UNIVERSITY OF HERTFORDSHIRE

Faculty of Science, Technology and the Creative Arts

Modular BSc Honours in Computer Science

6COM0282 – Computer Science Project

Final Report

April 2012

SQL INJECTION DETECTION AND SIMULATION (SQILD)

A. Hoque

Student ID: 09211877

Supervised by: David Bowes

Abstract

This report describes the research, design and implementation of a software to detect SQL security vulnerabilities (SQL Injections) in web applications coded in PHP and JAVA. The software aims to provide a user the ability to run automatic static analysis and SQL attack simulations on web applications.

The report evaluates the static analysis and the simulator against other freely available automatic detection tools. Although the software still does not have a stable version but still could still used to learn the techniques, complications and hurdles of building an automatic detection tool.

Table of Contents

1 Introduction 6

1.1 Report Structure 7

2 Research 8

2.1 Current State of Web Application Security 8

2.2 What is SQL Injection? 10

2.3 Automatic SQL Injection Detection Tools 12

2.3.1 Static Code Analysis (SCA) 12

2.3.2 Simulating SQL Injection 14

2.4 Limitations of Automatic Detection 15

3 Software Design Decisions 16

3.1 Complete Software Class Diagram 18

4 Static Analyzer 20

4.1 Requirements Specification 20

4.1.1 Functional Requirements 20

4.1.2 Core Functional Requirements 20

4.1.3 Non Functional Requirements 20

4.2 Design 21

4.2.1 Tainted Variables 22

4.2.2 SQL Statements 23

4.3 Static Analyzer Class Diagram 25

4.4 Class Diagram Explained 26

4.5 Analyzer Activity Diagram 27

4.6 Implementation 28

4.7 Testing 29

4.7.1 Testing Functional Requirements 29

4.7.2 Testing Core Functional Requirements 30

4.7.3 Testing Non - Functional Requirements 32

5 Simulator **33**
5.1 Requirements Specification 33
5.1.1 Functional Requirements 33
5.1.2 Core Functional Requirements 33
5.1.3 Non Functional Requirements 33
5.2 Simulation Design 34
5.3 Simulator Class Diagram 38
5.4 Class Diagram Explanation 39
5.5 Simulator Activity Diagram 40
6 Implementation **41**
7 Test **42**
7.1 Testing Functional Requirements 42
7.2 Testing Core Functional Requirements 42
7.2.1 Parse all forms in a webpage and output Injected Forms 42
7.2.2 Simulator detects all possible SQL vulnerable input points 43
This test was conducted on two vulnerable web applications Multidae and WebGoat. 43
7.3 Testing Non Functional Requirements 43
8 Software Evaluation **44**
9 Reflection **45**
10 Bibliography **46**
11 Appendix A **47**
11.1 Static Analyzer Test 47
11.1.1 Screen Shot 1 47
11.1.2 Screen Shot 2 47
11.1.3 Screen Shot 3 48
11.1.4 Screen Shot 4 48
11.1.5 Screen Shot 5 49
11.1.6 Screen Shot 7 50

12 Appendix B 51

12.1 Analyzer Package 51

12.1.1 Analyze.class 51

12.1.2 Scanner.class 53

12.1.3 JavaScanner.class 54

12.1.4 PhpScanner.class 59

12.2 Simulator Packages 64

12.2.1 Simulate.class 64

12.2.2 Form.class 66

12.2.3 SqlSignatures.class 70

12.2.4 Response.class 73

12.2.5 AnalyseResponses.class 75

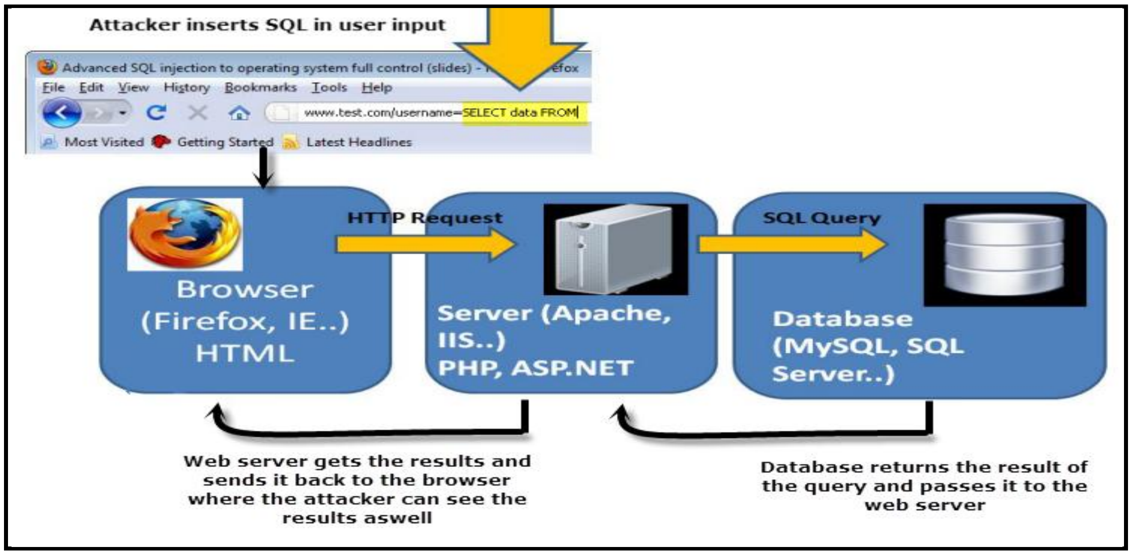
1 Introduction

Building a website or learning a popular programming language such as C++, Java, PHP, and JavaScript is quite easy nowadays, as there are huge amounts of tutorials, code snippets and resources available online. I started learning web development during the first year of my degree and after a few months I was confident enough to work on an actual website so I searched for some freelance work and took on some small projects which involved writing a few PHP scripts to handle client requests from the server. Whilst working on these projects I came across a few security vulnerabilities commonly known as **SQL (Structures Query Language) Injections**.

SQL injection is a type of security vulnerability where if the input from a client is not validated and sanitized by the web server it allows an attacker to pass SQL commands as valid inputs to extract or manipulate data on the back end database(Fig 1).

I had a few small and easy scripts to fix so it was quite easy to make sure all user inputs were validated but most servers usually accept a lot of data from the client and physically checking for vulnerable inputs in source code could be tedious, time consuming and sometimes complicated. My aim is to build a software which will automatically detect SQL injections to secure a web application and make a developers life a little bit easier.

Fig 1 Example of how an attacker performs a simple SQL injection


Attacker inserts SQL in user input
Advanced SQL injection to operating system full control (slides)
File Edit View History Bookmarks Tools Help
www.test.com/username=SELECT data FROM
Most Visited Getting Started Latest Headlines
HTTP Request
SQL Query
Browser (Firefox, IE..) HTML
Server (Apache, IIS..) PHP, ASP.NET
Database (MySQL, SQL Server..)
Web server gets the results and sends it back to the browser where the attacker can see the results aswell
Database returns the result of the query and passes it to the web server

1.1 Report Structure

This report first explains the current state of web application security and then the research that has been done in order to develop a static analysis and simulation tool.

Please note, because the software provides static analysis and simulation for detection so the report has been divided between these two techniques.

This report first explains the overall software design and then it explains each techniques design, test, implementation, and testing separately.

At the end I have evaluated the overall software and reflected upon what I have learnt from this experience

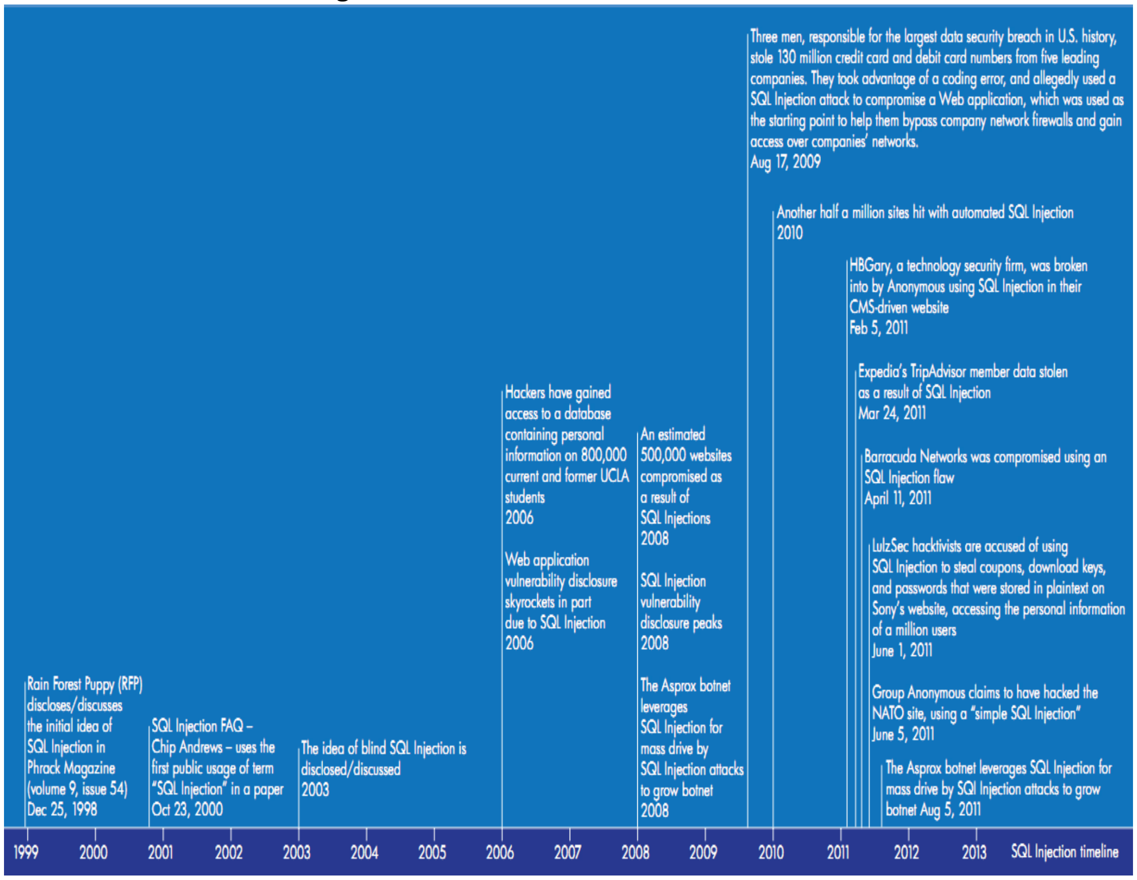
The appendices mostly contain screen shots and the source code.

2 Research

2.1 Current State of Web Application Security

SQL injection is not a new security problem; it was first brought to people's attention by RainForestPuppy in Phrack magazine in 1998. Since then there have been many high profile and expensive hacks due to SQL vulnerabilities in web applications (Hewelt Packard, 2011)(Figure 2).

Fig 2 Timeline and Evolution of SQL attacks



Rain Forest Puppy (RFP) discloses/discusses the initial idea of SQL Injection in Phrack Magazine (volume 9, issue 54) Dec 25, 1998

SQL Injection FAQ – Chip Andrews – uses the first public usage of term "SQL Injection" in a paper Oct 23, 2000

The idea of blind SQL Injection is disclosed/discussed 2003

Hackers have gained access to a database containing personal information on 800,000 current and former UCLA students 2006

Web application vulnerability disclosure skyrockets in part due to SQL Injection 2006

An estimated 500,000 websites compromised as a result of SQL Injections 2008

SQL Injection vulnerability disclosure peaks 2008

The Asprox botnet leverages SQL Injection for mass drive by SQL Injection attacks to grow botnet 2008

Three men, responsible for the largest data security breach in U.S. history, stole 130 million credit card and debit card numbers from five leading companies. They took advantage of a coding error, and allegedly used a SQL Injection attack to compromise a Web application, which was used as the starting point to help them bypass company network firewalls and gain access over companies' networks. Aug 17, 2009

Another half a million sites hit with automated SQL Injection 2010

HBGary, a technology security firm, was broken into by Anonymous using SQL Injection in their CMS-driven website Feb 5, 2011

Expedia's TripAdvisor member data stolen as a result of SQL Injection Mar 24, 2011

Barracuda Networks was compromised using an SQL Injection flaw April 11, 2011

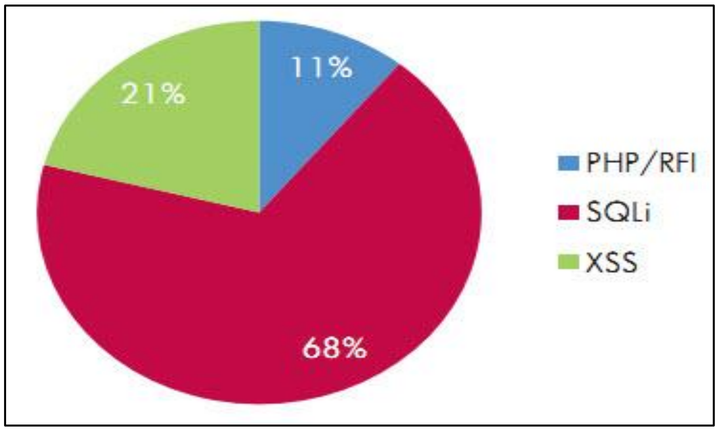
LulzSec hacktivists are accused of using SQL Injection to steal coupons, download keys, and passwords that were stored in plaintext on Sony's website, accessing the personal information of a million users June 1, 2011

Group Anonymous claims to have hacked the NATO site, using a "simple SQL Injection" June 5, 2011

The Asprox botnet leverages SQL Injection for mass drive by SQl Injection attacks to grow botnet Aug 5, 2011

The tools and techniques to detect and avoid SQL injections have improved tremendously over the years. Most of the languages used to code servers have built in functions to construct safe SQL statements. The issue of SQL vulnerabilities and its consequences has been clearly explained in online communities, tech magazines, newspapers etc. Even after all this publicity, according to a recent security report from HP more than half of web security attacks between Jan-June 2011 were done using SQL injections (Figure 3).

