Preventing SQL Injections by Hashing the Query Parameter Data

Manfred Jönsson
Patrik Lokby

Dept. Computer Science & Engineering
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden
This thesis is submitted to the Department of Computer Science & Engineering at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Bachelor of Science in IT-Security. The thesis is equivalent to 10 weeks of full-time studies.

Contact Information:
Author(s):
Manfred Jönsson
E-mail: maft14@student.bth.se
Patrik Lokby
E-mail: palc14@student.bth.se

University advisor:
Hans Tap
Dept. Creative Technologies

Dept. Computer Science & Engineering  Internet : www.bth.se
Blekinge Institute of Technology      Phone   : +46 455 38 50 00
SE–371 79 Karlskrona, Sweden        Fax     : +46 455 38 50 57
Abstract

Context. Many applications today use databases to store user information or other data for their applications. This information can be accessed through various different languages depending on what type of database it is. Databases that use SQL can maliciously be exploited with SQL injection attacks. This type of attack involves inserting SQL code in the query parameter. The injected code sent from the client will then be executed on the database. This can lead to unauthorized access to data or other modifications within the database.

Objectives. In this study we investigate if a system can be built which prevents SQL injection attacks from succeeding on web applications that is connected with a MySQL database. In the intended model, a proxy is placed between the web server and the database. The purpose of the proxy is to hash the SQL query parameter data and remove any characters that the database will interpret as comment syntax. By processing each query before it reaches its destination we believe we can prevent vulnerable SQL injection points from being exploited.

Methods. A literary study is conducted the gain the knowledge needed to accomplish the objectives for this thesis. A proxy is developed and tested within a system containing a web server and database. The tests are analyzed to arrive at a conclusion that answers ours research questions.

Results. Six tests are conducted which includes detection of vulnerable SQL injection points and the delay difference on the system with and without the proxy. The result is presented and analyzed in the thesis.

Conclusions. We conclude that the proxy prevents SQL injection points to be vulnerable on the web application. Vulnerable SQL injection points is still reported even with the proxy deployed in the system. The web server is able to process more http requests that requires a database query when the proxy is not used within the system. More studies are required since there is still vulnerable SQL injections points.

Keywords: SQL injection, Proxy, MD5 hash, Regex.
## Contents

Abstract

1 Introduction
   1.1 Background .................................................. 1
   1.2 Purpose .................................................... 3
   1.3 Aims and Objectives ........................................ 4
   1.4 Research Questions ....................................... 4

2 Related Work

3 Method
   3.1 Literary Study .............................................. 6
   3.2 System Overview .......................................... 6
   3.3 Test Development ......................................... 7

4 Implementation
   4.1 Proxy Server ............................................... 9
      4.1.1 Architecture ........................................ 9
      4.1.2 Query Processing .................................. 10
   4.2 Test Execution ........................................... 11
      4.2.1 Test 1 with proxy connected ...................... 12
      4.2.2 Test 2 without proxy connected ................. 13
      4.2.3 Test 3 with proxy connected ..................... 13
      4.2.4 Test 4 without proxy connected ................. 13
      4.2.5 Test 5 with proxy connected ..................... 13
      4.2.6 Test 6 without proxy connected ................. 14

5 Results
   5.1 Test 1 with proxy connected ............................. 15
   5.2 Test 2 without proxy connected ......................... 16
   5.3 Test 3 with proxy connected ............................. 17
   5.4 Test 4 without proxy connected ......................... 18
   5.5 Test 5 with proxy connected ............................. 19
   5.6 Test 6 without proxy connected ......................... 19
6 Analysis
7 Conclusions and Future Work
Bibliography
**List of Figures**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>OWASP Top Ten 2004 - 2013</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>NIST SQLI Vulnerability Statistics 2008 - 2016</td>
<td>2</td>
</tr>
<tr>
<td>3.1</td>
<td>System Overview</td>
<td>7</td>
</tr>
<tr>
<td>4.1</td>
<td>Proxy Server Sequence Diagram</td>
<td>10</td>
</tr>
<tr>
<td>5.1</td>
<td>SQLMap results with proxy connected</td>
<td>17</td>
</tr>
<tr>
<td>5.2</td>
<td>SQLMap results with the web server directly connected to the database</td>
<td>18</td>
</tr>
<tr>
<td>6.1</td>
<td>Output from proxy when a stacked query is bypassing the proxy</td>
<td>20</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Test 1 Results .................................................. 16
5.2 Test 2 Results .................................................. 16
5.3 Test 3 Results .................................................. 17
5.4 Test 4 Results .................................................. 18
5.5 Test 5 & 6 results ............................................. 19
Chapter 1

