Microsoft environment. Microsoft SQL Server 2000 was configured to listen on port 5555. TDSProxy routed TCP data from port 4444 to port 5555. Thus, any application making use of TDSProxy to connect to the database had to connect to port 4444. The port and address settings are maintained in the configuration file as are the filter settings.

The filter makes use of SQL signatures that are maintained in separate files. There are black, white, gray and pattern matching lists and these lists may be updated as more matching signatures are identified. The lists are used by the filter method when analysing the TDS payload for a SQL injection attack. Only clean queries are sent to the database. If an attack is detected, an alert is sent out via UDP to the Administrators and the attack time and query is logged.

The resulting software filters out SQL injection attack queries. The filter process is fast enough to make TDSProxy seem invisible with the maximum processing time being about one fortieth of a millisecond per query.

# Chapter 5 - Conclusion

## 5.1 Conclusion

SQL injection takes advantage of application flaws to execute additional queries on a database when parameters are chained together to create the database query and there is no parameter input validation. Most web applications are vulnerable to SQL injection or some form of hacker attack and it is common knowledge that many more vulnerable applications will be developed. There are many measures that can be implemented to reduce the chance of an attack. For instance, it is possible to ensure input validation, check database error log files and reduce the permissions of all application users in the database itself. However this is tedious and is often not put into practice by many developers who may need to produce software in a hurry.

With the increase in awareness of SQL injection flaws and the availability of automated vulnerability scanners, filtering malicious SQL statements seems to be the best solution in preventing SQL injection. Therefore, the project produced a filtering proxy server to prevent SQL injection. Ideally, the filtering application should sit on the same machine as the database. However, the filtering process may have a performance impact such that the filtering application and the database need to be on different machines.

Using SQL signature filtering as a preventative measure to SQL injection provides real time protection against SQL injection. TDSProxy is an autonomous application independent of flaws in the vulnerable application and independent of database configuration flaws. However, false positives may be allowed through the filter and false negatives may be blocked by the filter. This is due to the difficulty in creating and maintaining an exhaustive list of SQL injection signatures despite there being a finite

set of words in the SQL vocabulary. Black, white, gray and pattern matching lists are maintained in separate files and are used by the filter method when analysing the TDS packet for a SQL injection attack. However, going back to the principle of least privilege, by using a white list, it is possible to define what is allowed and thus prevent invalid signatures.

The design methodology made use of UML in an iterative process along with the implementation. Thus the development was done in stages, allowing problems to be tackled in manageable pieces. The final solution was developed iteratively in a fully Microsoft environment.

Timing the web transaction processing time per query provided useful information on the impact of the TDSProxy on web interface usage. The filter process is fast enough to make TDSProxy seem invisible. However, it is predictable that as the SQL injection signature set grows, there may be an effect due to the filtering process.

SQL injection will be around for a long time and the methods of defence other than correct input validation coding will only hinder the chances of an attack. Ways to overcome the obstacles are left up to the creative hacker to discover.

The project poster is available in Appendix A gives a good visual summary of the project. A code overview is available in Appendix C. All project code, documentation, references and software used are available on CD in the Rhodes University Computer Science Department. The CD contents are described in Appendix B.

## **5.2 Future Work**

The system only implemented a skeleton of the possible functionality for this application. In order to be used commercially, several additions can be made to this proof of concept project. Some of the additions are outlined below:

- The timing of the filtering process and web transaction processing provided some interesting results. The order of filtering may have a performance impact too. This can be investigated by changing the order that filter uses the signatures.

- The project could be extended to handle other databases such as MySQL, Oracle and Postgres as well as other operating systems.
- A further extension of the project could involve an investigation into the performance impact of the proxy server on data transfer. One question worth asking is: “What is the maximum number of connections or number of queries possible?” [Beynon, Sussman and Saltz, 1999]. This can be investigated by allowing multiple client to connect to TDSProxy by using threads
- The project could also be extended to work for SSL and allow for secure connection to the proxy server.
- Another useful implementation could be to filter data coming back from the database, checking for column names and data types that the user should not be allowed to see.
- To be a useful intrusion detection system, the filter should be able to find the attackers. Finding the user or attacker means logging login information for inspection. The filter would need a timestamp as well as the source and destination IP address [Finnigan, 2003].

---

**References**

- [Acunetix Ltd. 2005] *Acunetix Web Vulnerability Scanner – Features* [Online].  
<http://www.acunetix.com/wvs/wvs2manual.pdf>  
[Last accessed: 07/11/05]
- [Anley, C 2002a] *Advanced SQL injection* [Online]. Available:  
[http://www.nextgenss.com/papers/advanced\\_sql\\_injection.pdf](http://www.nextgenss.com/papers/advanced_sql_injection.pdf)  
[Last accessed: 07/11/05]
- [Anley, C 2002b] *(more) Advanced SQL Injection* [Online]. Available:  
[http://www.nextgenss.com/papers/more\\_advanced\\_sql\\_injection.pdf](http://www.nextgenss.com/papers/more_advanced_sql_injection.pdf)  
[Last accessed: 07/11/05]
- [Beynon, M D,  
Sussman, A,  
and Saltz, J  
1999] *Performance impact of proxies in data intensive client-server applications*. ACM Journal: Proceedings of the 13th international conference on Supercomputing, Rhodes, Greece. ACM Press, New York. pp: 383 – 390. ISBN:1-58113-164-X.
- [Beyond Security Ltd.  
2002] *SQL Injection Walkthrough* [Online]. Available:  
<http://www.securiteam.com/securityreviews/5DP0N1P76E.html> [Last accessed: 07/11/05]
- [Bruns B, Wheeler B,  
Schaal M, Ziglio F et  
al. 2005] *TDS Protocol Documentation*  
[Online]. Available: <http://www.freetds.org/tds.html>  
[Last accessed: 07/11/05]
- [Cerrudo, C 2004] *Manipulating Microsoft SQL Server Using SQL Injection*  
[Online]. Available:  
[http://www.appsecinc.com/presentations/Manipulating\\_SQL\\_Server\\_Using\\_SQL\\_Injection.pdf](http://www.appsecinc.com/presentations/Manipulating_SQL_Server_Using_SQL_Injection.pdf) [Last accessed: 07/11/05]
- [Check Point Software  
Technologies Ltd.  
2004] *Connectra Web Security Gateway* [Online]. Available:  
[http://www.securehq.com/images/checkpoint/connectra\\_data\\_sheet.pdf](http://www.securehq.com/images/checkpoint/connectra_data_sheet.pdf) [Last accessed: 02/10/05]
- [Chuvakin, A and  
Peikari, C 2004] *Security Warrior*, O'Reilly Media Inc., Sebastopol. pp 374-390
- [Davis, L 2005] *OSI Stack: OSI Protocol Description* [Online]. Available:  
[http://www.interfacebus.com/Design\\_OSI\\_Stack.html](http://www.interfacebus.com/Design_OSI_Stack.html)