Fig 3 Web attacks Pie chart **(**Heweltt Packard**,** 2011**)**



So why do we still keep making the same mistakes which allow SQL injection? Well from my experience as a programmer who used write insecure code, there are a few reasons;

- Website tutorials and code examples on online forums are insecure which is used by inexperienced programmers without the understanding of, security aspects regarding the code.
- Programming and website development books completely lack or hardly emphasize on the topic of web application security or defensive programming.
- Even extremely popular web development tutorials website w3schools doesn’t mention SQL injections or web application security in general.

2.2 What is SQL Injection?

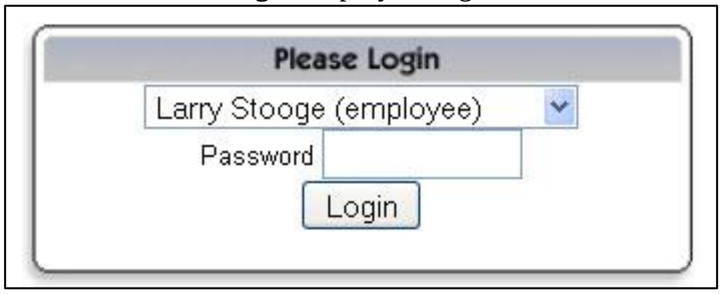
As I mentioned previously SQL injections are simply security holes created when inputs are not validated on the web server thus allowing attackers to insert SQL commands in user inputs to manipulate back end database.

It is extremely important to be clear that the validation of user inputs should be done on the **Server**. Validation on the client side is not enough to prevent SQL injections because the data sent to the server using HTTP protocol can be easily manipulated by an attacker on the client side.

Here I will try and explain what actually happens in the code of a web server during a SQL attack.

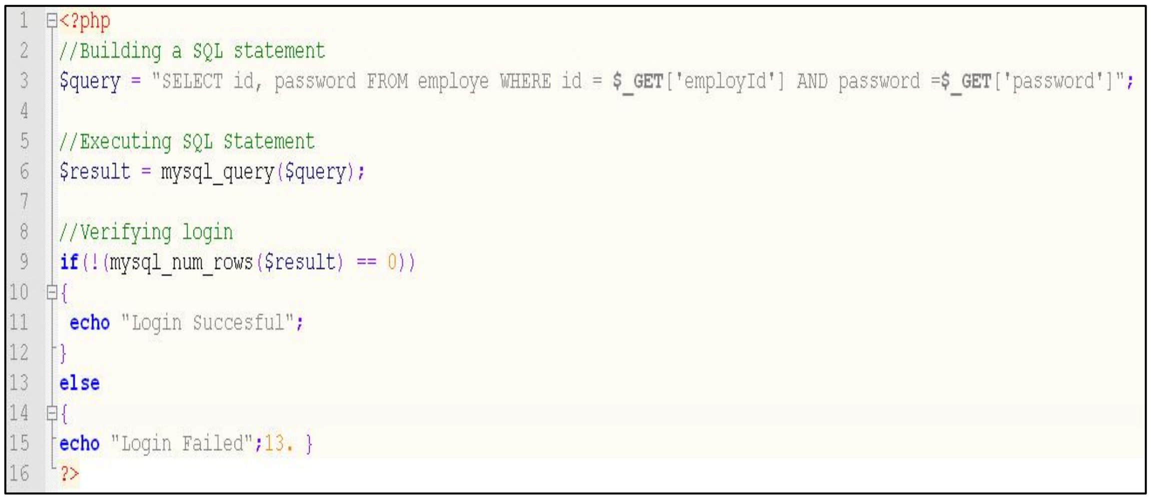
The following example is a scenario of an employee login verification (Figure 4). The website uses a PHP web server which has a MySQL back end database.

Fig 4 Employee Login



The PHP code written to build a SQL statement and query the database is shown below

Code Snippet 1 Login code snippet



```php
<?php
//Building a SQL statement
$query = "SELECT id, password FROM employe WHERE id = $_GET['employId'] AND password =$_GET['password']";

//Executing SQL Statement
$result = mysql_query($query);

//Verifying login
if(!(mysql_num_rows($result) == 0))
{
 echo "Login Succesful";
}
else
{
echo "Login Failed";13. }
?>
```

As you can see in Code Snippet 1 the server receives the input variables in HTTP GET methods shown below have not been validated

```
$_GET['user'], $_GET['password']
```

Lack of input validation allows an attacker to insert an inline SQL injection string

```
val' or '1'='1'
```

The server receives the input and builds a SQL statement as shown below

```
SELECT id, password FROM employe WHERE id = 106 AND password = val' or
'1'= '1';
```

The statement shown will command the database to return the id and password row of the employee table where

```
'id = 106 AND password = val'  OR '1'= '1'
```

The database will not be able to find a row with the specified parameters because it doesn't exist but the **OR** operand will cause the compiler to always return true because one always equals to one, this will cause the database to return all the rows in the table.

The server will receive the data and store it in the `$result` variable as an array.

The `"if"` statement on line seven (Code Snippet 1) checks to see whether the result array size is equal to zero or not, if the size is greater than zero it will reply `"Login Successful"` to the client and if the result array is empty then the server will reply `"Login unsuccessful"`.

The inline SQL injection shown above is one of the many possible ways to formatting and manipulating a SQL statement. These will be discussed in detail later in the report.

Although a SQL injection is possible due to the lack of input validation there are other problems with the code such as exceptions from the server are not being caught, which will allow an attacker to cause an unexpected error and possibly see the error message from the database that could reveal sensitive information(Database Name, Version etc).

2.3 Automatic SQL Injection Detection Tools

Physically reviewing the code and testing web applications is tedious, error prone and at times extremely complicated, thus leading to the creation of tools that automatically detect errors in the code. Detecting SQL vulnerabilities can be done using two automated techniques Static Code Analysis and Simulating SQL injections, these techniques have been discussed in detail below

2.3.1 Static Code Analysis (SCA)

The term Static Code Analysis literary means analyzing source code without executing it. Basically static analysis is just like physically reading lines of code and finding errors but the difference is, static analysis does it faster. Table 1 lists some good Static code analyzers available online to detect SQL injections. Static analysis has many applications other than detecting security errors such as Syntax Highlighting, Style Checking, and Automated Documentation etc.

Table 1 List of Static Code Analyzers

Tool	**Supported Languages**	**Availability**	**Last Update**	**Supported OS**
CodeSecure	ASP.NET, VB.NET, C#, Java/J2EE, JSP, EJB,PHP, Classic ASP and VBScript	Commercial and Free	Apr-11	Windows, Linux
Pixy	PHP 4	Free	Jul-07	Windows, Linux
RIPS	C, C++, Perl, PHP and Python	Free	Jan-09	Windows Linux
LAPSE +	JAVA	Free	Mar-11	Windows, Linux

As mentioned before SCA is just like a programmer physically reading the code so in building a SCA to understand source code just like a programmer is an extremely complicated task which is achieved by using variations of different high level techniques. Most of the tools listed in Table 1 have their own variation of algorithms and are developed differently but they all have two common, extremely important and main steps in developing a SCA tool which is Parsing and Analysis.

2.3.1.1 Parsing

Parsing is a mechanism of analyzing code and producing tokens of a programming language by using the specified grammar, the tokens are then arranged in a hierarchical form know as parse tree. To better understand parsing, consider a programming language such as Java.

In Java two very basic rules are

- A variable is always defined as a specific type (String, int, Boolean etc) before its name
- The symbol **“;”** means end of the line.

Code Snippet 2 Java Example

```
String example = “test”;
String name = “Ace";
```

The grammar specified to a Java Parser will be the two rules stated above and the parser will scan the source code and define variables “example” and “name” as a token of type String and their corresponding value will be “test” and “Ace”.

A Parser can be built using a language development tool like ANTLR (Another Tool For Language Recognition), Yacc (Yet Another Compiler Compiler) etc. The list of language development tools is quite long and each of them is best suited for a particular type of programming language.

2.3.1.2 Analysis

In the Analysis procedure the system will keep track of all

- Variables state and its properties at every line in the code where the variable is being used.
- Functions (or methods) in the code and the effect of the function when it is used.

At the analysis procedure we have to specify what the static analysis is supposed to do such as in the case of detecting SQL injections we will specify the system to look for all the HTTP methods (GET, POST, REQUEST, COOKIES) in the source code and inform us if the data in the source code is being passed to another variable (also called sink) without sanitation, changed or being used at any point in the source code.

2.3.2 Simulating SQL Injection

Simulation is a complete opposite of static analysis. In static analysis the code of the web application is tested without being executed whereas in SQL simulation actual attacks are performed on the web application while it is running. Simulation gives a developer a closer look at a web servers working when it is hit by an actual SQL injection.

In most cases of website hacks done using SQL injection the attacker uses a browser, poking the website and trying to find a point of entry to insert and execute a SQL command. The art of constructing and inserting a SQL statement along with the input is highly complicated and requires a lot of patience.

It is extremely important to understand the difference between detecting a SQL vulnerability and Exploiting the vulnerability. For example in the scenario explained in Section 2.2 (page 10), the password input field allows an attacker to insert a SQL injection so we have detected a SQL vulnerability (a possible way of inserting and executing SQL commands) but using this vulnerability to extract information from the database is defined as exploiting the SQL vulnerability.

Detection of SQL injection via simulation includes four main steps as described below

- Identify all points of data entry
- Identify the type of data entry(String, number, or Boolean)
- Prepare data for injection
- Analyze the response from the server to conclude weather a SQL injection is possible.

The first two steps mentioned above are achieved by using a HTML parser to extract information from the website such as forms, input fields and their properties. The data collected from the parser is then used to prepare data that will be sent to the server. The prepared data is injected with inline SQL injection as shown in example in Section 2.2(Page 9).

The last and crucial step in detecting SQL vulnerability is analyzing the response to confirm the presence of a possible point of SQL injection. Usually if the response from server is a HTTP 500 error, HTTP 302, or errors generated by the database it can be concluded that a SQL injection is possible.

2.4 Limitations of Automatic Detection

Automatic detection tools available to us right now are extremely powerful and efficient but they do have limitations.

In instances where the user input has been validated and sanitized, a static analysis will conclude that there are no vulnerabilities but this might not be true because the validation process itself might be poorly coded which a static analyzer will not be able to detect. A few static analyzers such as Pixy have tried to improve on this problem by adding functionality of checking some common techniques of validation; but it is still not the perfect solution as web applications use varieties of validation methods.

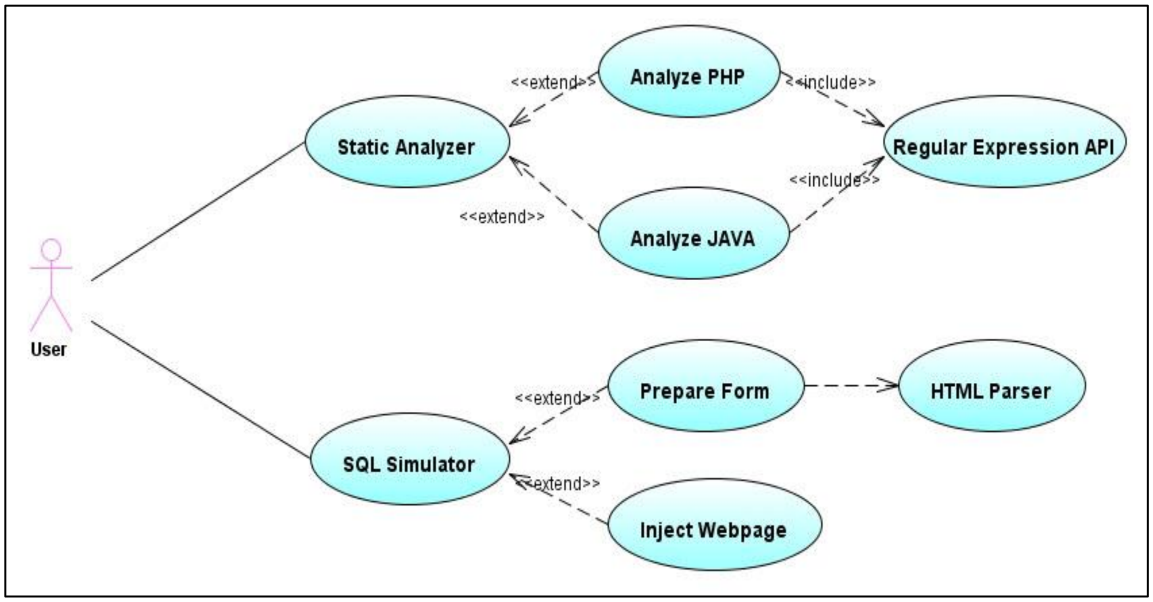
Detecting the existence of a SQL vulnerability via simulation is not always easy and in some cases impossible if the server never responds with an error message, but hackers have got around this problem by using a form of SQL attack commonly known as Blind SQL Injection. Unfortunately performing all forms of blind injections via simulation is not possible due the complex nature of these attacks.

Results from an automatic tool should not be used to declare that an application is free of security holes. Automatic tools are mostly used because commercial software's usually contain more thousand lines of codes.

3 Software Design Decisions

The software has been divided into two modules; Static Analysis and Simulation. Both these modules are completely independent and don't share common functions or classes but they have the same goal, detecting SQL injections.

Use Case 1 Complete Software Overview



The Use Case diagram above shows a general overview of the whole software. The Static Analyzer can scan SQL vulnerabilities in PHP and Java source code but the analyzer does not use a parser like other traditional static analyzers.

Initially I tried searching for a readymade open source parser. For Java I did find some parsers but their grammars files have not been updated in over a year. In the case of PHP I could not find a suitable open source parser at all other than an extremely good compiler called PHC which would only run in a UNIX environment but as the goal of my project was to build a software which could be run on windows and linux platforms, I could not use PHC.