Introduction

1.1 Background

Nowadays most organizations and companies are dependent and based on the Internet. Companies, private persons and even government are relying on digital services in order for their communication and business to work. The digital information that travels between the endpoints are often stored so that it can be retrieved on a later occasion. The piles of paper are gone and the information is stored in digital format. This causes web-based attacks to be a major threat to all parties involved using web services.

The collections of data that web applications store is commonly stored within databases. This data can be accessed through various different languages depending on what kind of database it is. In this project we will focus on SQL (‘Structured Query Language’) databases. SQL is used in these types of databases to retrieve and manipulate information [1]. SQL can maliciously be exploited with SQLI (‘Structured Query Language Injection’) [2]. This type of attack involves inserting SQL queries into the query parameter [3]. The injected code that is sent from the client will then be executed on the database and can lead to unauthorized access to data or modifications within the database. The code injected into the SQL query parameter will be interpreted as regular SQL code and executed on the database. In order to avoid this kind of attack then you would have to know as a precondition which part of the query that is to be interpreted as code and which part to be treated as literal characters. Without such preconditions an SQLI attack could succeed.

OWASP (‘Open Web Application Security Project’) is an organization working for security within software applications [4]. Every third year OWASP presents a top 10 document that contains the ten most critical security risks to web applications [5]. SQLI attacks has gone from place number six in 2004 to taking the number one spot since 2010 on the list. In 2013 the latest report from OWASP, SQLIA still was the rank one security risk for organizations [6].
The National Vulnerability Database (‘NVD’) is a database managed by the National Institute of Standards and Technology (‘NIST’). It is a database containing vulnerabilities that is based on the Common Vulnerabilities and Exposures (‘CVE’) dictionary [7][8]. In 2008 NIST reported 1092 SQLI vulnerabilities within applications [9]. That number has fortunately dropped over the last few years. However vulnerabilities continuously gets reported and the problem persists even with newly developed applications.
1.2 Purpose

The communication with databases is commonly integrated with web applications and the SQL queries can be run through a variety of scripting languages such as PHP and Python. Security around web applications often lies in the hands of the developer. Without the right training and knowledge, a developer can produce a web application vulnerable to SQLI. The programming languages used to build web applications often come with secure implementation techniques to avoid such vulnerabilities. However, these techniques will only be as good as the developer and will therefore not be a guaranteed protection to SQLIA. This has made secure code practice a critical and important aspect when developing web applications.

There are already several existing mitigation techniques that protect against SQLI. Prepared statements is a commonly used mitigation technique that makes it possible for the database to distinguish between the actual code and the SQL query parameters [10]. With prepared statements, the SQL code is first defined and then the query parameters is passed to the database. This allows the database to distinguish the code to be executed from the code that may be found in the query parameter. Another technique used to mitigate SQLI is character escaping [11]. This technique is based on escaping the query parameter data before the SQL query gets passed to the database [2]. The escaped characters will not be interpreted as code whenever it is escaped but instead treated as a literal character. However, this mitigation technique has shown to be less good but is still used today by developers.

In this thesis, we will investigate the possibility of using a proxy between the web server and database that prevents SQLI from succeeding. The purpose of the proxy is to prevent code within an SQL query parameter from being executed. We will investigate if hashed query parameter data can prevent any vulnerable injection points from being exploited. With a proxy, we can process SQL queries before they reach the database to make sure that no malicious code is executed.
1.3 Aims and Objectives

- Explore if hashing the parameter data within SQL queries can prevent SQLI from succeeding.