- [Last accessed: 07/11/05]
- [Ethereal 2005a] *Ethereal* [Online]. Available:  
<http://www.ethereal.com/download.html>  
[Last accessed: 07/11/05]
- [Ethereal 2005b] *Supported Capture Media* [Online]. Available:  
<http://www.ethereal.com/media.html>  
[Last accessed: 07/11/05]
- [Finnigan, P 2002] *SQL Injection and Oracle, Part One* [Online]. Available:  
<http://www.securityfocus.com/infocus/1644>  
[Last accessed: 07/11/05]
- [Finnigan, P 2003] *Detecting SQL Injection in Oracle* [Online]. Available:  
<http://securityfocus.com/infocus/1714>  
[Last accessed: 07/11/05]
- [Fortify Software Inc. 2004] *Fortify product overview* [Online]. Available:  
<http://www.fortifysoftware.com/products/overview.jsp>  
[Last accessed: 07/11/05]
- [FreeTDS 2005] *TDS Protocol Documentation* [Online]. Available:  
<http://www.freetds.org/tds.html> [Last accessed: 07/11/05]
- [Giacobbi, G 2004] *The GNU Netcat project* [Online]. Available:  
<http://netcat.sourceforge.net/> [Last accessed: 07/11/05]
- [Grossman, J 2004] *The Challenges of Automated Web Application Scanning* [Online]. Available:  
<http://www.blackhat.com/presentations/win-usa-04/bh-win-04-grossman/bh-win-04-grossman-up.pdf>  
[Last accessed: 07/11/05]
- [Hoglund G and McGraw G, 2004] *Exploiting software: how to break code.* Addison –Wesley, pp 24, 41, 49, 56, 78 and 149
- [Hotchkies, C 2004] *Blind SQL Injection Automation Techniques* [Online]. Available: <http://www.blackhat.com/html/bh-media-archives/bh-archives-2004.html#USA-2004>  
[Last accessed: 07/11/05]
- [Howard, M and

- LeBlanc, D 2003] *secure application coding in a networking world*. 2<sup>nd</sup> Edition, Microsoft Press, Redmond, Washington, pp 400-411
- [Huang, Y, Huang, S, Lin, T, and Tsai, C 2003] *Web application security assessment by fault injection and behavior monitoring* in Proceedings of the 12th international conference on World Wide Web, Budapest, Hungary. SESSION: Data integrity. ACM Press, New York. pp: 148 – 159. ISBN:1-58113-680-3
- [Imperva Inc. 2004] *SQL injection - glossary* [Online]. Available: [http://www.imperva.com/application\\_defense\\_center/glossary/sql\\_injection.html](http://www.imperva.com/application_defense_center/glossary/sql_injection.html) [Last accessed: 07/11/05]
- [Imperva Inc. 2005] *SecureSphere™: Dynamic Profiling Firewall™* [Online]. Available: <http://www.imperva.com/products/securesphere/resources.asp?show=datasheet> [Last accessed: 07/11/05]
- [Kavado Inc. 2005] *Kavado* [Online]. Available: [http://www.kavado.com/pdf/ScanDo\\_Datasheet.pdf](http://www.kavado.com/pdf/ScanDo_Datasheet.pdf) [Last accessed: 07/11/05]
- [Kc, G S, Keromytis, A D, and Prevelakis V 2003] *Countering code-injection attacks with instruction-set randomization* in Proceedings of the 10th ACM conference on Computer and communication security. Washington D.C., USA. ACM Press, New York. pp. 272 - 280
- [Kline, K E 2004] *SQL in a Nutshell*, 2nd Edition, O'Reilly Media Inc., Sebastopol
- [Kocak, F 2004] *Packet Capture and Analyzer* [Online]. Available: <http://www.codeproject.com/csharp/pacanal.asp> [Last accessed: 07/11/05]
- [Lawson, L 2005] *Introduction to SQL Injection* Available: <http://www.securitydocs.com/pdf/3348.PDF> [Last accessed: 07/11/05]
- [Linux Journal 2004] *Real-world PHP security*, vol. 2004, Issue 120 (April 2004) pp. 1
- [Litchfield, D 2001] *Web Application Disassembly with ODBC Error Messages* [Online]. Available:

- <http://www.blackhat.com/presentations/win-usa-01/Litchfield/BHWin01Litchfield.doc>  
[Last accessed: 07/11/05]
- [Litchfield, D 2005] *Data-mining with SQL Injection and Inference* [Online]. Available:  
<http://www.ngssoftware.com/papers/sqlinference.pdf>  
[Last accessed: 07/11/05]
- [Litwin, P 2005] *Stop SQL Injection Attacks Before They Stop You* [Online]. Available:  
<http://msdn.microsoft.com/msdnmag/issues/04/09/SQLInjection/default.aspx> [Last accessed: 07/11/05]
- [Maor, O and Shulman, A 2003] *Blind SQL Injection* [Online]. Available:  
[http://www.imperva.com/application\\_defense\\_center/white\\_papers/blind\\_SQL\\_server\\_injection.html](http://www.imperva.com/application_defense_center/white_papers/blind_SQL_server_injection.html)  
[Last accessed: 07/11/05]
- [Maor, O and Shulman, A 2004] *SQL Injection Signatures Evasion* [Online]. Available:  
[http://www.imperva.com/application\\_defense\\_center/white\\_papers/sql\\_injection\\_signatures\\_evasion.html](http://www.imperva.com/application_defense_center/white_papers/sql_injection_signatures_evasion.html)  
[Last accessed: 07/11/05]
- [Microsoft 2003a] *Secure Multi-tier Deployment* [Online]. Available:  
<http://www.microsoft.com/technet/prodtechnol/SQL/2000/maintain/sp3sec03.msp> [Last accessed: 07/11/05]
- [Microsoft 2003b] *Checklist: Security best practices* [Online]. Available:  
<http://www.microsoft.com/technet/prodtechnol/SQL/2000/maintain/sp3sec04.msp> [Last accessed: 07/11/05]
- [Microsoft 2005] *Reserved Keywords* [Online]. Available:  
[http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tsqlref/ts\\_ra-rz\\_9oj7.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tsqlref/ts_ra-rz_9oj7.asp) [Last accessed: 07/11/05]
- [Network Chemistry 2005] *Packetyzer - Packet Analyzer for Windows* [Online]. Available:  
<http://www.networkchemistry.com/products/packetyzer/>  
[Last accessed: 07/11/05]