After failing to find a readymade parser I did think about trying to write my own parser for both PHP and Java but as you can see from my research(Section 2.3.1, page 12) building a parser is a sensitive task which requires time and the knowledge of using a language development tool. Although I could jump into learning a LDT (language development tool) but that would require some time which I did not have because by the time I finished my research, I only had enough time to finish the whole software

Eventually after discussing this issue with my supervisor I decided to use the Regular Expression API in Java to analyze the source code for SQL vulnerabilities.

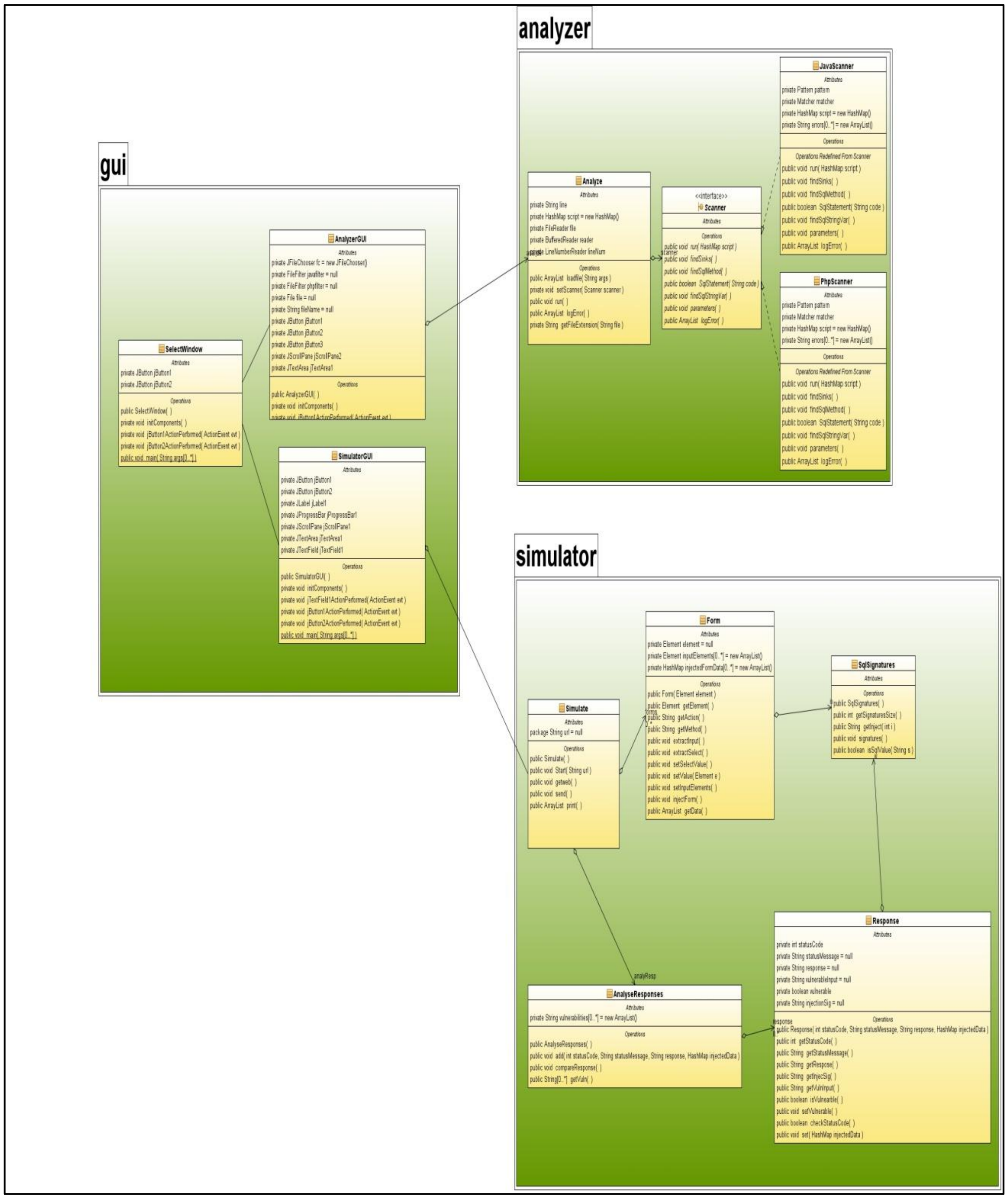
Using Regular Expressions is easy and there are good tools (such as Regex Buddy) available online that can immensely help in building complicated regular expression but unlike a parser, building a regular expression can get messy, even a single extra space between words will cause a mismatch.

Initially it seems like a good alternative to a parser because I was only building a software to detect SQL injections but the consequences of using the regular expression will be seen later in the report in my evaluation of the static analyzer.

I also had a big problem with the HTML parser that I was using for the simulator because I discovered the problem right at the end of the testing phase. The parser that I was using before was called HTML Unit. The problem was it could not parse a few web pages. Eventually after some research into this problem I realized that HTML Unit is not used for parsing web pages, it is used to test web pages automatically so it mimics web browsers, so just like a web browser it will try and run flash or other technologies used on the web pages. The user of the software had to provide the support for the required technologies for the website being tested and each website comes with its own set of requirements. I did some digging and I eventually found a full fledged HTML parser know as JSoup. This parser has a java API and is extremely easy to use. Although it took me a while to rewrite my code to use the Jsoup API but it did solve my problem.

3.1 Complete Software Class Diagram

Class Diagram 1 Complete Software


gui
SelectWindow
Attributes
private JButton jButton1
private JButton jButton2
Operations
public SelectWindow( )
private void initComponents( )
private void jButton1ActionPerformed(ActionEvent evt)
private void jButton2ActionPerformed(ActionEvent evt)
public void main(String args[0..*])
AnalyzerGUI
Attributes
private JFileChooser fc = new JFileChooser()
private FileFilter javafilter = null
private FileFilter phpfilter = null
private File file = null
private String fileName = null
private JButton jButton1
private JButton jButton2
private JButton jButton3
private JScrollPane jScrollPane2
private JTextArea jTextArea1
Operations
public AnalyzerGUI( )
private void initComponents( )
private void jButton1ActionPerformed(ActionEvent evt)
SimulatorGUI
Attributes
private JButton jButton1
private JButton jButton2
private JLabel jLabel1
private JProgressBar jProgressBar1
private JScrollPane jScrollPane1
private JTextArea jTextArea1
private JTextField jTextField1
Operations
public SimulatorGUI( )
private void initComponents( )
private void jTextField1ActionPerformed(ActionEvent evt)
private void jButton1ActionPerformed(ActionEvent evt)
private void jButton2ActionPerformed(ActionEvent evt)
public void main(String args[0..*])
analyzer
Analyze
Attributes
private String line
private HashMap script = new HashMap()
private FileReader file
private BufferedReader reader
private LineNumberReader lineNum
Operations
public ArrayList loadfile(String args)
private void setScanner(Scanner scanner)
public void run( )
public ArrayList logError( )
private String getFileExtension(String file)
analyze
scanner
<<interface>>
Scanner
Attributes
Operations
public void run(HashMap script)
public void findSinks( )
public void findSqlMethod( )
public boolean SqlStatement(String code)
public void findSqlStringVar( )
public void parameters( )
public ArrayList logError( )
JavaScanner
Attributes
private Pattern pattern
private Matcher matcher
private HashMap script = new HashMap()
private String errors[0..*] = new ArrayList()
Operations
Operations Redefined From Scanner
public void run(HashMap script)
public void findSinks( )
public void findSqlMethod( )
public boolean SqlStatement(String code)
public void findSqlStringVar( )
public void parameters( )
public ArrayList logError( )
PhpScanner
Attributes
private Pattern pattern
private Matcher matcher
private HashMap script = new HashMap()
private String errors[0..*] = new ArrayList()
Operations
Operations Redefined From Scanner
public void run(HashMap script)
public void findSinks( )
public void findSqlMethod( )
public boolean SqlStatement(String code)
public void findSqlStringVar( )
public void parameters( )
public ArrayList logError( )
simulator
Simulate
Attributes
package String url = null
Operations
public Simulate( )
public void Start(String url)
public void getweb( )
public void send( )
public ArrayList print( )
Form
Attributes
private Element element = null
private Element inputElements[0..*] = new ArrayList()
private HashMap injectedFormData[0..*] = new ArrayList()
Operations
public Form(Element element)
public Element getElement( )
public String getAction( )
public String getMethod( )
public void extractInput( )
public void extractSelect( )
public void setSelectValue( )
public void setValue(Element e)
public void setInputElements( )
public void injectForm( )
public ArrayList getData( )
forms
0..*
SqlSignatures
Attributes
Operations
public SqlSignatures( )
public int getSignaturesSize( )
public String getInject(int i)
public void signatures( )
public boolean isSqlValue(String s)
analyResp
AnalyseResponses
Attributes
private String vulnerabilities[0..*] = new ArrayList()
Operations
public AnalyseResponses( )
public void add(int statusCode, String statusMessage, String response, HashMap injectedData)
public void compareResponse( )
public String[0..*] getVuln( )
response
Response
Attributes
private int statusCode
private String statusMessage = null
private String response = null
private String vulnerableInput = null
private boolean vulnerable
private String injectionSig = null
Operations
public Response(int statusCode, String statusMessage, String response, HashMap injectedData)
public int getStatusCode( )
public String getStatusMessage( )
public String getRespose( )
public String getInjecSig( )
public String getVulnInput( )
public boolean isVulneartble( )
public void setVulnerable( )
public boolean checkStatusCode( )
public void set(HashMap injectedData)

As you can see from the class diagram provided above the software is divided into three packages

- **gui :** This package contains three GUI's to control the complete software.

  SimulatorGUI class is used to visually perform simulations and displays all outputs of the simulation. It will also allow a user to switch to Code Analysis.

  AnalyzerGUI class is used to visually perform code analysis and display all outputs of the analysis. It will also allow the user to switch to Simulation

  SelectWIndow is a small window that will start when the user start the software. It will allow a user to either open the Code Analyzer or the Simulator according to his/her preference

- **analyzer :** This package contains all the classes and code that are used to run static analysis. All the analysis operations are done in this package and then the output is passed to the AnalyzerGUI class in the GUI package. The AnalyzerGUI class displays this output to the user.

- **simulator** : This package contains all the classes and code that are used to run simulations. All the simulation operations are done in this package and then the output is passed to the SimulatorGUI class in the gui package. The AnalyzerGUI class displays this output to the user

The class diagram in the previous page clearly shows that any future changes will be easy to make as the coupling between classes is extremely low. During the development of this software it has been extremely easy to add and delete code due the amount of low coupling between classes but this design was achieved because of an iterative approach that has been taken to design and build this software. Although I did do the required research but still I have never built a static analyzer or a simulator before so I had to attain the knowledge and skills as I worked through each and every problem.

4 Static Analyzer

4.1 Requirements Specification

4.1.1 Functional Requirements

- Analyze PHP source code
- Analyze Java source code
- Allow the user to select the source code file from the system where this software is being deployed
- Display (Output) number of vulnerable user inputs detected in the source code
- Display (Output) vulnerable lines of code along with the line number
- Display (Output) number of sinks detected in the source code
- Display (Output) each sink and the line number where the sink was detected in the source code

4.1.2 Core Functional Requirements

The following requirements are the core applications of the static analyzer. These requirements has to be implemented

- Detect SQL vulnerabilities in PHP code
- Detect SQL vulnerabilities in Java code

4.1.3 Non Functional Requirements

- The software should be able to run on Linux and Windows
- The GUI should be simple and easy to use

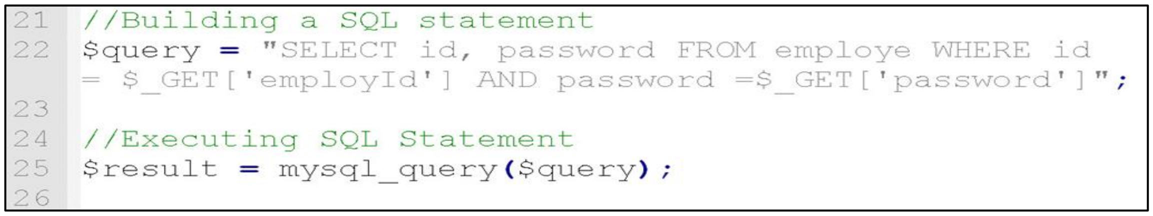
4.2 Design

Both PHP and Java share a few similarities when it comes to using SQL

- A statement can be built as a string and then be executed by providing a reference to the variable holding the string or a statement can be directly passed to the execution method to be executed.
- The parameters of a SQL statement can either be direct user inputs or references to the variables holding the user inputs.
- Both languages have built in methods to work with SQL statements.

The Code Snippets shown below support the observations made above.

Code Snippet 3 PHP SQL Example

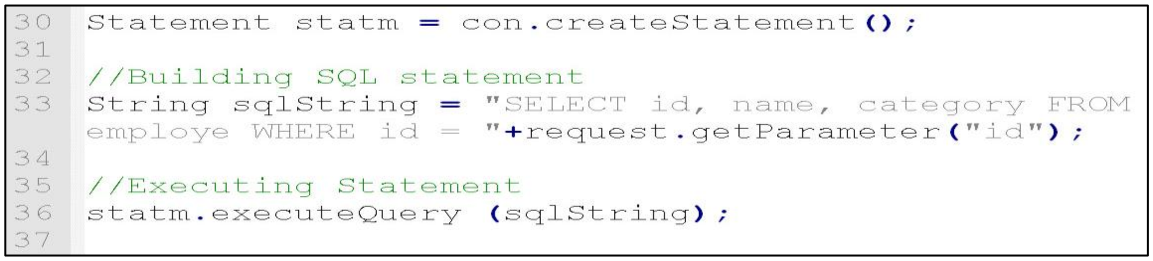


```
//Building a SQL statement
$query = "SELECT id, password FROM employe WHERE id = $_GET['employId'] AND password =$_GET['password']";

//Executing SQL Statement
$result = mysql_query($query);

```

Code Snippet 4 Java SQL Example



```
Statement statm = con.createStatement();

//Building SQL statement
String sqlString = "SELECT id, name, category FROM employe WHERE id = "+request.getParameter("id");

//Executing Statement
statm.executeQuery (sqlString);

```

After examining some more Java and PHP servers I can say that most of the SQL vulnerable PHP or Java source code files have the first two patterns mentioned above.