- Investigate how a system can be implemented and built to protect web applications from SQLI attacks.

- Perform penetration tests on a built-in web application with no defense mechanisms against SQLI. The tests should be performed when the proxy is in use and when it is not in use.

1.4 Research Questions

- Will the proxy prevent any vulnerable SQL injection points in the built web application from being exploited.

- What is the dilatory effect on our system when it’s connected with the proxy server versus without it.

**SQL injection points**, queries that the web application might be vulnerable to. Such as boolean, error, union, stacked, time and inline based queries.

**Dilatory effect** is based on the average time difference on 1000*10 http requests when the proxy is used versus without it.
Chapter 2

Related Work

Even if SQL injection attacks have been known for a long time, the risk for it happening is still high as OWASP plans to rank injections in general as number one security risk in 2017 [12]. There are ways to avoid being vulnerable to SQL injection attacks, and one is secure coding. With secure coding you make sure that no user input manipulates the SQL query and executes a query that was not intended [13]. The way a programmer could do this is by using prepared statements to execute a template query to the database where the user input is left out. When the result is returned, the user input can be added to the missing parameters in the SQL template executing the full query. This defense involves that the programmer makes no programming errors resulting in a vulnerable web application.

Another way to deal with attacks is to use a proxy that intercepts requests made to the database to make sure that no malicious query parameters is sent to the database [14][15][16]. These proxies are built to handle queries as the traffic is directed from the web application to the proxy. The proxy will try to validate the input data or hash it or even do both. Our proxy will not need to get redirected traffic from the web application as it works between the the web server and database making it unknown to the applications. This makes it effortless for the programmer to implement our proxy as he would not need to make any changes to his own code.
Chapter 3

Method

A strategic implementation of our methods has been done to get a strong and credible result. We believe that the methods we used during our thesis give us the ability to answer our research questions.

3.1 Literary Study

The literary study focused on gathering information and knowledge on how similar systems was built as well as mitigation techniques against SQLI. The purpose was also to get the theoretical knowledge needed to build a proxy in Java, capable of processing the communication between a web server and a MySQL database. The system architecture was an important aspect for us during the study. We wanted to build a system architecture that required minimal changes to an already deployed system.

3.2 System Overview

One of our main focus for this project was to create a system architecture that required minimal changed to an already deployed web application that was connected with a database. To simulate this environment we developed a vulnerable web application that was connected with a database. The system consists of a Apache web server, a web application that was built on PHP and a MySQL database. The web application has two different pages. A login page that requires username and password to login and a welcome screen that shows up after a successful login. The database has a user table which consists of username and password. With these components we simulated an already deployed system that was functioning for the purpose it was built. To minimize the changes that has to be done on the system we had to place the proxy in a spot that wouldn’t require much changes to the current code. Research on other similar system models suggests that the web server should forward the entire http request to the proxy [14]. The http request would then be sent back to the web server, which would then send the SQL query to the database. This would require the web server
to make an extra request before it can contact the database. Other examples on
similar models have placed the proxy before the web server [15]. The proxy server
would then have to be very advanced in order for it to work on protocols other
than http, for example https traffic would have to be decrypted before any of the
queries can get processed. We found our solution for the system model to be a
proxy placed between the web server and the database. The proxy will receive
each query from the web server and then process the query parameters. The
queries will then be forwarded to the database and the response gets forwarded
back to the web server in its pristine form.

![Diagram of System Overview]

Figure 3.1: System Overview

With the system model from the figure above we can minimize the code change
for a developer as well as keeping the desirable features on the proxy server.

3.3 Test Development

The tests in this thesis was produced to be both manually and automatically
performed. The manual part of the tests are done by injecting SQL code into the
login form on the website. The purpose of the injected SQL code is to manipulate
the query sent to the database in order to gain access to the welcome page. The
part of the automatic tests that performs SQLI are used with SQLMap [17].
Chapter 3. Method

SQLMap is an open source tool that can perform SQLI and produce logs of the results. SQLMap can be used with custom settings. The settings used with SQLMap are the same on all the performed tests whenever the tool is used. Settings that was used (see 4.2).