- [Networks Associates Technology, Inc. 2005] *McAfee System Protection: McAfee® Entercpt® Database Edition* [Online]. Available: [http://www.mcafee.com/us/local\\_content/datasheets/ds\\_entercpt\\_dt\\_edition.pdf](http://www.mcafee.com/us/local_content/datasheets/ds_entercpt_dt_edition.pdf) [Last accessed: 07/11/05]
- [Nummish and Xeron, 2005] *Absinthe* [Online]. Available <http://www.0x90.org/releases/absinthe/> [Last accessed: 07/11/05]
- [Overstreet, R 2004] *Protecting Yourself from SQL Injection Attacks* [Online]. Available: <http://www.4guysfromrolla.com/webtech/061902-1.shtml> [Last accessed: 07/11/05]
- [OWASP (The Open Web Application Security Project) 2004] *Top Vulnerabilities in Web Applications* [Online]. Available: <http://umn.dl.sourceforge.net/sourceforge/owasp/OWASPTopTen2004.pdf> [Last accessed: 07/11/05]
- [Phrack 2005] *Phrack* [Online]. Available: <http://www.phrack.org/show.php?p=54> [Last accessed: 07/11/05]
- [Postel, J 2004] *RFC 1700 - Assigned Numbers* [Online]. Available <http://www.faqs.org/ftp/rfc/pdf/rfc1700.txt.pdf> [Last accessed: 07/11/05]
- [Ristic, I 2005] *ModSecurity for Java* [Online]. Available: <http://www.modsecurity.org/projects/modsecurity/java/> [Last accessed: 07/11/05]
- [Rob, P and Coronel, C 2002] *Database Systems: Design, Implementation, & Management*. Fifth Edition. Course Technology. Boston Massachusetts, 02210
- [Seclutions, A G 2003] *AirLock - application security gateway* [Online]. Available: [http://www.seclutions.com/en/downloads/AirLock\\_Overview\\_Nov\\_2003.pdf](http://www.seclutions.com/en/downloads/AirLock_Overview_Nov_2003.pdf) [Last accessed: 07/11/05]
- [SoftLogica LLC. 2005] *Web Application testing (WAPT) Version 3* [Online]. Available: <http://www.loadtestingtool.com/> [Last accessed: 07/11/05]
- [Sommariskog, E 2005] *The Curse and Blessings of Dynamic SQL* [Online]. Available: [http://www.sommariskog.se/dynamic\\_sql.html](http://www.sommariskog.se/dynamic_sql.html)

[Last accessed: 07/11/05]

- [Spett, K 2002] *SQL Injection Are Your Web Applications Vulnerable?*  
[Online]. Available:  
<http://www.spidynamics.com/whitepapers/WhitepaperSQLInjection.pdf> [Last accessed: 07/11/05]
- [Spolsky, J 2000] *The Joel Test: 12 Steps to Better Code* [Online]. Available:  
<http://www.joelonsoftware.com/articles/fog0000000043.html>  
[Last accessed: 07/11/05]
- [Vicomsoft Ltd. 2003] *Firewall White Paper: What is the best firewall for me, and how can it improve Internet security?* [Online]. Available:  
[http://www.firewall-software.com/firewall\\_faqs/firewall\\_network\\_models.html](http://www.firewall-software.com/firewall_faqs/firewall_network_models.html)  
[Last accessed: 07/11/05]
- [WebCohort, Inc. 2004] *Only 10% of Web Applications are Secured Against Common Hacking Techniques* [Online]. Available:  
<http://www.imperva.com/company/news/2004-feb-02.html>  
[Last accessed: 07/11/05]
- [Weitz, E 2005] *The Regex Coach - interactive regular expressions* [Online]. Available: <http://www.weitz.de/regex-coach/>  
[Last accessed: 07/11/05]
- [WhiteHat Security, Inc. 2005] *WhiteHat Sentinel* [Online]. Available:  
<http://www.whitehatsec.com/services.shtml>  
[Last accessed: 07/11/05]