Below I have explained the regular expressions that I have used and how the expression is in detecting SQL vulnerability patterns.

4.2.1 Tainted Variables

Variables that have been passed directly from HTTP request methods without sanitation or validation are called tainted variables.

Examples

```
PHP  : $name = $_GET['name'];

Java : String name = request.getParameter[‘name’];
```

To detect lines of code like the examples shown above I have constructed the following regular expression

Regular Expression

- Expression to detect tainted variables in PHP

```
Expression 1 : "(\\$\\w+) *?= *?(' *?\\$_|\"
*?\\$_|\\$_)(GET|POST|REQUEST|COOKIE|SESSION)"
```

- Expression to detect tainted variables in JAVA

```
Expression 2 : "(\\w+) *?= *?\\w+?\\.(getParameter|getParameterValues|getHeader
|getHeaders|getRequestedSessionId|getCookies|getValue|getAttribute)"
```

The analyzer will read each line from the source code and then use the Regex API to compare the expression with the line. If a match is found the line number and the line is saved and displayed as vulnerable.

4.2.2 SQL Statements

SQL statements are where direct HTTP requests or tainted variables are most likely to be used, so first the analyzer will search built in methods that execute a statement on the database.

Examples

```
PHP  : $result = mysql_query("SELECT * table WHERE id = 1");

Java : ResultSet rs = stmt.executeQuery("SELECT * FROM Cust") ;
```

To detect the SQL execution method in a line of code like the examples shown above I had to construct the following regular expressions

Regular Expression

- Expression to detect statement execution methods in PHP

```
Expression 3 : "(mysql|mysql_db)_query\\((.+?)\\)"
```

- Expression to detect statement execution methods in Java

```
Expression 4 : "(executeQuery\\(|executeUpdate\\(|execute\\()(.+?)\\)"
```

The analyzer will read each line in the code and look for a match, if a match is found then the line number is recorded along with the line of code. The regular expressions are built to output the string inside the green highlighted brackets. The string can either be a name of the variable that holds the SQL statement or the SQL statement itself.

The analyzer will use the regular expression shown below to verify the string inside the green highlighted brackets.

- Expression is same for PHP and JAVA

```
Expression 5 : "\\\"(SELECT|INSERT|UPDATE|DROP)"
```

If the string between the brackets matches the expressions then the analyzer will search this line for HTTP request method

- Expression to detect HTTP request methods in a line written in PHP

```
Expression 6 : "\\$\\_(POST|GET|REQUEST|SESSION|COOKIE) *?\\( *?' *?\\$\\w*?
*?'*?\\)"
```

- Expression to detect HTTP request methods in a line written in JAVA

```
Expression 7 : "(getParameter|getParameterValues|getHeader|getHeaders
|getRequestedSessionId|getCookies|getValue|getAttribute)\\s*?\\(.+?\\) *?\\+"
```

If the string between the green highlighted brackets shown in Expression 3&4 are not SQL statements then the analyzer will assume it is the name for the variable that holds the SQL statement as string. The analyzer will use the following expression to find the line where the variable found has been assigned the SQL statement as String

- Expression to detect HTTP request methods in a line written in PHP

```
Expression 8 : "\\"+value+" *?= *?\" *?(SELECT|UPDATE|INSERT|DROP)"
```

- Expression to detect variable and the SQL statement assigned as string

```
Expression 9 : "(" + value + " *?= *?\\( *?\" *?(SELECT|UPDATE|INSERT|DROP))|
("+ value +" *?= *?\" *?(SELECT|UPDATE|INSERT|DROP))"
```

Note: value in the expression is a place holder for the variable found between the green highlighted brackets in Expression 3&4.

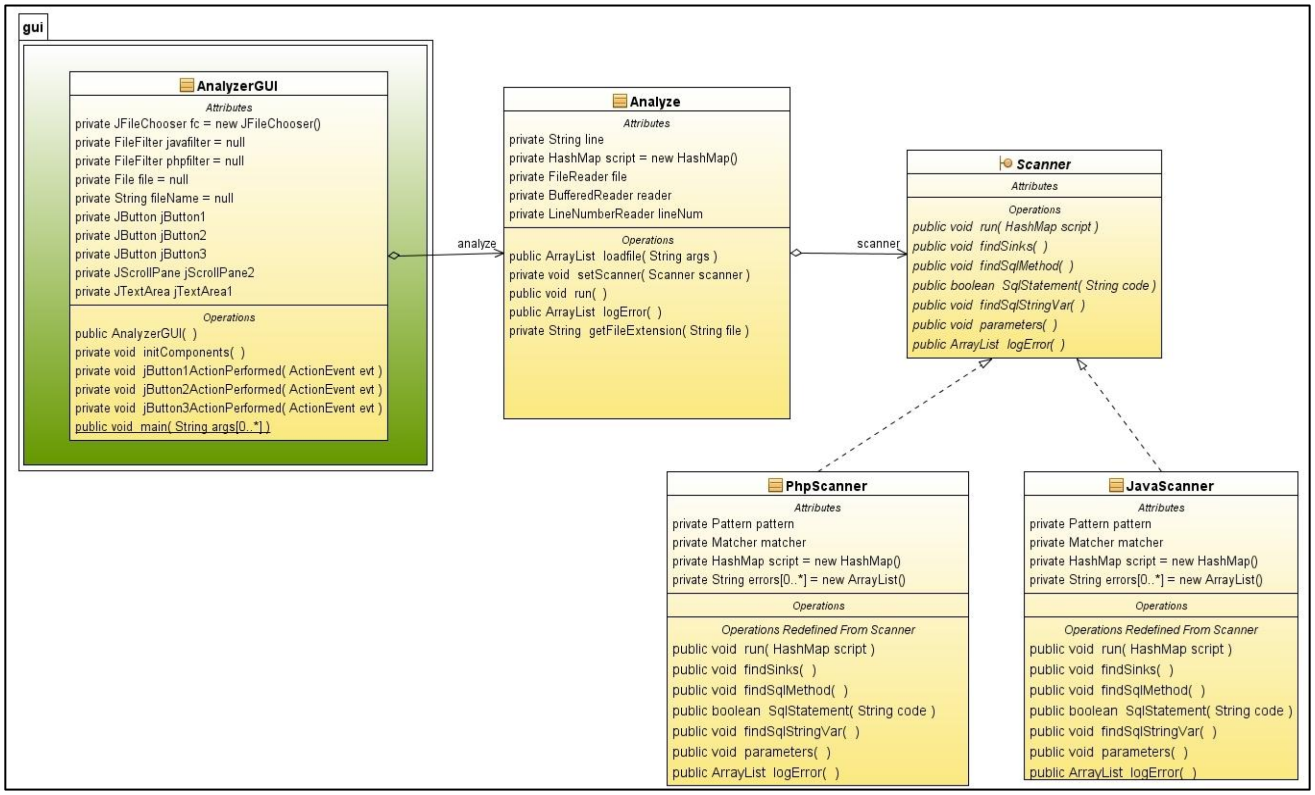
If the analyzer finds a line of code that matches Expression 8 and 9 then it scans that line for HTTP request methods and uses the Expression 6 and 9 accordingly.

If the analyzer finds HTTP request methods in the line then it will record that line and declare it as vulnerable to SQL injection.

There might be cases where the SQL statement will contain variables as parameters instead of HTTP request methods or along with HTTP request methods. So the analyzer will check if any of the tainted variables found earlier are being used in the SQL statement.

4.3 Static Analyzer Class Diagram

Class Diagram 1 Static Analyzer


gui
AnalyzerGUI
Attributes
private JFileChooser fc = new JFileChooser()
private FileFilter javafilter = null
private FileFilter phpfilter = null
private File file = null
private String fileName = null
private JButton jButton1
private JButton jButton2
private JButton jButton3
private JScrollPane jScrollPane2
private JTextArea jTextArea1
Operations
public AnalyzerGUI( )
private void initComponents( )
private void jButton1ActionPerformed(ActionEvent evt)
private void jButton2ActionPerformed(ActionEvent evt)
private void jButton3ActionPerformed(ActionEvent evt)
public void main(String args[0..*])
analyze
Analyze
Attributes
private String line
private HashMap script = new HashMap()
private FileReader file
private BufferedReader reader
private LineNumberReader lineNum
Operations
public ArrayList loadfile(String args)
private void setScanner(Scanner scanner)
public void run( )
public ArrayList logError( )
private String getFileExtension(String file)
scanner
Scanner
Attributes
Operations
public void run(HashMap script)
public void findSinks( )
public void findSqlMethod( )
public boolean SqlStatement(String code)
public void findSqlStringVar( )
public void parameters( )
public ArrayList logError( )
PhpScanner
Attributes
private Pattern pattern
private Matcher matcher
private HashMap script = new HashMap()
private String errors[0..*] = new ArrayList()
Operations
Operations Redefined From Scanner
public void run(HashMap script)
public void findSinks( )
public void findSqlMethod( )
public boolean SqlStatement(String code)
public void findSqlStringVar( )
public void parameters( )
public ArrayList logError( )
JavaScanner
Attributes
private Pattern pattern
private Matcher matcher
private HashMap script = new HashMap()
private String errors[0..*] = new ArrayList()
Operations
Operations Redefined From Scanner
public void run(HashMap script)
public void findSinks( )
public void findSqlMethod( )
public boolean SqlStatement(String code)
public void findSqlStringVar( )
public void parameters( )
public ArrayList logError( )

4.4 Class Diagram Explained

CRC's are mostly used in the initial design stages but I found it extremely useful even when I was programming the software. It gave me a clear view about each class's responsibilities. As I have used an iterative approach to design and development, my software CRC's where extremely handy in sticking to the most basic rule of object oriented programming which is low coupling and high cohesion

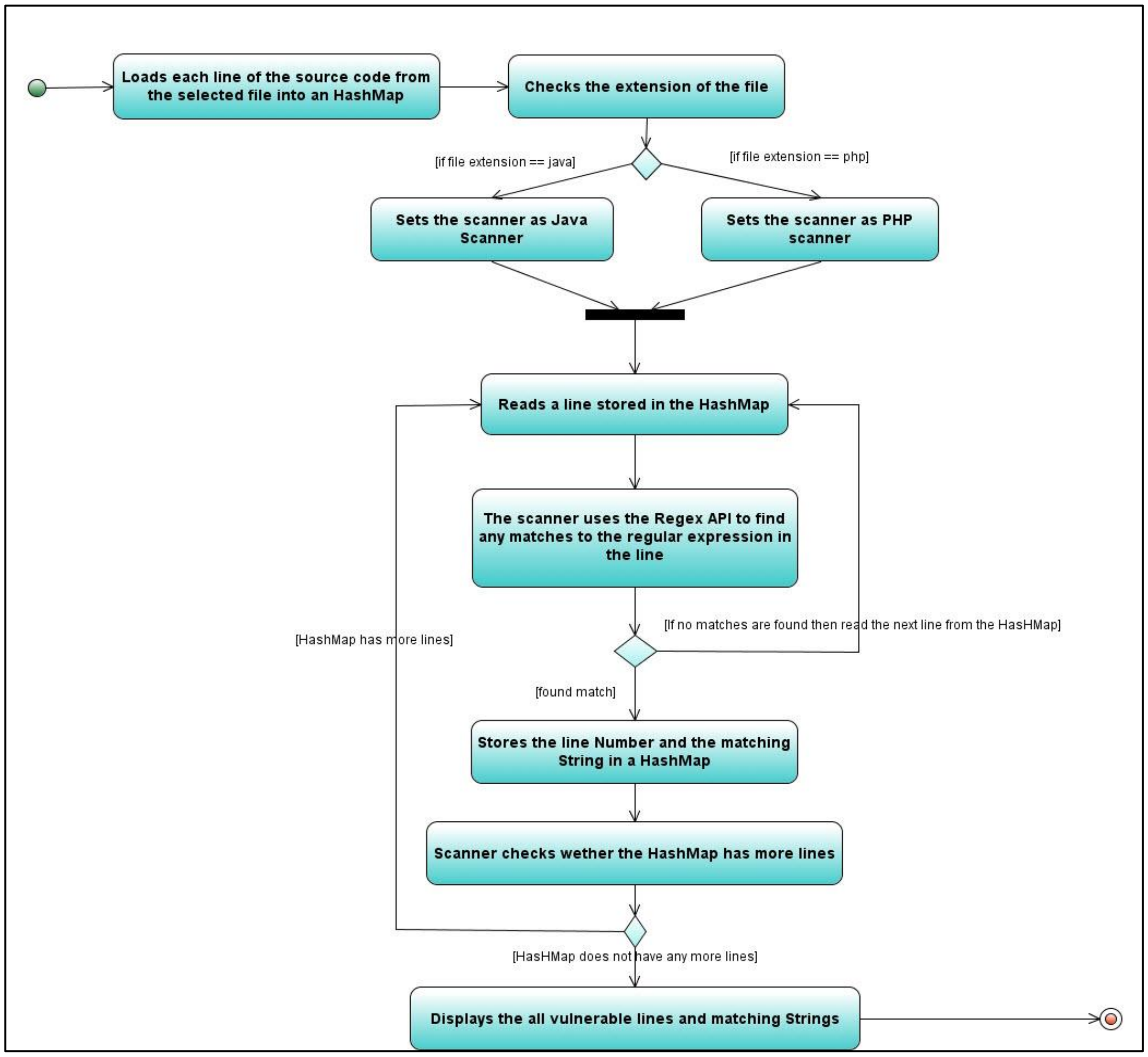
Below is the CRC for each class that is used for Static Analysis