Test 1. The purpose of the test is to get redirected to the welcome page from the login page by performing SQLI. The SQL code used in this test should be the same as in test 2. The web server should be connected with the proxy during this test.

Test 2. The purpose of the test is to get redirected to the welcome page from the log in page by performing SQLI. The SQL code used in this test should be the same as in test 1. The web server should be directly connected with the database during this test.

Test 3. The purpose of this test is to use SQLMap as an automated audit tool to perform SQLI on the web server. The setting used with SQLMap on this test should be the same as in test 4. The web server should be connected with the proxy during this test.

Test 4. The purpose of this test is to use SQLMap as an automated audit tool to perform SQLI on the web server. The setting used with SQLMap on this test should be the same as in test 3. The web server should be directly connected with the database during this test.

Test 5. The purpose of this test is to measure the average time it takes to make 1000*10 requests to the web server. A bash script is used to make the requests while it measures the time it takes to complete all the requests. The request used in this test should be the same as in test 6. The web server should be connected with the proxy during this test.

Test 6. The purpose of this test is to measure the average time it takes to make 1000*10 requests to the web server. A bash script is used to make the requests while it measures the time it takes to complete all the requests. The request used in this test should be the same as in test 5. The web server should be directly connected with the database during this test.
This chapter describes how the proxy server is implemented and how the tests were executed.

4.1 Proxy Server

The proxy server is developed in Java 8 and uses sockets to transfer data over the network. Whenever the proxy server interprets any data as SQL parameter data it hashes it using MD5 and removes any characters that the database can interpret as comment syntax. MD5 was chosen as hash function as it was simple to implement and had the capability needed to transform the parameter data into a hash sum that would not be interpreted as code by the database.

4.1.1 Architecture

The architecture within the proxy was essential for having the communication between the web server and database to work. The proxy does not require much knowledge about the MySQL protocol in order for it to forward the communication correctly. The proxy is built on a main process. This process listens for any new incoming connections and binds the incoming connection with a port, creating a socket. The main process also creates the socket connection with the database. As soon as a connection has been established, two threads takes over the two socket connections.

Thread 1. A socket in Java 8 consists of two streams, output and input stream [18]. Thread 1 is responsible of reading all the data that is coming from the client and forwarding the data that has been read to the database. Whenever thread 1 receives a data stream from the client it gets parsed to see if it contains any queries, or if it is the connection phase between the client and the database. The MySQL protocol requires a connection phase in order for a connection to be established [19]. If the data stream is clean from SQL queries then it gets forwarded to the database immediately. If the data stream contains any SQL queries then it gets processed and thread 1 sends an updated SQL query to the database. If
thread 1 encounters any exceptions during the SQL query processing, then thread 1 and 2 exits and the communication is closed between the database and client.

**Thread 2.** Thread 2 is responsible of reading all the data that is coming from the database and forwarding the data that has been read to the client. The reason this thread exists is so we can have a full duplex communication between the client and the database. Otherwise the proxy server would have to know the sequence of when the database is writing and when the client is writing.

![Figure 4.1: Proxy Server Sequence Diagram](image)

### 4.1.2 Query Processing

Whenever the proxy server has received a data stream from a client it gets processed for SQL query checking. The proxy server will then extract any data from the stream that it interpret as parameter data. The proxy is using pattern matching regular expressions to find the parameter data within the SQL queries. Whenever a pattern match occurs the proxy extract the data that is matching the pattern.

Example: `SELECT * FROM users WHERE username = 'root' AND password = 'toor'`
The regular expression used by the proxy will find "root" and "toor" to match the pattern from the SQL query above. The data will then be extracted and hashed. The hashing algorithm used is MD5 within Java 8. The part of the SQL query that contains "root" and "toor" is then replaced with the hashed representation of its value.