---

# **Appendix A – Project Poster**

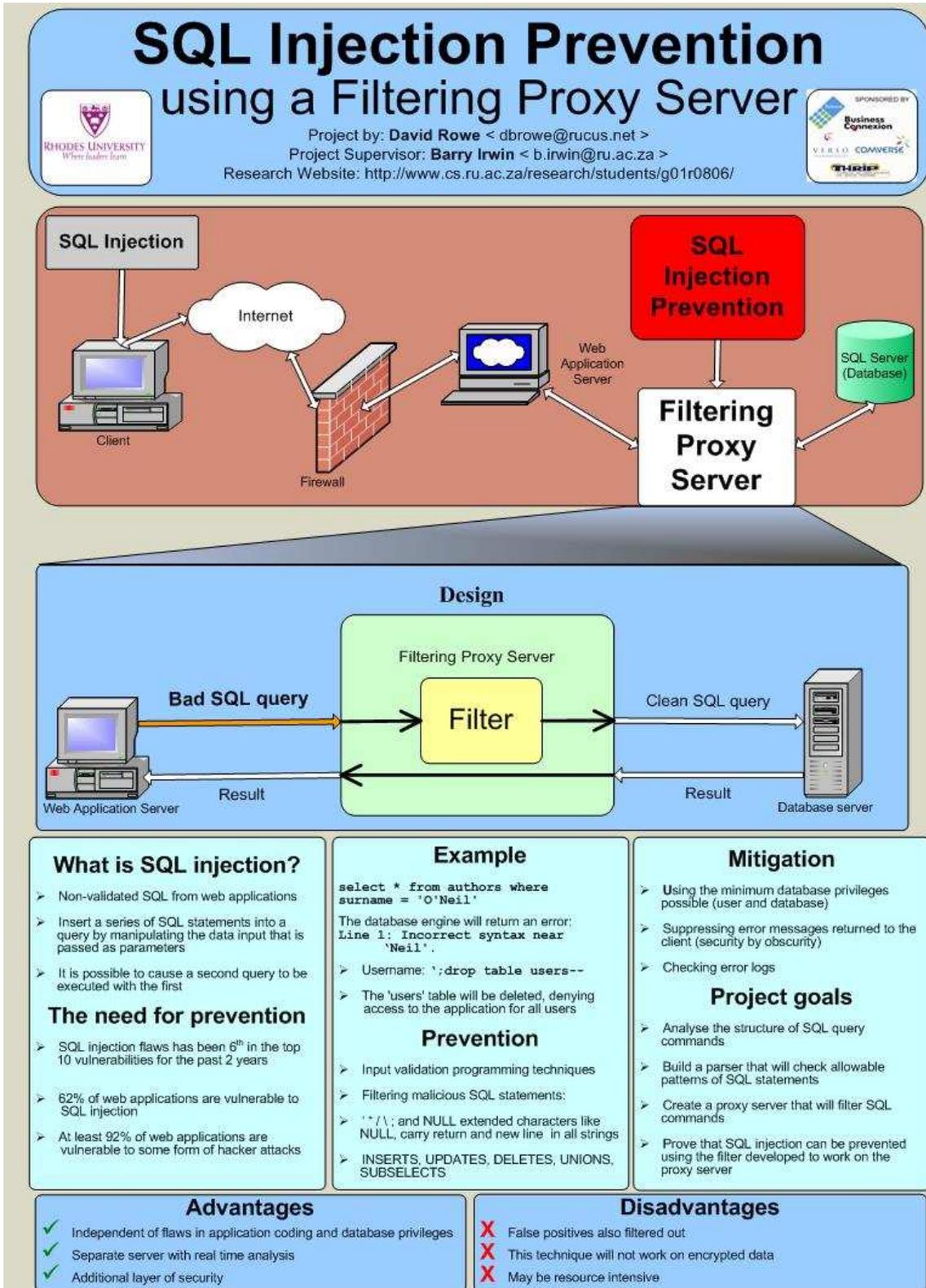
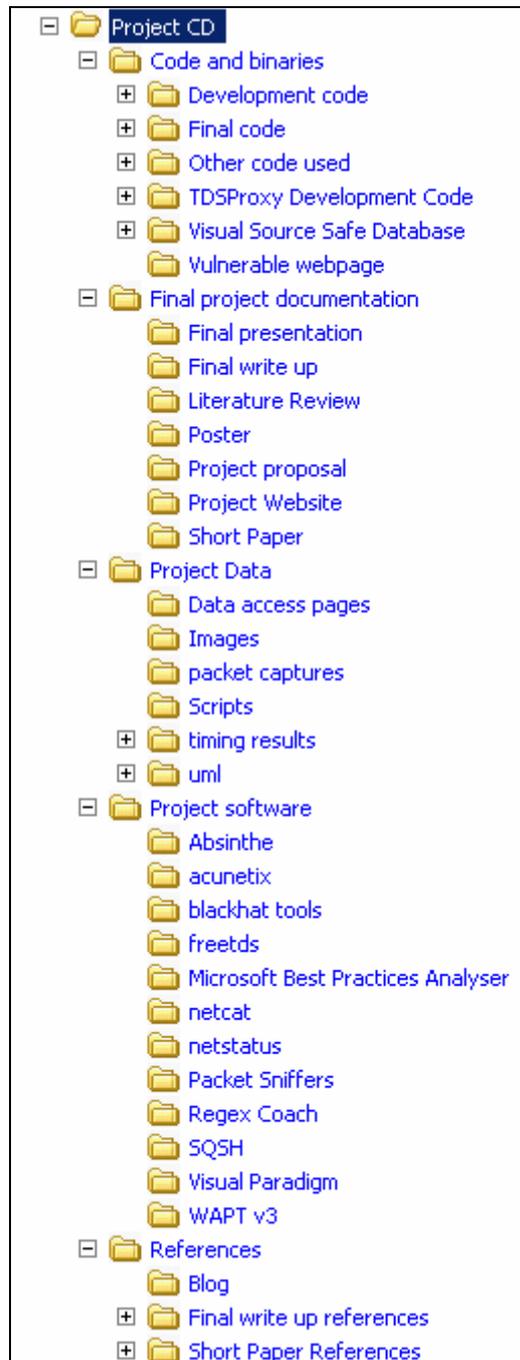


Figure A.1: Project Poster

---

# **Appendix B – CD Contents**



**Figure B.1: Contents of project CD**

---

# Appendix C – Code overview

```

1 using System;
2 using System.Net;
3 using System.Net.Sockets;
4 using System.Text;
5 using System.Configuration;
6 using System.Windows.Forms;
7 using System.Threading;
8
9 namespace TDS
10 {
11     /// <summary>
12     /// this program allows access to connect to my machine and then forward the packet to
13     /// the port that the database on my machine is listening on.
14     ///
15     /// Version 6 will aim at extending the code to be able to:
16     /// a)extract the sql from the packets
17     /// b)config file for port settings
18     /// c)filter the sql extracted from the packets
19     ///
20     /// This class handles the declarations
21     /// reads the config file
22     /// creates the logger
23     /// initialises the front end sockets
24     /// initialises the proxy backend
25     /// receives connection from the client
26     /// send the requests to the backend
27     /// forwards responses from the backend
28     /// </summary>
29     public class TcpSock
30     {
31         Timing for statistical purposes.
32
33         private static void OpenSocket(ref Socket serverSocket, IPEndPoint ipEndPointHost)...
34
35         public static void Main()
36         {
37             debug options for log file and console printouts
38
39             software title
40
41             declarations and reading config file
42
43             creating the logger, filter and proxy back end
44
45             setting the IP end points of the host and database machine.
46
47             initialising the front end serverSocket and accepting a connection
48
49             sending and receiving data in a while loop
50
51         } //void main
52     } //class
53 } //namespace
54

```

Figure C.1: TCPSock Class

```

1 using System;
2 using System.Net;
3 using System.Net.Sockets;
4 using System.Text;
5 using System.IO;
6 using System.Threading;
7
8 namespace TDS
9 {
10     /// <summary>
11     /// This class is used to:
12     /// interface with the front end socket,
13     /// receive the client query and filter it,
14     /// take appropriate action should an attack be noted and the database database
15     /// send the clean query to the database
16     /// handle the database response
17     /// return this response to the proxy front end
18     /// </summary>
19
20     public class ProxyBackend
21     {
22         debug options for log file and console printouts
27
28         declarations
56
57         constructor
75
76         connect to database given ipcp
104
105         send the message to the database after filtering it
288
289         receive from the database
346
347         release
361
362         timing methods
383
384     }
385 }