Class	Responsibility	Collaborators
User	• Uses the system path of a source file to find it and load each line into a HashMap • Checks the extension of the file to determine the type of source code inside the file. • Uses the Scanner class to load the PHP scanner if it's a PHP source file • Runs the JAVA scanner if it's a Java source file • Gets the analysis results from the scanner class and pass the output to the class in the GUI package	Scanner
Scanner	Earlier in the in **Section 4.2** I explained that the regular expression to scan PHP and Java is mostly different. The Analyze class uses the Scanner interface to load the appropriate scanner depending on the type of source code to be analyzed.	PhpScanner JavaScanner
PhpScanner	• This class is used to scan PHP source code for SQL vulnerabilities • It implements all the Regular Expression that was discussed in Section 4.2. • Once a vulnerability is found it saves the line number and vulnerability inside an ArrayList. • Provides a get method which can be used by other classes to retrieve the ArrayList that stores the vulnerabilities	Implements the Scanner Interface
JavaScanner	This class is exactly similar to the PHP scanner, the only difference is, it uses different Regular expressions to find SQL vulnerabilities in JAVA source code	Implements the Scanner Interface

4.5 Analyzer Activity Diagram

The Activity diagram below gives a clearly picture of analyzer operations once the user of the software selects a file

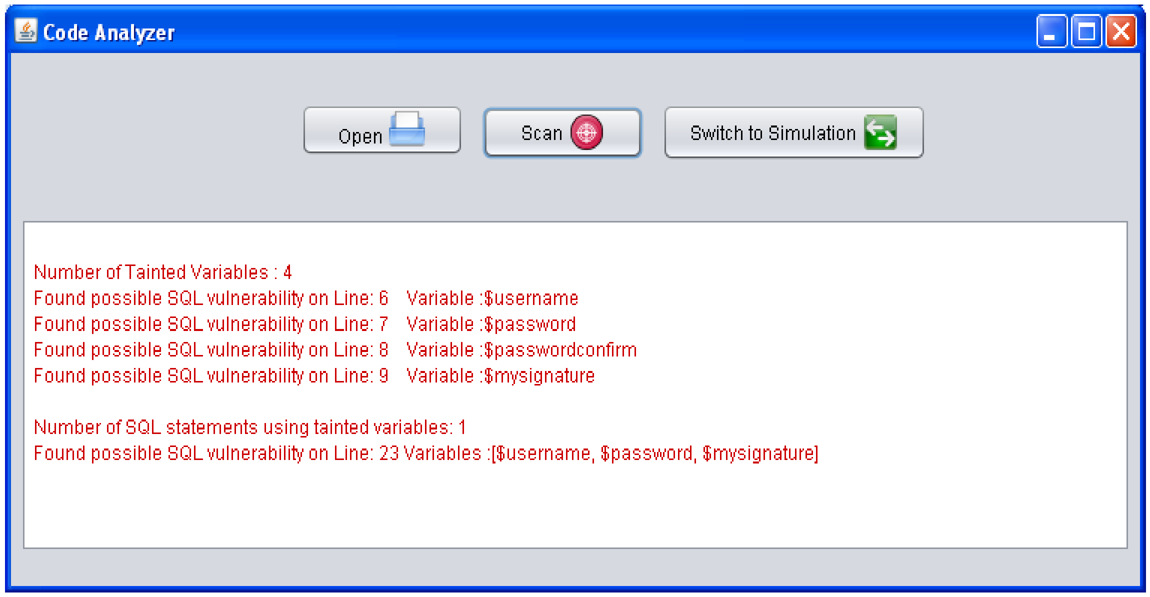
Activity Diagram 1 Scanner



Loads each line of the source code from the selected file into an HashMap
Checks the extension of the file
[if file extension == java]
[if file extension == php]
Sets the scanner as Java Scanner
Sets the scanner as PHP scanner
Reads a line stored in the HashMap
The scanner uses the Regex API to find any matches to the regular expression in the line
[If no matches are found then read the next line from the HasHMap]
[HashMap has more lines]
[found match]
Stores the line Number and the matching String in a HashMap
Scanner checks wether the HashMap has more lines
[HasHMap does not have any more lines]
Displays the all vulnerable lines and matching Strings

4.6 Implementation

Static Analysis GUI



```
Number of Tainted Variables : 4
Found possible SQL vulnerability on Line: 6    Variable :$username
Found possible SQL vulnerability on Line: 7    Variable :$password
Found possible SQL vulnerability on Line: 8    Variable :$passwordconfirm
Found possible SQL vulnerability on Line: 9    Variable :$mysignature

Number of SQL statements using tainted variables: 1
Found possible SQL vulnerability on Line: 23 Variables :[$username, $password, $mysignature]
```

As you can see the Static Analyzer is extremely easy to use, the user selects a file from his/her system by clicking on the **Open** button and then clicks on **Scan** to start the Analysis.

After analysis the system displays the SQL vulnerabilities found in the source code with the line number.

The user interface has been built using the Netbeans GUI Builder so most the code was generated by Netbeans.

4.7 Testing

4.7.1 Testing Functional Requirements

Requirements	Result	View in Appendix A
Analyze PHP source code	Pass	Screen Shot 1
Analyze Java source code	Pass	Screen Shot 3
Allow the user to select the source code file from the system where this software is being deployed	Pass	Screen Shot 2
Display (Output) number of vulnerable user inputs detected in the source code	Pass	Screen Shot 5
Display (Output) vulnerable lines of code along with the line number	Pass	Screen Shot 5
Display (Output) number of tainted variables detected in the source code	Pass	Screen Shot 4
Display (Output) each tainted variable and the line number where the sink was detected in the source code	Pass	Screen Shot 4

4.7.2 Testing Core Functional Requirements

The tests conducted in this section has two aims

- To make sure the analyzer is working and detecting vulnerabilities in the code
- Determine the efficiency of the detection algorithm, used in the analyzer by comparing the results of the analyzer with the results of some good static analyzers that have already been mentioned in my research (Section 2.3, page 12).

The source files used for this section of the test are all vulnerable to SQL injection and have been taken from a web application know as **Multidae 2.1.7** and **Juliet Test Suite**. These test subjects are not subjected to any copyright and is legally provided in order to learn about web application security and code analyzers.

4.7.2.1 Detect SQL vulnerabilities in PHP code

Comparing results from my analyzer SQUILD to other static analyzers

PHP Scripts	**SQUILD**		**RIPS**	
	Tainted Variables	**Vulnerable SQL Statement**	**Tainted Variables**	**Vulnerable SQL Statement**
Vuln Script (1)	4	1	4	1
Vuln Script (2)	2	1	2	1
Vuln Script (3)	2	0	2	1
Vuln Script (4)	1	0	1	0
Vuln Script (5)	0	0	0	0
Vuln Script (6)	1	0	1	0
Vuln Script (7)	3	1	3	2
Vuln Script (8)	1	0	1	0
Vuln Script (9)	0	0	0	0
Vuln Script (10)	1	0	1	1
Total	15	3	15	6

Conclusion: The PHP Scanner is able to detect most of the vulnerabilities but fails to detect a few. So I manually checked the files, the problem was the Regular Expression being used by the analyzer does not detect SQL statement written on multiple lines. Unfortunately I could not fix this problem on time.

4.7.2.2 Detect SQL vulnerabilities in JAVA code

Comparing results from my analyzer SQUILD to other static analyzers

JAVA Scripts	SQUILD		LAPSE+	
	Tainted Variables	Vulnerable SQL Statement	Tainted Variables	Vulnerable SQL Statement
JavaSource(1)	0	0	0	5
JavaSource(2)	0	0	0	2
JavaSource (3)	0	0	0	2
JavaSource (4)	1	1	1	2
JavaSource (5)	0	0	0	5
JavaSource (6)	0	0	0	2
JavaSource (7)	1	1	0	1
JavaSource (8)	0	0	0	3
JavaSource (9)	0	0	0	3
JavaSource (10)	0	0	0	4
Total	2	2	1	29

Conclusion: The Java scanner was just able to detect two vulnerabilities which are not good enough. So I checked the source code myself and the I found that the Java code analyzer was displaying the right results. The problem lied with the LASPS+ scanner

The LAPSE+ scanner was flagging lines of code which were using fixed variables as SQL parameters. These types of line are completely legal and are not vulnerable to SQL injections because it cannot be manipulated by an attacker so I don’t think these should be flagged as vulnerable lines.

The analyzer also could not detect SQL statements that were written in small letters. As the Regular Expression I have built is case sensitive it will not detect statements that are not written in the capital letters so I made a small change to the regular expression and that fixed the problem.

In the case of scanning java source code the analyzer had the same problem as it had with scanning PHP code. The analyzer could not detect multiple lines of source.

4.7.3 Testing Non - Functional Requirements

Requirement	Result	Appendix Page Number
The software should be able to run Windows and Linux platforms	The software runs smoothly on Windows and Linux	Screen Shot 7
The GUI should be simple and easy to use	The GUI is extremely simple. Please see the appendix	See Appendix 1

5 Simulator

5.1 Requirements Specification

5.1.1 Functional Requirements

- Allow the user to enter the target URL on which the attack simulation is supposed to be conducted
- Connect to the web link provided by the user
- Display (Output) the name or ID of the input field where a SQL vulnerability is detected

5.1.2 Core Functional Requirements

- Parse all forms in a webpage and inject it
- Simulator detects all possible SQL vulnerable input points

5.1.3 Non Functional Requirements

- The simulator should be able to run on Linux and Windows
- The GUI should be simple and easy to use

5.2 Simulation Design

The design of the simulator is based on four main steps of simulating SQL injection mentioned briefly in the research section of this report. The design and implementation of these steps have been further discussed below.

Step 1: Identify all points of data entry

Generally most websites contain forms to provide users the ability to send, search and get data from the web server. Based on this theory I have used a HTML Parser (Jsoup) to extract all forms and its input fields from a given web page. Most of the work here is done by the parser because the HTML parser (Jsoup) is so good that once a connection has been made it is as easy as using a single method to extract all the forms from the web page with a single method.

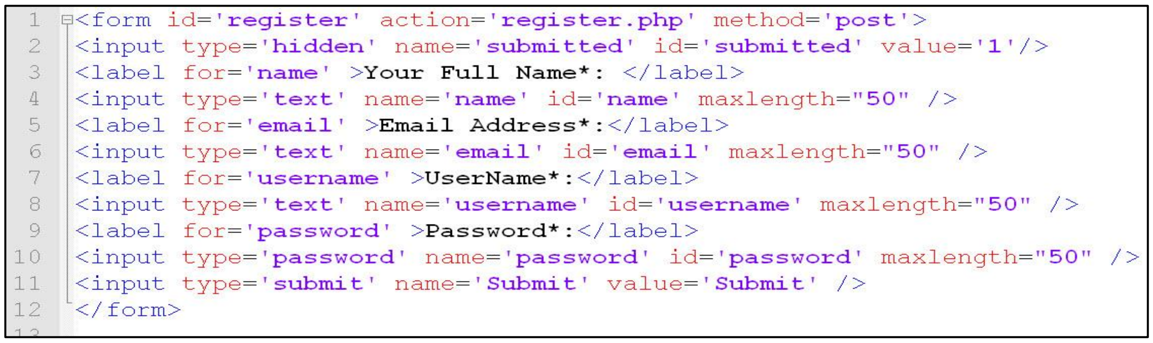
Once the forms have been collected it is necessary to filter through the collection and only use the forms that have an “action” attribute because forms without this attribute will not know where to submit the data from the form.

Sending data to a server from a HTML Form can be done in my ways such as using JavaScript but to keep things simple and easy I have only used the parser to extract Forms which use the general way of sending data to server. These features can be added in the future once the core functionalities have been completed.

Step 2: Prepare form data

In this step the data extracted from the webpage is used to build a form filled with dummy data. To better understand how the data for the form is prepared let's examine the HTML Form shown below

Code Snippet 4 HTML Form

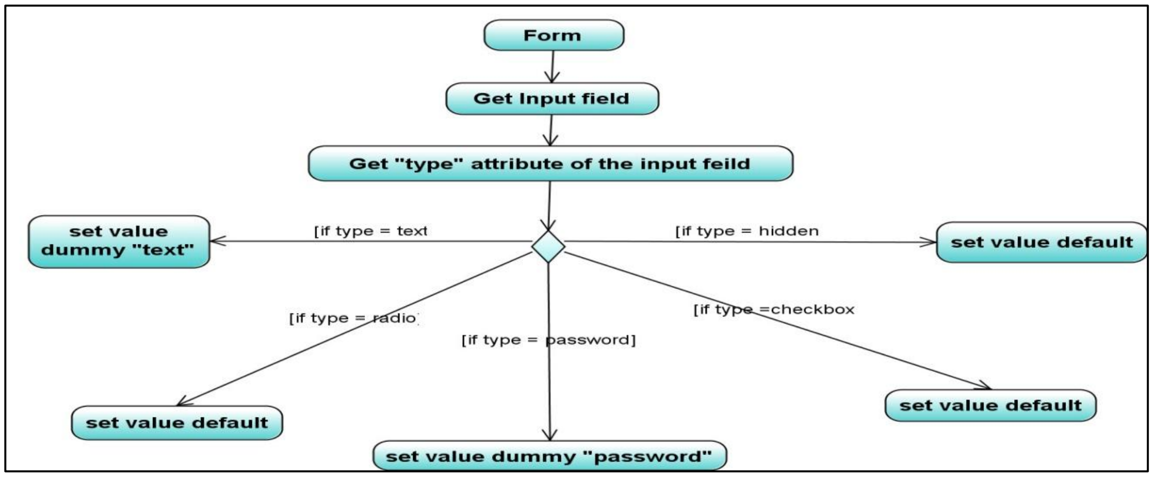


```
<form id='register' action='register.php' method='post'>
<input type='hidden' name='submitted' id='submitted' value='1'/>
<label for='name' >Your Full Name*: </label>
<input type='text' name='name' id='name' maxlength="50" />
<label for='email' >Email Address*:</label>
<input type='text' name='email' id='email' maxlength="50" />
<label for='username' >UserName*:</label>
<input type='text' name='username' id='username' maxlength="50" />
<label for='password' >Password*:</label>
<input type='password' name='password' id='password' maxlength="50" />
<input type='submit' name='Submit' value='Submit' />
</form>
```

All the input fields are extracted with their corresponding attributes (name, id, type). The "type" attribute is used to determine what dummy value is going to be set for the input field.