Example: SELECT * FROM users WHERE username = '63a9f0ea7bb98050796b649e85481845' AND password = '7b24afe8bc80e548d66c4e7ff72171c5'

The parameter data within the SQL query above is now replaced with a MD5 hash of "root" and "toor". When extraction and hashing of the parameter data is complete a new pattern matching check is done on the query. This time any sequence of characters that represent comment syntax in SQL will be removed. The characters are found with a similar pattern match as the technique used above to find parameter data. The characters are then extracted from the SQL query.

Whenever a change in the SQL query has occurred a new payload size has to be calculated. The packets sent with the MySQL protocol includes a header [20]. This header consists of two field, payload length and sequence id. The payload length is the field that now has to be adjusted when the length of the SQL query has changed. The length of the payload is represented as hex values. The proxy server converts the SQL query from bytes into hex. The new payload length is then calculated and the packet header length field is then replaced with the new calculated value. The SQL query processing is now complete and the updated query is sent to the database. The processed and updated query sent to the database has MD5 hashed query parameter data, excluded comment syntax and a new packet header payload length field calculated.

4.2 Test Execution

A total of six tests were performed on the built system. The system contained the components previously mentioned in 3.2. The web server was either connected directly to the database or connected with the proxy that forwards every request to the database. The web server as well as the proxy were run on a virtual machine with Ubuntu as operating system. Both the web server and proxy were run on the same virtual machine. The MySQL database was run on a Windows 7 machine. The Virtual machine were run on the same Windows 7 machine that hosted the database. SQLMap is used to perform test 1 and 3. A bash script was used to send the requests to the web server and measure the time on test 5 and 6. Both SQLMap and the bash script was run on the same virtual machine that was hosting the proxy and web server.
Chapter 4. Implementation

The SQLMap commands below was used when when test 1 and 3 were performed. The SQLMap commands is an example OWASP provides on how to use SQLMap to automate the auditing [21].

- **V 2** Verbosity level: (0-6) [22]
- **-user-agent=SQLMAP** HTTP User-Agent header value [22]
- **-timeout=15** Seconds to wait before timeout connection [22]
- **-retries=2** Retries when the connection timeouts [22]
- **-batch** Never ask for user input, use the default behaviour [22]
- **-dbms=MySQL** Force back-end DBMS to this value [22]
- **-os=windows** Force back-end DBMS operating system to this value [22]
- **-level=5** Level of tests to perform (1-5) [22]
- **-risk=3** Risk of tests to perform (1-3) [22]
- **-banner** Retrieve DBMS banner [22]
- **-is-dba** Detect if the DBMS current user is DBA [22]
- **-dbs** Enumerate DBMS databases [22]
- **-tables** Enumerate DBMS database tables [22]
- **-technique=BEUSTQ** SQL injection techniques to use [22]
- **-s** Load session from a stored [22]
- **-flush-session** Flush session files for current target [22]
- **-t** DBMS database table(s) to enumerate [22]
- **-fresh-queries** Ignore query results stored in session file [22]

4.2.1 Test 1 with proxy connected

The purpose of the test is to get redirected to the welcome page from the login page by performing SQLI. The SQLI code was manually crafted and inserted within the form on the login page. The web server is connected with the proxy during this test.

Queries used:

1. SELECT * FROM users WHERE username=’’ OR 1=1 #AND password=’’
2. SELECT * FROM users WHERE username=’’ OR ’1’=’1’ #AND password=’’
3. SELECT * FROM users WHERE username=’admin’ AND password=’’ OR ’1’=’1’ – ’
4. SELECT * FROM users WHERE username=’admin’ AND password=’’ OR 1=1 – ’
4.2.2 Test 2 without proxy connected

The purpose of the test is to get redirected to the welcome page from the login page by performing SQLI. The SQLI code was manually crafted and inserted within the login form on the login page. The web server is directly connected with the database during this test.