```

Figure C.2: ProxyBackend Class

```

1 using System;
2 using System.IO;
3
4 namespace TDS
5 {
6     /// <summary>
7     /// This class is used to log things.
8     /// It appends the data to the end of the file specified
9     /// This approach was taken to prevent many files with date stamps and the like being produced.
10    /// So, there is one huge file with all the logs in it
11    ///
12    /// ***This can later be extended to log things directly into the database***
13    ///
14    /// </summary>
15    public class Logger
16    {
17        private string fileName = "SqlInjectionLog.txt";
18
19        constructors
33
34        log and dumplog methods
65    } //class Logger
66 } //namespace
67

```

Figure C.3: Logger Class

```

1 using System;
2 using System.Collections;
3 using System.Configuration;
4 using System.IO;
5 using System.Text.RegularExpressions;
6
7 namespace TDS
8 {
9     /// <summary>
10    /// this class will allow us to interpret what is in the tds packet.
11    /// this should be loaded at the beginning so that the data is read
12    /// into the array before any queries are made to the database
13    /// logging - [query typeOf(black,white,grey) timestamp ruleThatCaughtIt]
14    /// </summary>
15    public class Filter
16    {
17        declarations
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51    public Filter()...
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94    getting and setting a query flag - used for timing
95
96
97
98
99
100
101
102
103
104
105
106    /// <summary>
107    /// Search: Searches theQuery for matches in the order given in the configuration file.
108    /// </summary>
109    /// <param name="theQuery"></param>
110    /// <returns></returns>
111    public bool Search(string theQuery)...
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173    /// <summary>
174    /// SearchQuery: search for matches using regex
175    /// </summary>
176    /// <param name="query"></param>
177    /// <param name="myAL"></param>
178    /// <returns></returns>
179    private bool SearchQuery(string query, ArrayList myAL)...
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234    /// <summary>
235    /// ReadFileInfo: Read the text file and put the values into an ArrayList
236    /// </summary>
237    /// <param name="filename"></param>
238    /// <param name="myAL"></param>
239    private void ReadFileInfo(String filename, ArrayList myAL)...
240
241
242
243
244
245
246
247
248
249
250
251
252
253    /// <summary>
254    /// print method used to print the values in the ArrayList
255    /// </summary>
256    /// <param name="myList"></param>
257    private void PrintValues( IEnumerable myList )...
258
259
260
261
262    } //class Filter
263 } //namespace
264 } //namespace
265

```

Figure C.4: Filter Class

```
1 using System;
2 using System.Net;
3 using System.Net.Sockets;
4 using System.Threading;
5 using System.Text;
6
7 namespace TDS
8 {
9     /// <summary>
10    /// This class is a UDP server which broadcasts the SQL injection attack query that has been filtered
11    /// </summary>
12    public class UdpServer
13    {
14        variable declarations
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31    constructor which sets the ports and calls Initialise() which starts the udpserver
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46    initialise the UDP server
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87    send the message
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108    terminate
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126    } //class
127
128 } //namespace
129
```

Figure C. 5: UdpServer Class

---

# Appendix D – Timing tests

## D.1 Local host tests

The following tests were done using a tool called web application testing (WAPT) version 3 [SoftLogica LLC., 2005]. The local host tests were done on hons08 where TDSProxy and the database were being hosted.

### D.1.1 Select statement, direct to the database

--- Basic statistics ---

Page name: hons08 select page – direct to database

Min web transaction (without images): 3.56

Avg web transaction (without images): 4.05

Max web transaction (without images): 10.41

--- Network traffic details ---

Total bytes sent: 384891

Total bytes received: 601067

Average server bandwidth (Kbits/sec): 1076.37

Average user bandwidth (Kbits/sec): 1076.37

--- Summary times ---

Virtual Users statistics:

User	Pages	Time	AP/S	AT/P
1	1000	7328	136.46	7.33

Total work time: 7328

Total pages made: 1000

Total average pages per second: 136.46

--- HTTP response codes details ---

Code	Count
200	1000

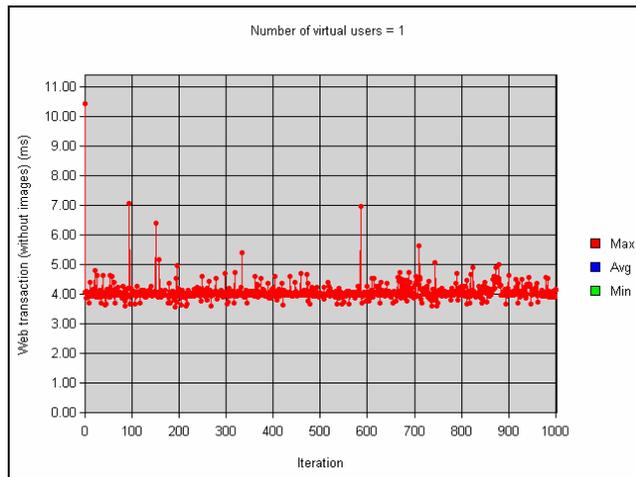


Figure D.1: Graph showing hons08 select page – direct to database

### D.1.2 Insert statement, direct to the database

--- Basic statistics ---

Page name: hons08 insert page – direct to database

Min web transaction (without images): 4.03

Avg web transaction (without images): 4.85

Max web transaction (without images): 10.33

--- Network traffic details ---

Total bytes sent: 429891

Total bytes received: 628067

Average server bandwidth (Kbits/sec): 1014.46

Average user bandwidth (Kbits/sec): 1014.46

--- Summary times ---

Virtual Users statistics:

User	Pages	Time	AP/S	AT/P
1	1000	8343	119.86	8.34

Total work time: 8343

Total pages made: 1000

Total average pages per second: 119.86

--- HTTP response codes details ---

Code	Count
200	1000

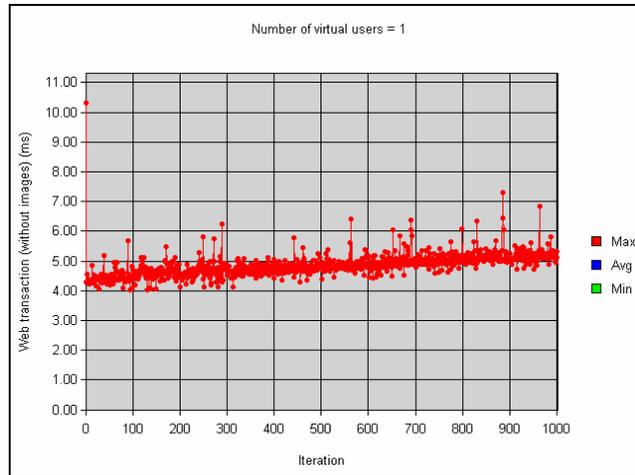


Figure D.2: Graph showing hons08 insert page – direct to database

### D.1.3 Select statement, through TDSProxy without filtering

--- Basic statistics ---

Page name: hons08 select page, TDSProxy, no filter

Min web transaction (without images):	4.05
Avg web transaction (without images):	4.42
Max web transaction (without images):	42.74

--- Network traffic details ---

Total bytes sent:	384891
Total bytes received:	601067
Average server bandwidth (Kbits/sec):	963.32
Average user bandwidth (Kbits/sec):	963.32

--- Summary times ---

Virtual Users statistics:

User	Pages	Time	AP/S	AT/P
1	1000	8188	122.13	8.19

Total work time: 8188  
 Total pages made: 1000  
 Total average pages per second: 122.13

--- HTTP response codes details ---

Code	Count
200	1000

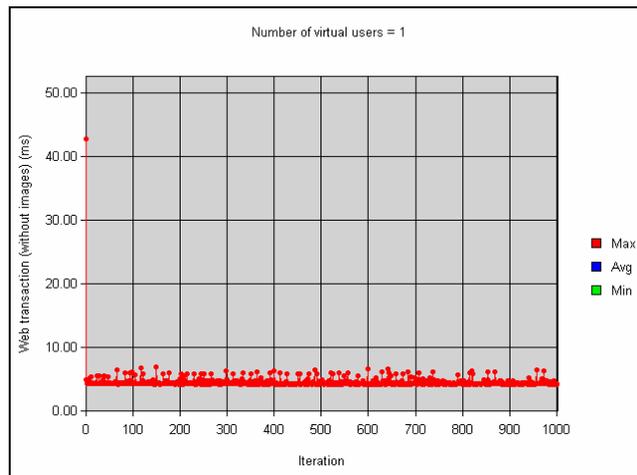


Figure D.3: Graph showing hons08 select page, TDSProxy, no filter

#### D.1.4 Select statement, through TDSProxy with filtering

--- Basic statistics ---

Page name: hons08 select page, TDSProxy, filter  
 Min web transaction (without images): 24.33  
 Avg web transaction (without images): 25.33  
 Max web transaction (without images): 88.51

--- Network traffic details ---

Total bytes sent: 384891  
 Total bytes received: 601067  
 Average server bandwidth (Kbits/sec): 271.41  
 Average user bandwidth (Kbits/sec): 271.41

--- Summary times ---

Virtual Users statistics:

User	Pages	Time	AP/S	AT/P
1	1000	29062	34.41	29.06

Total work time: 29062  
 Total pages made: 1000  
 Total average pages per second: 34.41

--- HTTP response codes details ---

Code	Count
200	1000

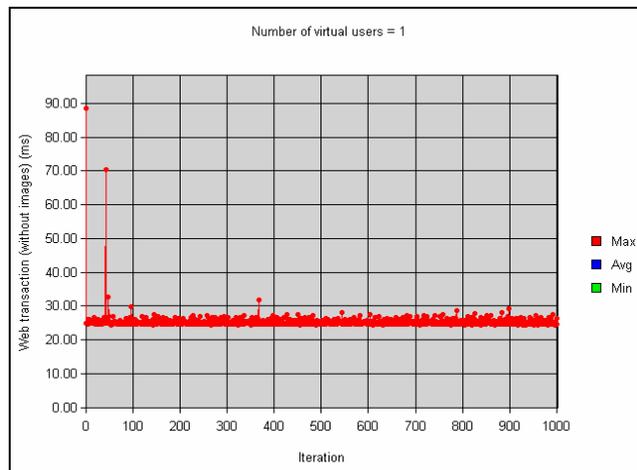


Figure D.4: Graph showing hons08 select page, TDSProxy, filter

### D.1.5 Insert statement, through TDSProxy without filtering

--- Basic statistics ---

Page name: hons08 insert page - TDSProxy, no filter

Min web transaction (without images): 4.57  
 Avg web transaction (without images): 5.29  
 Max web transaction (without images): 41.86

--- Network traffic details ---

Total bytes sent: 429891  
 Total bytes received: 628067

Average server bandwidth (Kbits/sec): 908.90

Average user bandwidth (Kbits/sec): 908.90

--- Summary times ---

Virtual Users statistics:

User	Pages	Time	AP/S	AT/P
1	1000	9312	107.39	9.31

Total work time: 9312

Total pages made: 1000

Total average pages per second: 107.39

--- HTTP response codes details ---

Code	Count
200	1000

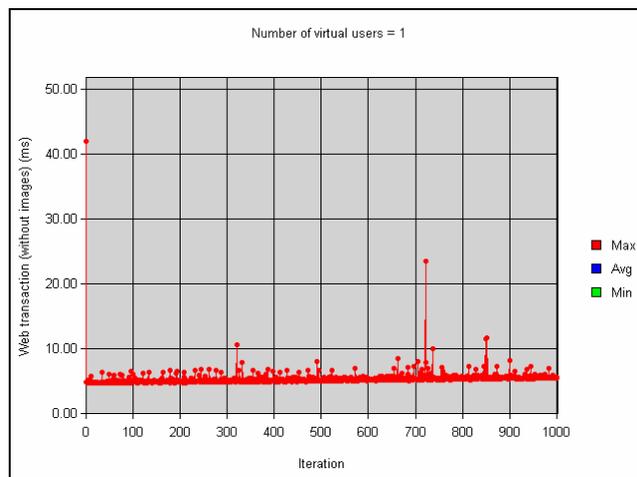


Figure D.5: Graph showing hons08 insert page - TDSProxy, no filter

### D.1.6 Insert statement, through TDSProxy with filtering

--- Basic statistics ---

Page name: hons08 insert page – TDSProxy, filter

Min web transaction (without images): 24.84

Avg web transaction (without images): 26.30

Max web transaction (without images): 89.76

--- Network traffic details ---

Total bytes sent: 429891  
 Total bytes received: 628067  
 Average server bandwidth (Kbits/sec): 281.53  
 Average user bandwidth (Kbits/sec): 281.53