The activity diagram below shows how the type attribute is used to set a value for the input field.

Activity Diagram 2 Assigning values to input field



The input fields shown in the HTML Form (Code Snippet 4) in the previous goes through the procedure shown above and is assigned the relevant dummy value. The input fields of type; radio, checkbox, hidden are assigned their default values which was set when the HTML form was programmed. In the case of example HTML Form (Code Snippet 4) the hidden input field will be assigned its default value 1.

Step 3: Prepare data for Injection

The aim of an automatic SQL injection is to perform attacks just like a hacker but a hacker will have the skills to craft a SQL injection depending on the web application. Building SQL injections is not an easy task it requires advanced skills, knowledge, patience which the simulator can't mimic. But there are some very common tried and tested SQL injections commands which can be used to detect the possibility of a SQL injection. These SQL injections have been listed in Table 2.

Table 2 List of SQL signatures (Clarke, 2009**)**

Testing String	Variations	Expected Results
'		Error triggering. If successful, the database will return an error
1' or '1'= '1	1') or ('1'='1	Always true condition. If successful, it returns every row in the table
value' or '1'= '2	value') or ('1'= '2	No condition. If successful, it returns the same result as the original value
1' and '1'= '2	1') and ('1'= '2	Always false condition. If successful, it returns no rows from the table
1' or 'ab'='a'+ 'b	1') or ('ab'='a'+ 'b	Microsoft SQL Server concatenation. If successful, it returns the same information as an always true condition
1' or 'ab'='a' 'b	1') or ('ab'='a' 'b	MySQL concatenation. If successful, it returns the same information as an always true condition
1' or 'ab'='a'|| 'b	1') or ('ab'='a'||'	Oracle concatenation. If successful, it returns the same information as an always true condition

The simulator puts each of the above testing strings into one of the input fields of a form. This process is repeated until each of the input field has been inserted at least once, with every single one of the strings shown above.

Step 4: Analyze the Response from the Server

This is an extremely crucial and important step which decides whether the web page is vulnerable to SQL injections. The simulator records the status code and the response from the server for every submission made on the server.

The response from the server depends on how the server has been programmed to handle errors. If the server performs good validation and sanitation on the input it receives from the client then it will immediately detect the anomaly in the input which won't allow a SQL injection, otherwise the SQL injection will cause the expected behavior for which it was injected.

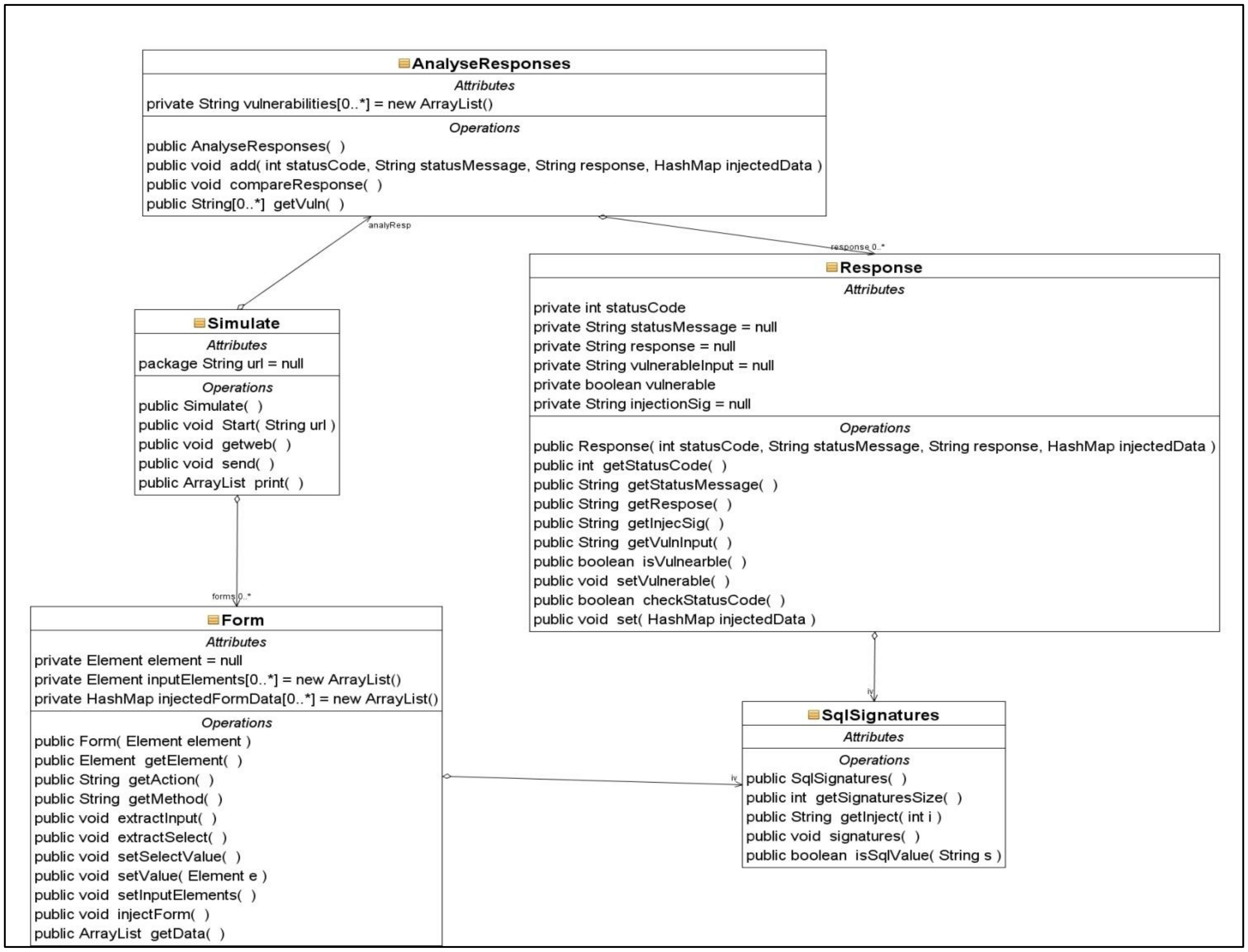
The simulator analyzes the response and looks for the following problems

- If the status code is equal to HTTP 500 error or 302 Redirection then there is a very good possibility that the application is vulnerable to SQL injection because it caused some kind of an error.
- Servers which don't handle errors from the database properly will most probably show the error message in the response body. If the simulator detects such a database error message it will conclude that the web app is vulnerable to SQL injection.

The response analyses is conducted in the order shown above and as soon as one of the above responses is true the simulator will display (output) the Name or the ID of the input field that was used to inject the SQL injection.

5.3 Simulator Class Diagram

Class Diagram 3 Simulator


AnalyseResponses
Attributes
private String vulnerabilities[0..*] = new ArrayList()
Operations
public AnalyseResponses( )
public void add(int statusCode, String statusMessage, String response, HashMap injectedData)
public void compareResponse( )
public String[0..*] getVuln( )
analyResp
response 0..*
Response
Attributes
private int statusCode
private String statusMessage = null
private String response = null
private String vulnerableInput = null
private boolean vulnerable
private String injectionSig = null
Operations
public Response(int statusCode, String statusMessage, String response, HashMap injectedData)
public int getStatusCode( )
public String getStatusMessage( )
public String getRespose( )
public String getInjecSig( )
public String getVulnInput( )
public boolean isVulnearble( )
public void setVulnerable( )
public boolean checkStatusCode( )
public void set(HashMap injectedData)
Simulate
Attributes
package String url = null
Operations
public Simulate( )
public void Start(String url)
public void getweb( )
public void send( )
public ArrayList print( )
forms 0..*
Form
Attributes
private Element element = null
private Element inputElements[0..*] = new ArrayList()
private HashMap injectedFormData[0..*] = new ArrayList()
Operations
public Form(Element element)
public Element getElement( )
public String getAction( )
public String getMethod( )
public void extractInput( )
public void extractSelect( )
public void setSelectValue( )
public void setValue(Element e)
public void setInputElements( )
public void injectForm( )
public ArrayList getData( )
iv
iv
SqlSignatures
Attributes
Operations
public SqlSignatures( )
public int getSignaturesSize( )
public String getInject(int i)
public void signatures( )
public boolean isSqlValue(String s)

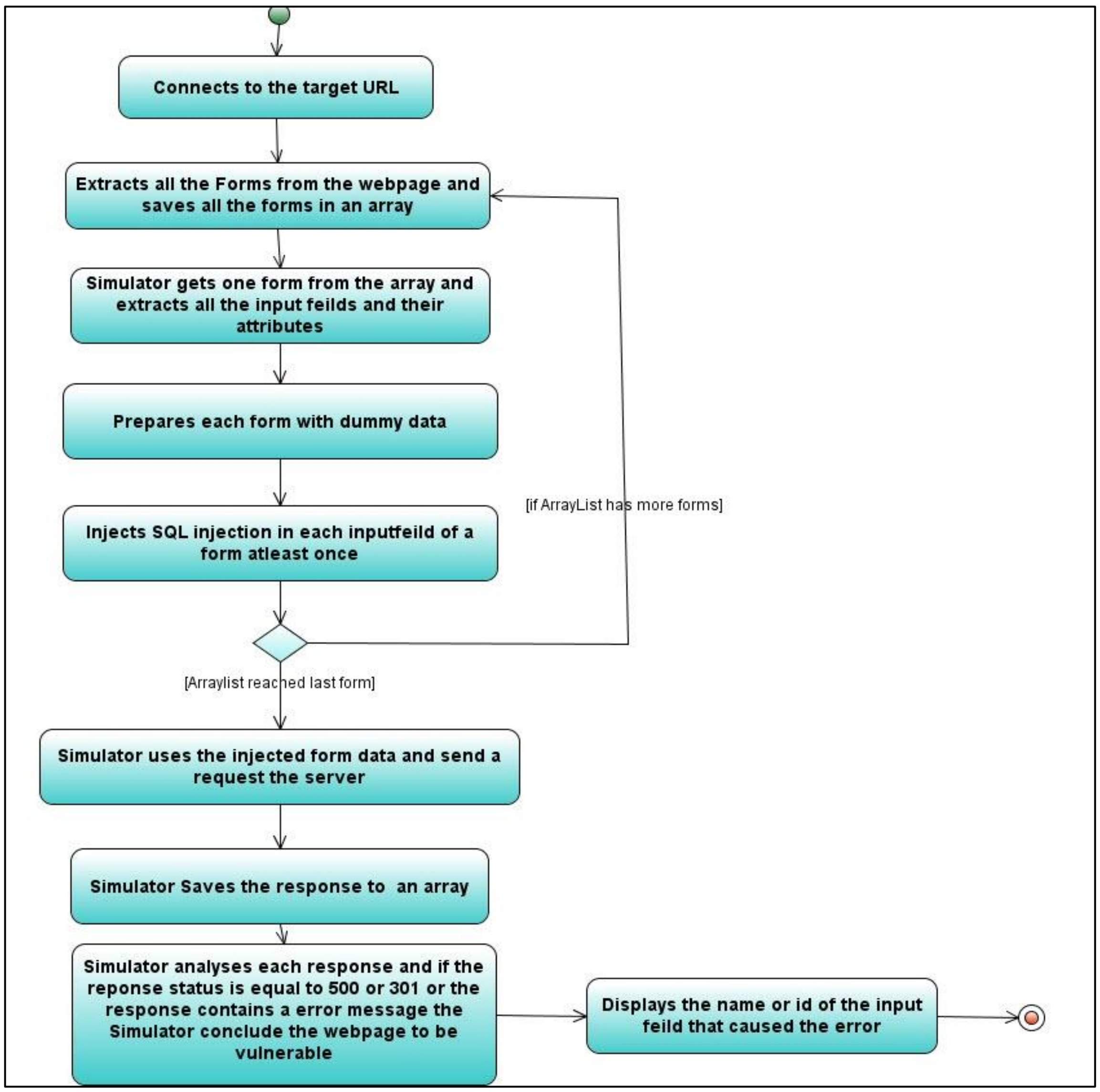
5.4 Class Diagram Explanation

Class	Responsibility	Collaborators
Simulate	• Uses the URL submitted by the user to connect to the webpage • Uses the parser to retrieve all the forms from the webpage • Uses the Form class to retrieve all the injected forms • Submits the injected data on the server • Receives the response from the server and pass the response to the AnalyzeResponse class • Gets the detected data from the AnalyzeResponse class which is then displayed in the GUI	Form AnalyzeResponse
Form	• Prepares the form data with injected signatures	Simulate SqlSignatures
SqlSignatures	• Stores all the SQL signatures and provides getter methods to retrieve a specific signature	Form Response
Response	• This class is used to format the response and can be used by other classes to get the response body or its status code	Response AnalyzeResponse
AnalyzeResponse	• Analyses the response from the server and if a Mysql error is found or the status code is 500, 302 then it flags the input field through which the sql signature was inserted	Simulate Response