Queries used:

1. SELECT * FROM users WHERE username=" OR 1=1 #’ AND password="’
2. SELECT * FROM users WHERE username=" OR ’1=’1 #’ AND password="’
3. SELECT * FROM users WHERE username='admin' AND password=" OR '1=1' - '
4. SELECT * FROM users WHERE username='admin' AND password=" OR 1=1 - '

4.2.3 Test 3 with proxy connected

The purpose of this test is to use SQLMap as an automated audit tool. The queries SQLMap use is boolean based blind, error based, time based blind, stacked queries, generic UNION queries and UNION queries. The web server is connected with the proxy during this test.

4.2.4 Test 4 without proxy connected

The purpose of this test is to use SQLMap as an automated audit tool. The queries SQLMap use is boolean based blind, error based, time based blind, stacked queries, generic UNION queries and UNION queries. The web server was directly connected with the database during this test.

4.2.5 Test 5 with proxy connected

The purpose of this test is to measure the average time it takes to make 1000*10 requests to the web server when it is connected to the proxy. The requests contains a valid username and password. The requests are performed by a bash script that sent 1000*10 requests in a loop while it records the time it take to perform the task.

Request used:
http://localhost/websida/login.php?username=admin&password=admin

4.2.6 Test 6 without proxy connected

The purpose of this test is to measure the average time it takes to make 1000*10 requests to the web server when it is directly connected with the database. The requests contains a valid username and password. The requests are performed by a bash script that sent 1000*10 requests in a loop while it records the time it take to perform the task.

Request used:

http://localhost/websida/login.php?username=admin&password=admin
Chapter 5

Results

Here are the results from the executed tests.

5.1 Test 1 with proxy connected

The list below shows the queries after the proxy has processed them.

Hashed Queries

1. SELECT * FROM users WHERE username = " OR '1'=c4ca4238a0b923820d509a6f75849b' AND password = "

2. SELECT * FROM users WHERE username = " OR '1'=c4ca4238a0b923820d509a6f75849b' AND password = "

3. SELECT * FROM users WHERE username = '21232f297a57a5a743894a0e4a801fc3' and password = " OR '1'=c4ca4238a0b923820d509a6f75849b' '

4. SELECT * FROM users WHERE username = '21232f297a57a5a743894a0e4a801fc3' and password = " OR '1'=c4ca4238a0b923820d509a6f75849b' '

All of the above queries generated syntax error response from the database after the proxy had hashed the query parameter data and removed the comments from the SQL code. Non of the crafted SQLI code granted access to the welcome page.
Chapter 5. Results

The table below describes the result and the database response from each tested SQL query.

<table>
<thead>
<tr>
<th>Query</th>
<th>Result</th>
<th>Database Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Unsuccessful</td>
<td>Syntax Error</td>
</tr>
<tr>
<td>2</td>
<td>Unsuccessful</td>
<td>Syntax Error</td>
</tr>
<tr>
<td>3</td>
<td>Unsuccessful</td>
<td>Syntax Error</td>
</tr>
<tr>
<td>4</td>
<td>Unsuccessful</td>
<td>Syntax Error</td>
</tr>
</tbody>
</table>

Table 5.1: Test 1 Results

5.2 Test 2 without proxy connected

All manually crafted SQLI code granted access to the welcome page. The queries got successfully injected and executed on the database.

The table below describes the result and the database response from each tested SQL query.

<table>
<thead>
<tr>
<th>Query</th>
<th>Result</th>
<th>Database Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Successful</td>
<td>Admin User</td>
</tr>
<tr>
<td>2</td>
<td>Successful</td>
<td>Admin User</td>
</tr>
<tr>
<td>3</td>
<td>Successful</td>
<td>Admin User</td>
</tr>
<tr>
<td>4</td>
<td>Successful</td>
<td>Admin User</td>
</tr>
</tbody>
</table>

Table 5.2: Test 2 Results
5.3 Test 3 with proxy connected

SQLMap reported the injection points, stacked and time based queries to be injectable. The other tested injection points was reported as not injectable or might not injectable.