--- Summary times ---

Virtual Users statistics:

User	Pages	Time	AP/S	AT/P
1	1000	30063	33.26	30.06

Total work time: 30063  
 Total pages made: 1000  
 Total average pages per second: 33.26

--- HTTP response codes details ---

Code	Count
200	1000

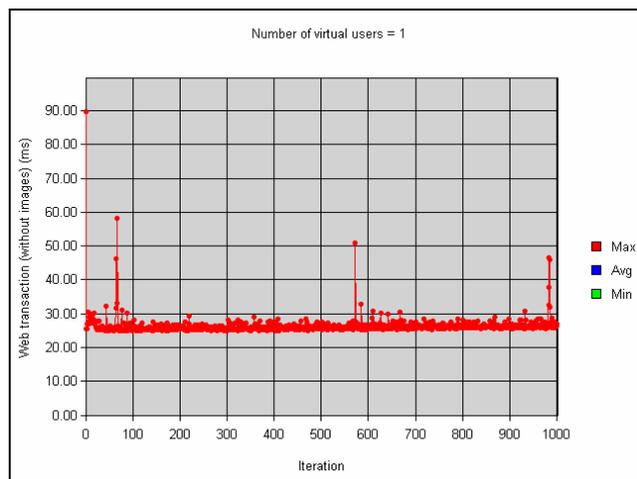


Figure D.6: Graph showing hons08 insert page – TDSProxy, filter

## D.2 Netserv tests

Netserv is the computer on which the following tests were run. TDSProxy and the database were on hons08, a remote machine.

### **D.2.1 Select statement, direct to the database**

--- Basic statistics ---

Page name: netserv select page – direct to database

Min web transaction (without images): 4.20  
Avg web transaction (without images): 6.42  
Max web transaction (without images): 17.57

--- Network traffic details ---

Total bytes sent: 402881  
Total bytes received: 601067  
Average server bandwidth (Kbits/sec): 553.90  
Average user bandwidth (Kbits/sec): 553.90

--- Summary times ---

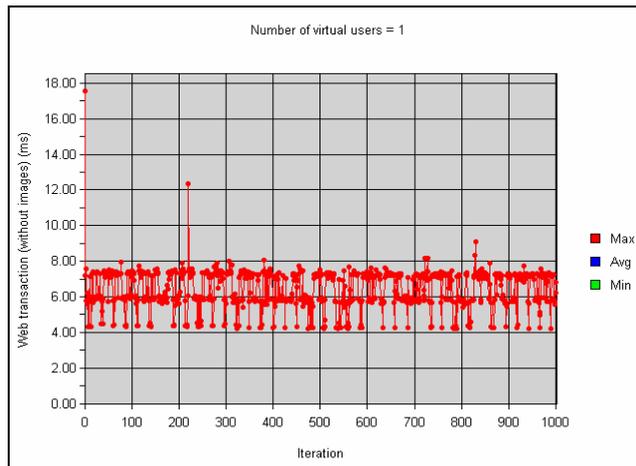
Virtual Users statistics:

User	Pages	Time	AP/S	AT/P
1	1000	14500	68.97	14.50

Total work time: 14500  
Total pages made: 1000  
Total average pages per second: 68.97

--- HTTP response codes details ---

Code	Count
200	1000



**Figure D.7: Graph showing netserv select page – direct to database**

### D.2.2 Insert statement, direct to the database

--- Basic statistics ---

Page name: netserv insert page – direct to database

Min web transaction (without images): 4.61  
 Avg web transaction (without images): 7.24  
 Max web transaction (without images): 11.36

--- Network traffic details ---

Total bytes sent: 459881  
 Total bytes received: 636067  
 Average server bandwidth (Kbits/sec): 572.56  
 Average user bandwidth (Kbits/sec): 572.56

--- Summary times ---

Virtual Users statistics:

User	Pages	Time	AP/S	AT/P
1	1000	15313	65.30	15.31

Total work time: 15313  
 Total pages made: 1000  
 Total average pages per second: 65.30

--- HTTP response codes details ---

Code	Count
200	1000

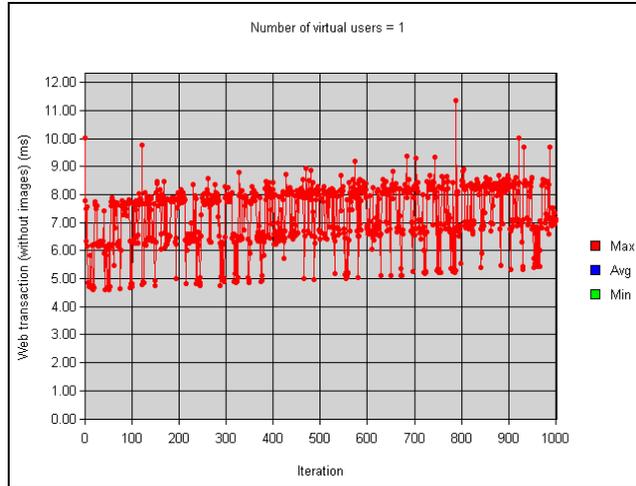


Figure D.8: Graph showing netserv insert page – direct to database

### D.2.3 Select statement, through TDSProxy without filtering

--- Basic statistics ---

Page name: netserv select page, TDSProxy, no filter

Min web transaction (without images):	4.79
Avg web transaction (without images):	6.70
Max web transaction (without images):	37.60

--- Network traffic details ---

Total bytes sent:	402881
Total bytes received:	601067
Average server bandwidth (Kbits/sec):	553.33
Average user bandwidth (Kbits/sec):	553.33

--- Summary times ---

Virtual Users statistics:

User	Pages	Time	AP/S	AT/P
1	1000	14515	68.89	14.52

Total work time: 14515  
 Total pages made: 1000  
 Total average pages per second: 68.89