5.5 Simulator Activity Diagram

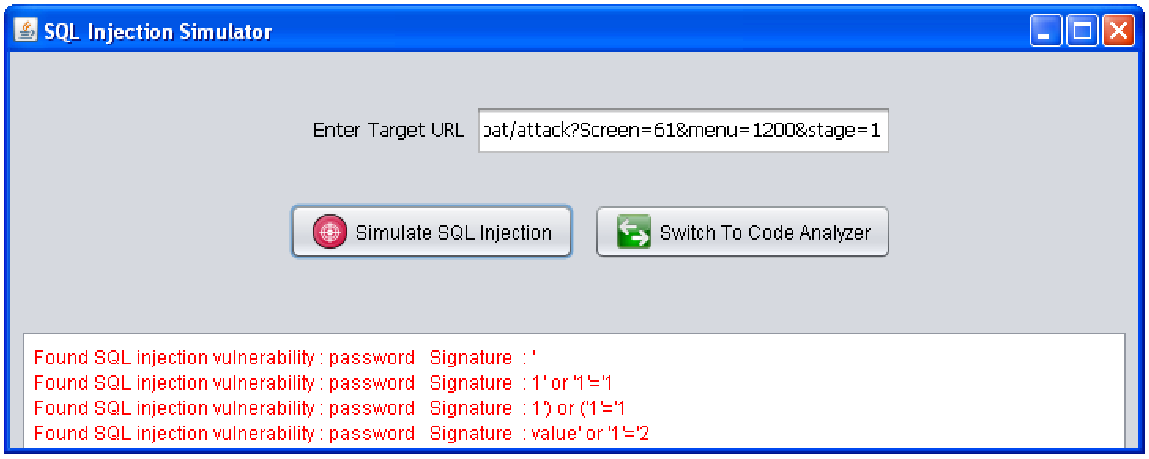
The sequence diagram below explains how the simulator operates once the user provides it with a target URL

Activity Diagram 3 Simulator


Connects to the target URL
Extracts all the Forms from the webpage and saves all the forms in an array
Simulator gets one form from the array and extracts all the input feilds and their attributes
Prepares each form with dummy data
Injects SQL injection in each inputfeild of a form atleast once
[if ArrayList has more forms]
[Arraylist reached last form]
Simulator uses the injected form data and send a request the server
Simulator Saves the response to an array
Simulator analyses each response and if the reponse status is equal to 500 or 301 or the response contains a error message the Simulator conclude the webpage to be vulnerable
Displays the name or id of the input feild that caused the error

6 Implementation

Simulator GUI



```
Enter Target URL  pat/attack?Screen=61&menu=1200&stage=1
```

```
Found SQL injection vulnerability : password   Signature  : '
Found SQL injection vulnerability : password   Signature  : 1' or '1'='1
Found SQL injection vulnerability : password   Signature  : 1') or ('1'='1
Found SQL injection vulnerability : password   Signature  : value' or '1'='2
```

The user enters the target link and clicks the “Simulate SQL Injection” button to initiate the simulation. Once all the simulations are completed the simulator displays all the vulnerable input points detected and the signature used to perform the injection.

7 Test

7.1 Testing Functional Requirements

Requirements	Result	Page Number
• Allow the user to enter the target URL on which the attack simulation is supposed to be conducted	Pass	41
• Display (Output) the name or ID of the input field where a SQL vulnerability is detected	Pass	41

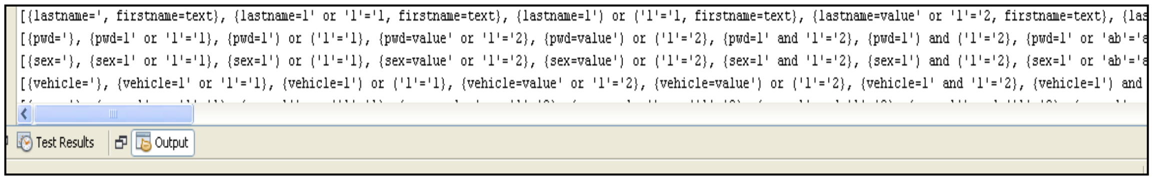
7.2 Testing Core Functional Requirements

The web applications used for this test are **Multidae 2.1.7** and **WebGoat**. These test subjects are not subjected to any copyright and is legally provided in order to learn about web application security.

7.2.1 Parse all forms in a webpage and output Injected Forms

The aim of this test is to verify that the simulator is able to parse a webpage and print out a set of injected Form data that will be used to perform SQL injections on the webpage.

The test was conducted on a webpage and as you can see in the image below the simulator was successful in retrieving all the forms and injecting it with SQL injection strings



```
[{lastname=', firstname=text}, {lastname=1' or '1'='1, firstname=text}, {lastname=1') or ('1'='1, firstname=text}, {lastname=value' or '1'='2, firstname=text}, {las
[{pwd='}, {pwd=1' or '1'='1}, {pwd=1') or ('1'='1}, {pwd=value' or '1'='2}, {pwd=value') or ('1'='2}, {pwd=1' and '1'='2}, {pwd=1') and ('1'='2}, {pwd=1' or 'ab'='a
[{sex='}, {sex=1' or '1'='1}, {sex=1') or ('1'='1}, {sex=value' or '1'='2}, {sex=value') or ('1'='2}, {sex=1' and '1'='2}, {sex=1') and ('1'='2}, {sex=1' or 'ab'='a
[{vehicle='}, {vehicle=1' or '1'='1}, {vehicle=1') or ('1'='1}, {vehicle=value' or '1'='2}, {vehicle=value') or ('1'='2}, {vehicle=1' and '1'='2}, {vehicle=1') and
```

7.2.2 Simulator detects all possible SQL vulnerable input points

This test was conducted on two vulnerable web applications **Multidae** and **WebGoat**.

The results from the test are compared with another popular SQL simulator called sqlMap.

Web Application	**SQUILD**	**sqlMap**
Multidae	• mysql error	• mysql error • extracts data from database
WebGoat	• Login without password	• Login without password • Extracts data from databse

Conclusion: The results from the test show that simulator SQUILD is detecting vulnerabilities but the when the tests were performed with no database error response from the web application the Simulator is not able to detect any errors at all. The sqlMap is able to detect all vulnerabilities and also exploit the web application. The reason sqlMap detects all vulnerabilities is because it can also perform Blind and Timed SQL injections which are used to detect vulnerabilities when the server does not respond with errors.

7.3 Testing Non Functional Requirements

Requirement	**Result**	**Appendix Page Number**
The software should be able to run on Windows and Linux platforms	The software runs smoothly on Windows and Linux	Screen Shot 7
The GUI should be simple and easy to use	The GUI is extremely simple.	Page 41

8 Software Evaluation

The Static Analyzer is detecting most of the SQL injections although I could not fix the analyzer to detect multiple lines but the analyzer still performed pretty well compared to the free tool (RIPS) it was tested against. But unfortunately my tool won't be able to perform type checks and validation checks which is provided by the free tool I tested the SQUILD analyzer against.

Even though the tests on the static analyzer shows that it performed well it will be an extremely tedious and difficult process to extend it in future to a proper commercially used analyzer. Commercial tools perform analysis on thousands of lines so it is extremely important that the analyzer being used is extremely good and accurate unfortunately the analysis technique (String Matching using Regular expressions) that I have used does not understand the semantics of the language to perform accurate analysis.

It was a good experience to learn and use the regular expression which without a doubt will be extremely helpful in future projects but the analyzer that I have built can only probably be used to detect small scripts and with the addition of useful features such as batch processing the analyzer will be more convenient to use.

Unlike the analyzer the simulator does not perform too well in the tests but the tool (sqlMap) that it was compared to has been improved and developed since 2006 which makes it extremely hard to fairly compare it to the simulator that I have built in two months.

Even if the simulator is not detecting all vulnerabilities it does have all the required basic components which will allow anybody to improve and add techniques such as blind SQL injections.

9 Reflection

When I submitted my initial project proposal I proposed to build a software to detect all web application security vulnerabilities, the feedback from my proposal advised me that my initial proposal was too ambitious and should rather concentrate on one specific security problem so as I knew and have dealt with SQL injections previously I decided to build a software to detect SQL injections.

I already knew about SQL injections and have used a few SQL simulators to test a few of my personal projects but I was clearly not aware of the time and skills that’s required to build an automatic detection tool so I think taking the advice from my feedback was a good decision.

The aim of the software was to detect SQL injection which it has successfully achieved but still the software can’t be used to for advanced detection environments hence it still is in its beta stage. Although I think If I had only concentrated on either building a simulator or a static analyzer I would have had more success.

The research done before submitting the proposal was not deep enough to realize the complexities of building an automatic detection tool. Still this has been an extremely good learning curve.

This was the first project in which I have actually managed my time for each part of the project and given a lot of consideration to my design which allowed me to code my software more easily, although this software was designed using an iterative approach my initial designs were extremely helpful.

Through this project I have also learned a lot about application security, defensive programming and software development which will be extremely useful after university.

10 Bibliography

PHP, 2012. [Online]
Available at: http://www.php.net/manual/en/set.mysqlinfo.php
[Accessed February 2012].

OWASP, n.d. [Online]
Available at: https://www.owasp.org/index.php/Main_Page
[Accessed October 2011].

Christoph Kern, A. K. N. D., 2007. *Foundations of Security: What Every Programmer Needs to Know.* s.l.:s.n.

Clarke, J., 2009. *SQL Injection Attacks and Defence.* Burlington: Syngres Publishing.

Cumming, A., 2009. [Online]
Available at: http://sqlzoo.net/
[Accessed Fenruary 2012].

Jsoup, n.d. *Jonathan Hedley.* [Online]
[Accessed February 2012].

Mcgraw, G., 2004. *Static Analysis for Security,* Nenad Jovanovic, C. K. E. K., 2007. *Precise Alias Analysis for Static Detection ofWeb Application Vulnerabilities,*

Orcale, n.d. [Online]
Available at: http://docs.oracle.com/javase/tutorial/essential/regex/
[Accessed February 2012].

Hewltt Packard, 2011. *Mid Year Top Cyber Security Risk Report*

Poel, N. L. d., 2010. *Automated Security Review of PHP Web Applications with*.

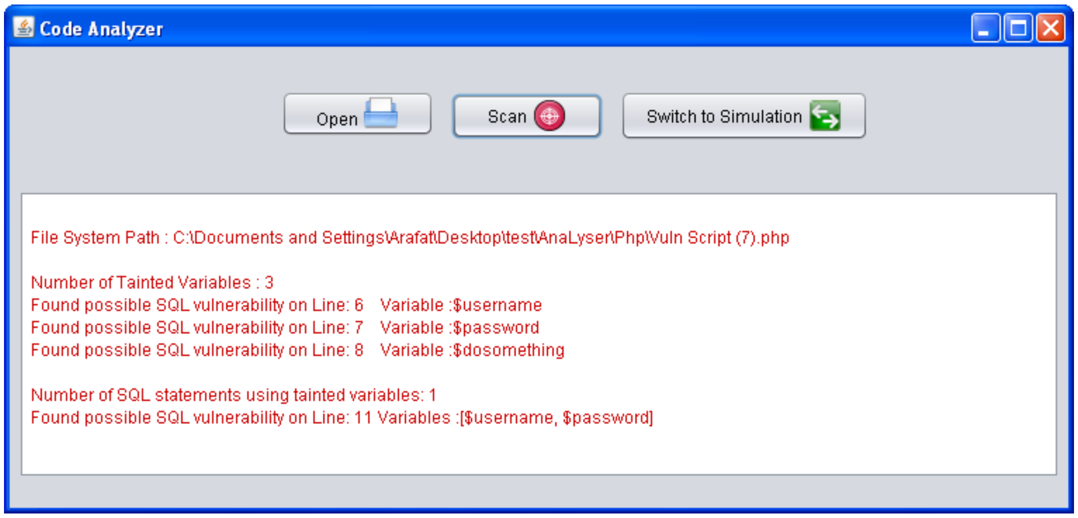
View, S. I. F. A. P. o., 2011. *Miroslav Stampar*

dev.mysql.com., 2011.*guide to php security*.

11 Appendix A

11.1 Static Analyzer Test

11.1.1 Screen Shot 1

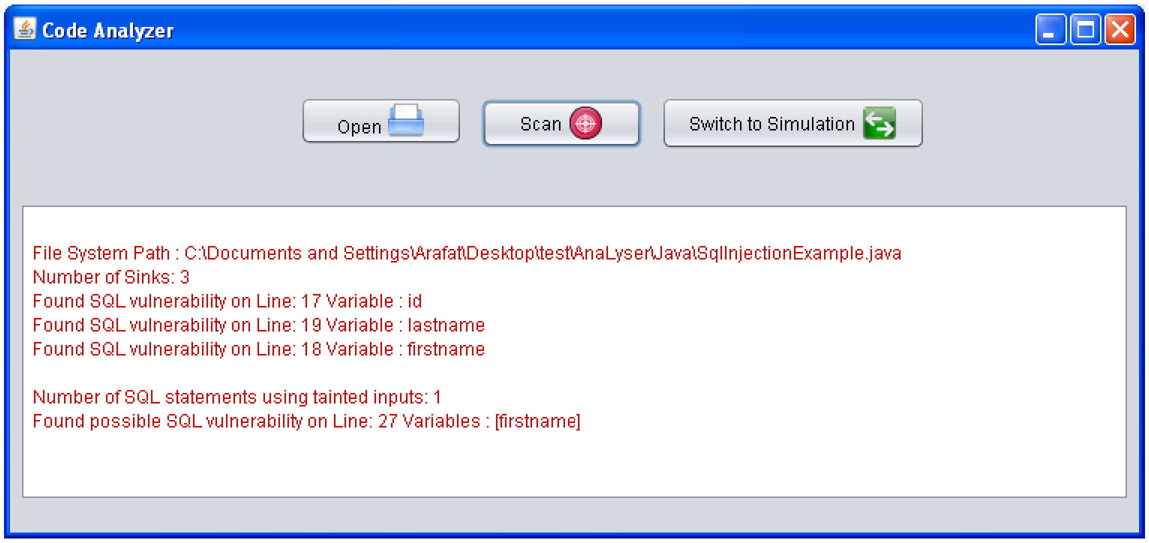


```
File System Path : C:\Documents and Settings\Arafat\Desktop\test\AnaLyser\Php\Vuln Script (7).php

Number of Tainted Variables : 3
Found possible SQL vulnerability on Line: 6   Variable :$username
Found possible SQL vulnerability on Line: 7   Variable :$password
Found possible SQL vulnerability on Line: 8   Variable :$dosomething

Number of SQL statements using tainted variables: 1
Found possible SQL vulnerability on Line: 11 Variables :[$username, $password]
```