<table>
<thead>
<tr>
<th>Injection points</th>
<th>SQLMap Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean Based</td>
<td>Not Injectable</td>
</tr>
<tr>
<td>Error Based</td>
<td>Not Injectable</td>
</tr>
<tr>
<td>Time Based</td>
<td>Injectable</td>
</tr>
<tr>
<td>Stacked Queries</td>
<td>Injectable</td>
</tr>
</tbody>
</table>

Table 5.3: Test 3 Results
Chapter 5. Results

5.4 Test 4 without proxy connected

SQLMap reported the injection points, stacked, boolean and time based queries to be injectable. The other tested injection points was reported as not injectable or might not injectable.

<table>
<thead>
<tr>
<th>Injection points</th>
<th>SQLMap Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean Based</td>
<td>Injectable</td>
</tr>
<tr>
<td>Error Based</td>
<td>Not Injectable</td>
</tr>
<tr>
<td>Time Based</td>
<td>Injectable</td>
</tr>
<tr>
<td>Stacked Queries</td>
<td>Injectable</td>
</tr>
</tbody>
</table>

Table 5.4: Test 4 Results
Chapter 5. Results

5.5 Test 5 with proxy connected

The result below is the average time measured after ten runs with the program.

Average time to perform 1000 requests to the webserver: \textbf{16.0869149988} seconds.

5.6 Test 6 without proxy connected

The result below is the average time measured after ten runs with the program.

Average time to perform 1000 requests to the webserver: \textbf{12.3729585974} seconds.

The table below lists the mean value, standard deviation and the effect size compared between test 5 and 6.

<table>
<thead>
<tr>
<th></th>
<th>Test 5</th>
<th>Test 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean Value</td>
<td>16.0869149988</td>
<td>12.3729585974</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>0.4995732892524</td>
<td>0.15264234917051</td>
</tr>
<tr>
<td>Effect Size r</td>
<td>0.9807855538164907</td>
<td>0.9807855538164907</td>
</tr>
</tbody>
</table>

Table 5.5: Test 5 & 6 results
Chapter 6

Analysis

From the manual testing in test 1 and 2 there is a significant difference in outcome. Without the proxy all of the the queries mentioned in the test case were successful and able to bypass the login on the website. With the proxy connected the result were the opposite. None of the queries were able to get redirected to the welcome page. This means the proxy were able to deny all SQL injections trying to bypass the login and that the proxy did the job to protect the web application against a possible attack.

When SQLMap was used to test the web application for vulnerabilities it was clear the manual testing was not enough to conclude that the proxy secured the web application. When testing with SQLMap, the scan was able to find vulnerabilities when the proxy was connected. The injection points found were stacked queries and time based queries. An example of a stacked query that could bypass the proxy is shown down below.

```sql
SELECT * FROM users WHERE username = 'admin' AND password = ''; DROP TABLE test; SELECT * FROM users WHERE username = 'admin'
```

![Figure 6.1: Output from proxy when a stacked query is bypassing the proxy](image)

After the proxy intercepted and handled the query it looked like this.

```sql
SELECT * FROM users WHERE username = '21232f297a57a5a7438940e4a801fc3' AND password = ''; DROP TABLE test; SELECT * FROM users WHERE username = '21232f297a57a5a7438940e4a801fc3'
```

This query is valid and will drop the table test. This shows that the proxy is still vulnerable to stacked queries.
Another injection point that was able to be exploited even with the proxy connected was time based queries. This injection point was used together with stacked queries. A query that used time based injection could bypass the proxy by manipulating the query to look like the following:

```
SELECT * FROM users WHERE username = 'admin' and password = ' or sleep(500); SELECT * FROM users WHERE username = 'admin'
```

```
SELECT * FROM users WHERE username = '21232f297a57a5a743894a0e4a801fc3' and password = ' or sleep(500); SELECT * FROM users WHERE username = '21232f297a57a5a743894a0e4a801fc3'
```

The query above has been processed by the proxy. The query is still valid and will execute the sleep command in the database making the response wait for 500 seconds. Without the proxy connected, 3 injection points were found using SQLMap. This means that the proxy eliminated one injection point. The injection point that the web application is vulnerable to when the proxy is not connected is Boolean based injections.