--- HTTP response codes details ---

Code Count  
 200 1000

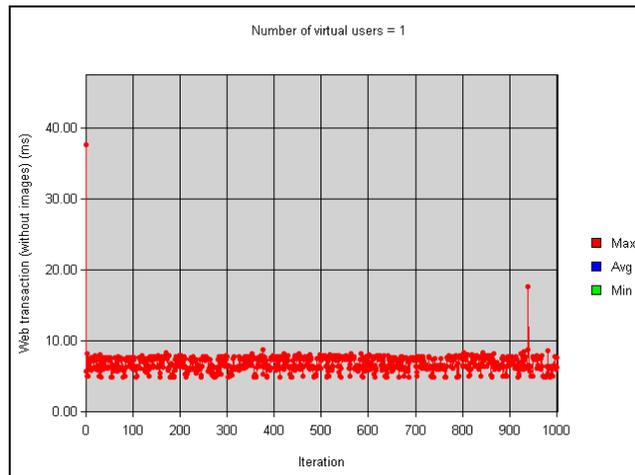


Figure D.9: Graph showing netserv select page, TDSProxy, no filter

#### D.2.4 Select statement, through TDSProxy with filtering

--- Basic statistics ---

Page name: netserv select page, TDSProxy, filter

Min web transaction (without images): 24.75  
 Avg web transaction (without images): 27.46  
 Max web transaction (without images): 100.62

--- Network traffic details ---

Total bytes sent: 402881  
 Total bytes received: 601067  
 Average server bandwidth (Kbits/sec): 227.14  
 Average user bandwidth (Kbits/sec): 227.14

--- Summary times ---

Virtual Users statistics:

User	Pages	Time	AP/S	AT/P
1	1000	35360	28.28	35.36

Total work time: 35360  
 Total pages made: 1000  
 Total average pages per second: 28.28

--- HTTP response codes details ---

Code	Count
200	1000

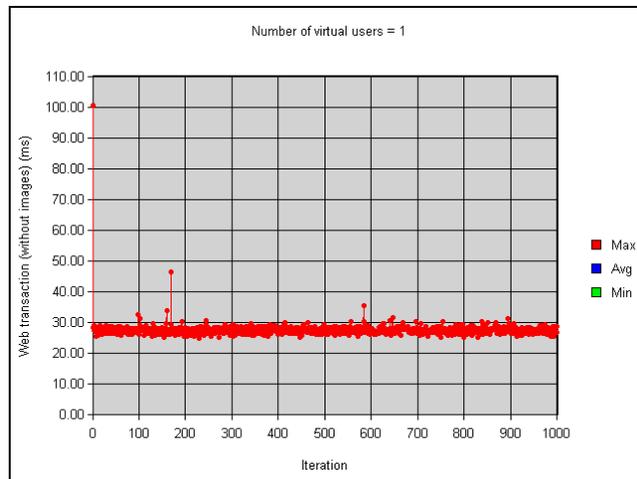


Figure D.10: Graph showing netserv select page, TDSProxy, filter

### D.2.5 Insert statement, through TDSProxy without filtering

--- Basic statistics ---

Page name: netserv insert page, TDSProxy, no filter

Min web transaction (without images): 5.96  
 Avg web transaction (without images): 8.02  
 Max web transaction (without images): 39.14

--- Network traffic details ---

Total bytes sent: 459881  
 Total bytes received: 636067  
 Average server bandwidth (Kbits/sec): 547.43

Average user bandwidth (Kbits/sec): 547.43

--- Summary times ---

Virtual Users statistics:

User	Pages	Time	AP/S	AT/P
1	1000	16016	62.44	16.02

Total work time: 16016

Total pages made: 1000

Total average pages per second: 62.44

--- HTTP response codes details ---

Code	Count
200	1000

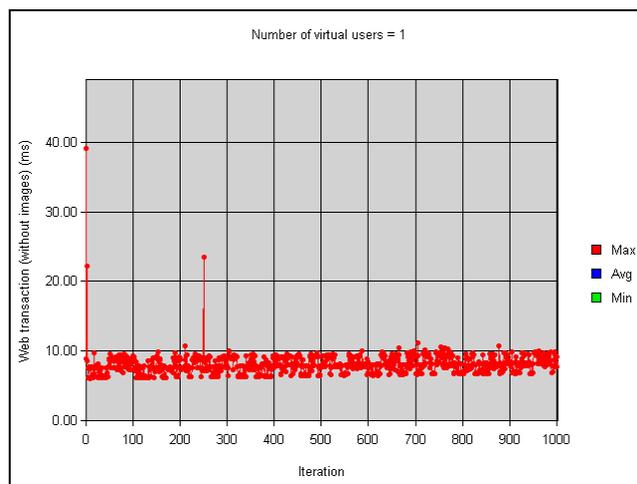


Figure D.11: Graph showing netserv insert page, TDSProxy, no filter

### D.2.6 Insert statement, through TDSProxy with filtering

--- Basic statistics ---

Page name: netserv insert page TDSProxy, filter

Min web transaction (without images): 25.52

Avg web transaction (without images): 28.43

Max web transaction (without images): 88.65

--- Network traffic details ---

Total bytes sent: 459881  
Total bytes received: 636067  
Average server bandwidth (Kbits/sec): 241.35  
Average user bandwidth (Kbits/sec): 241.35

--- Summary times ---

Virtual Users statistics:

User	Pages	Time	AP/S	AT/P
1	1000	36328	27.53	36.33

Total work time: 36328  
Total pages made: 1000  
Total average pages per second: 27.53

--- HTTP response codes details ---

Code	Count
200	1000

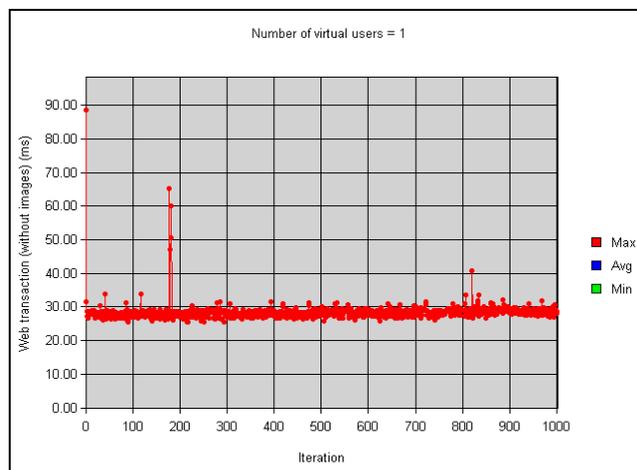


Figure D.12: Graph showing netserv insert page TDSProxy, filter