Test 5 showed an increased amount of time taken for the web application to handle HTTP requests that required database connections. Comparing test 5 and 6 shows that the proxy is slowing down the speed. The average value of 1000 requests sent to the web application was close to 4 seconds slower when the proxy server was used. Comparing the standard deviation between test 5 and 6 indicates that the speed is dispersed between a wider range of values on test 5. While the speed is more closely to the mean value on test 6. The effect size can be interpreted as large [23]. This means that the proxy has a big impact in the speed, causing the system to slow down.
When trying to get redirected to the login by using SQL injection it was clear that the proxy helped with the security on the web application. This was tested in test 1 and 2 where four SQL queries were tested to demonstrate if the proxy had an effect on the security. The proxy succeeded in test 1 and was able to deny all of the SQL injections tested making sure that no unauthorized access was granted. Without the proxy all of the SQL injections was able to bypass the login. In our research question we asked if the proxy would decrease the number of injection points in the web application. This manual test showed that it helped to have the proxy connected to the web application to prevent the tested SQL injections.

During test 3 and 4 an automatic penetration testing tool called SQLMap were used. SQLMap were able to find vulnerabilities on the web application. The two injection points was found when the web server was connected to the proxy. The injection points were stacked and time based queries. These injections were missed in the manual test cases. When SQLMap scanned the web application without the proxy it was able to find three injection points. These were stacked, boolean and time based queries. This shows that proxy had decreased the injection points from three to two. This strengthens the research question that the web server has less injection points when connected to the proxy than without it.

The web application was also tested for how long it would take to make 1000 requests. In test case 5 and 6, a bash script was used to send login requests to web server. The time measured how long it took to get all responses back from the web server. This showed that the web server was 2.6 seconds slower on handling 1000 requests when it was connected to the proxy. In the research question we asked if the proxy had a dilatory effect on the system. Test 5 and 6 proved that the proxy had slowed down the request handling, giving the system a dilatory effect.

For future work the proxy could be improved. The proxy still have problems with SQL injection and a big part of that is stacked queries. It is hard to separate what is legitimate user input from what is malicious as both are valid SQL queries. A
future work could be on how to detect stacked queries and eliminate it from the actual query.

When the proxy parses out the user input it also hashes the user input with a MD5 hash function. Work could be done on researching different techniques to incapacitate the query parameters, such a encryption. An encryption method makes it possible to reverse the data to its original form. This can increase the usability of the proxy since it makes is possible to extract the data to its original form. An hash function limits this possibility since the data cannot be reverted.

How does the proxy limit the way the web application can be run. Can the proxy handle huge amount of concurrent users or will requests go lost? If the proxy start to struggle with many concurrent users is there a way to tackle this problem or do you need to setup more proxies to help reduce the load. To use the proxy the information also has to be hashed in the database. How does this limit the way a developer can use the database to get and store information. If there is limitations on how the database can be used, is there a way to get around this problem and still use the proxy.
Bibliography


[14] S Fouzul Hidhaya and Angelina Geetha. Intrusion protection against sql

[15] Vrushali S Randhe, Archana B Chougule, and Debajyoti Mukhopadhyay. Re-
verse proxy framework using sanitization technique for intrusion prevention

[16] Anyi Liu, Yi Yuan, Duminda Wijesekera, and Angelos Stavrou. Sqlprob: a
proxy-based architecture towards preventing sql injection attacks. In *Pro-
cedings of the 2009 ACM symposium on Applied Computing*, pages 2054–


[18] Oracle socket class. [https://docs.oracle.com/javase/7/docs/api/
java/net/Socket.html](https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html).


page_protocol_basic_packets.html#sect_protocol_basic_packets_describing_packets](https://dev.mysql.com/doc/dev/mysql-server/latest/page_protocol_basic_packets.html#sect_protocol_basic_packets_describing_packets).


[23] Catherine O Fritz, Peter E Morris, and Jennifer J Richler. Effect size esti-
mates: current use, calculations, and interpretation. *Journal of experimental