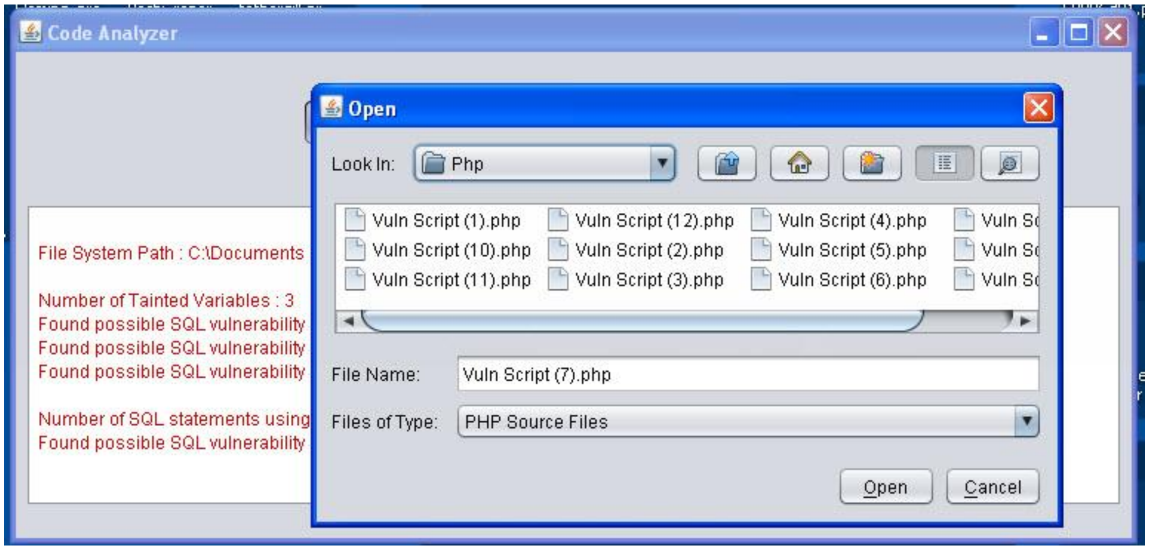
11.1.2 Screen Shot 2



```
File System Path : C:\Documents and Settings\Arafat\Desktop\test\AnaLyser\Java\SqlInjectionExample.java
Number of Sinks: 3
Found SQL vulnerability on Line: 17 Variable : id
Found SQL vulnerability on Line: 19 Variable : lastname
Found SQL vulnerability on Line: 18 Variable : firstname

Number of SQL statements using tainted inputs: 1
Found possible SQL vulnerability on Line: 27 Variables : [firstname]
```

11.1.3 Screen Shot 3

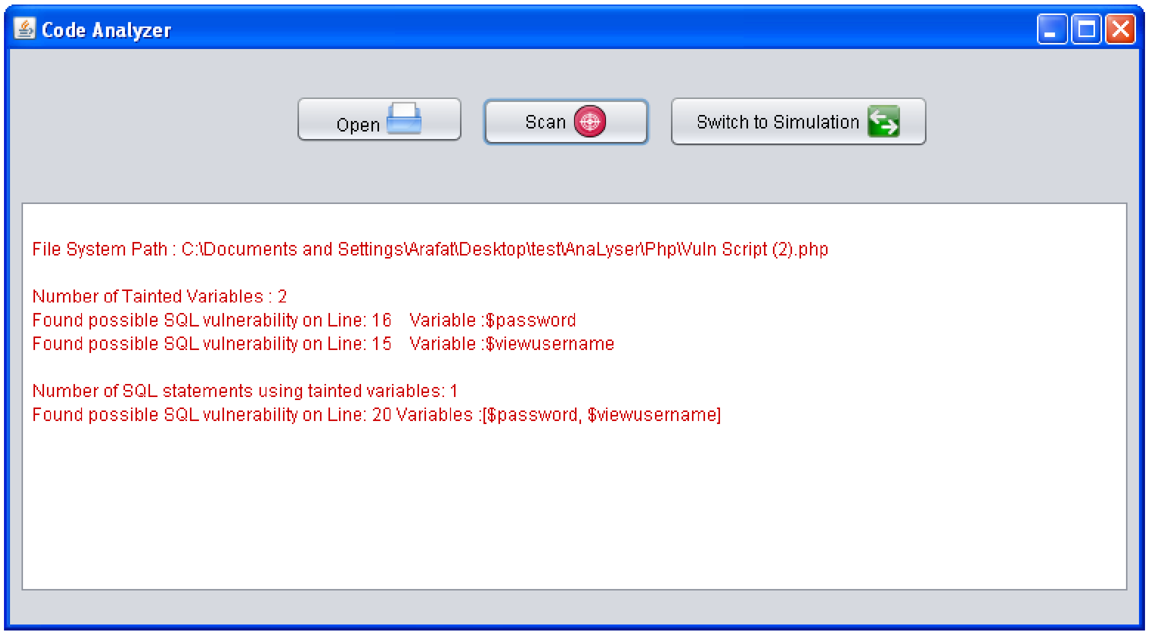


File System Path : C:\Documents

Number of Tainted Variables : 3
Found possible SQL vulnerability
Found possible SQL vulnerability
Found possible SQL vulnerability

Number of SQL statements using
Found possible SQL vulnerability

11.1.4 Screen Shot 4

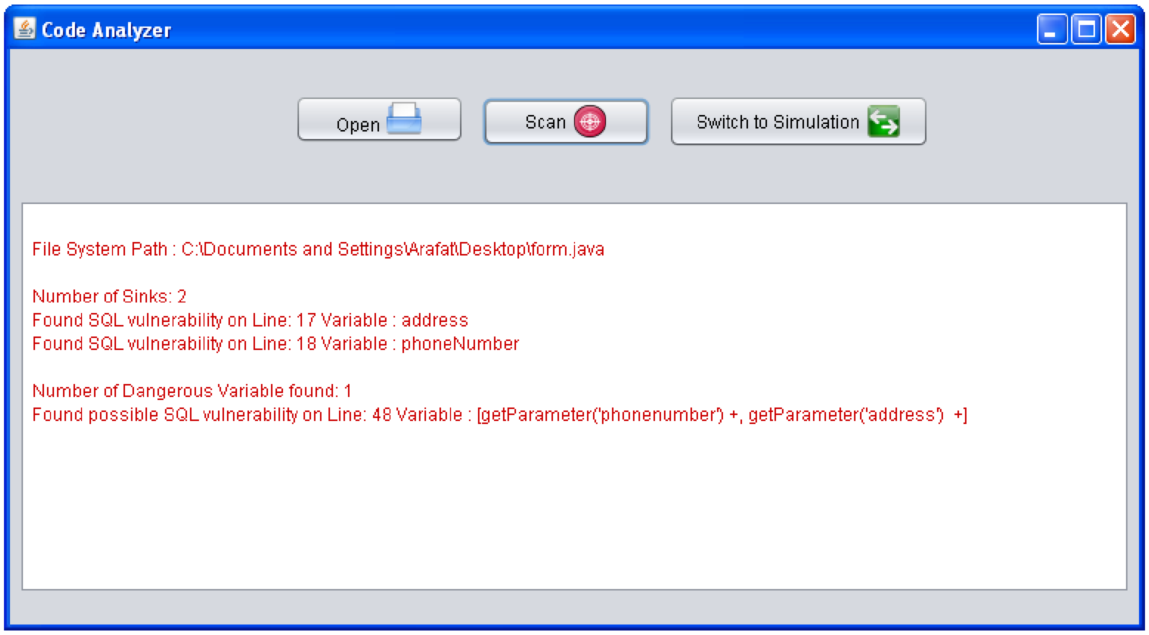


```
File System Path : C:\Documents and Settings\Arafat\Desktop\test\AnaLyser\Php\Vuln Script (2).php

Number of Tainted Variables : 2
Found possible SQL vulnerability on Line: 16    Variable :$password
Found possible SQL vulnerability on Line: 15    Variable :$viewusername

Number of SQL statements using tainted variables: 1
Found possible SQL vulnerability on Line: 20 Variables :[$password, $viewusername]
```

11.1.5 Screen Shot 5

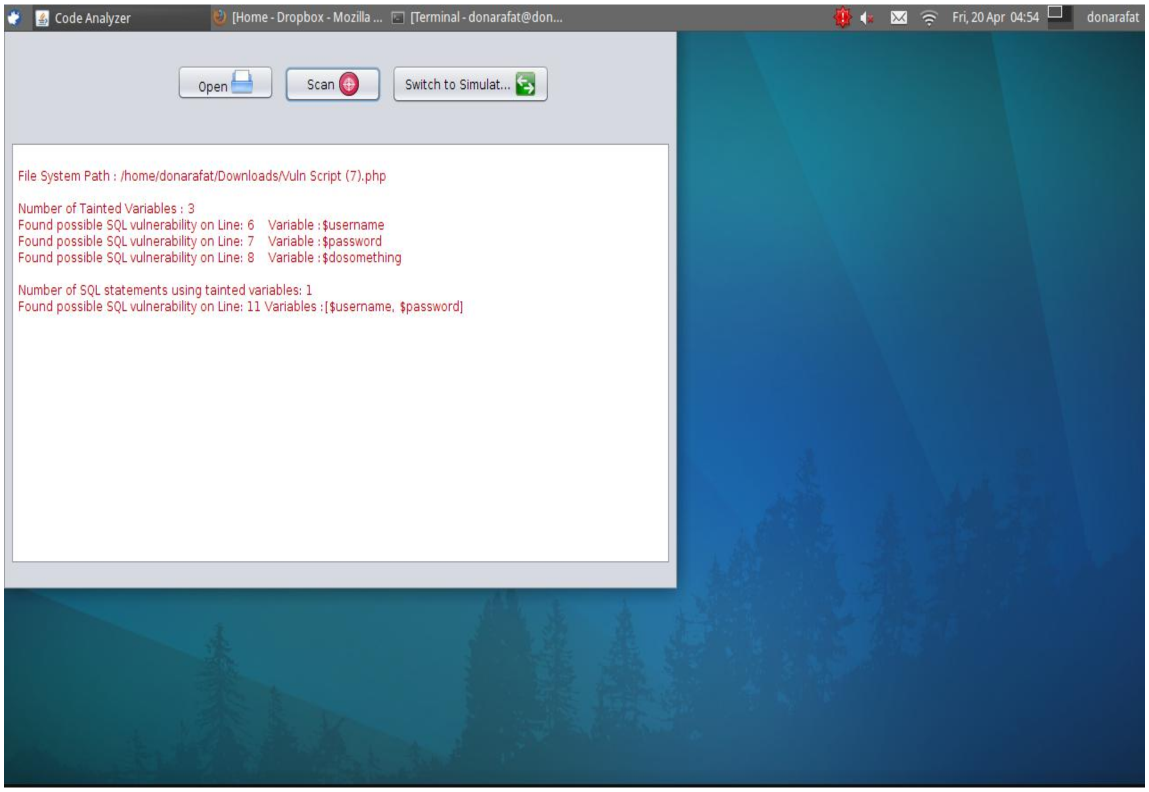


```
File System Path : C:\Documents and Settings\Arafat\Desktop\form.java

Number of Sinks: 2
Found SQL vulnerability on Line: 17 Variable : address
Found SQL vulnerability on Line: 18 Variable : phoneNumber

Number of Dangerous Variable found: 1
Found possible SQL vulnerability on Line: 48 Variable : [getParameter('phonenumber') +, getParameter('address')  +]
```

11.1.6 Screen Shot 7



```
File System Path : /home/donarafat/Downloads/Vuln Script (7).php

Number of Tainted Variables : 3
Found possible SQL vulnerability on Line: 6    Variable :$username
Found possible SQL vulnerability on Line: 7    Variable :$password
Found possible SQL vulnerability on Line: 8    Variable :$dosomething

Number of SQL statements using tainted variables: 1
Found possible SQL vulnerability on Line: 11 Variables :[$username, $password]
```

]

12 Appendix B

12.1 Analyzer Package

12.1.1 Analyze.class

```
package squild.analyzer;

import java.io.*;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.regex.Pattern;
import java.util.regex.Matcher;


public class Analyze {

    private String line;
    private HashMap script = new HashMap();
    private FileReader file;
    private BufferedReader reader;
    private LineNumberReader lineNum;
    private Scanner scanner;


    public ArrayList loadfile(String args)
    {
        try
            {
                FileReader file = file = new FileReader(args);
                lineNum = new LineNumberReader(file);

                while ((line = lineNum.readLine()) != null)
                {
                    script.put(lineNum.getLineNumber(), line);
                }
            }
            catch (IOException x)
            {
                System.err.format("IOException: %s%n", x);
            }

        if(getFileExtension(args).equals("php"))
        {
            setScanner(new PhpScanner());
            run();
            return logError();
        }

        if(getFileExtension(args).equals("java"))
        {
            setScanner(new JavaScanner());
            run();
            return logError();
        }

        return null;

    }


    private void setScanner(Scanner scanner)
    {
        this.scanner = scanner;
    }

    public void run()
    {
        scanner.run(script);
    }

    public ArrayList logError()
    {
        return scanner.logError();
    }


    private String getFileExtension(String file)
    {
        String ext = null;
        int i = file.lastIndexOf('.');

        if (i > 0 &&  i < file.length() - 1) {
            ext = file.substring(i+1).toLowerCase();
        }
        return ext;
    }

}
```

12.1.2 Scanner.class

```
package squild.analyzer;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.regex.PatternSyntaxException;


public interface Scanner {

     public void run(HashMap script);

     public void findSinks();

     public void findSqlMethod();

     public boolean SqlStatement(String code);

     public void findSqlStringVar();

     public void parameters();

     public ArrayList logError();


}
```

12.1.3 JavaScanner.class

```
package squild.analyzer;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Set;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.regex.PatternSyntaxException;

public class JavaScanner implements Scanner
{
    private Pattern pattern;
    private Matcher matcher;
    private HashMap script = new HashMap();
    private HashMap<Integer ,String> sqlStatement = new HashMap();
    private HashMap<Integer, String> sqlString = new HashMap();
    private HashMap<Integer ,ArrayList> dangLine = new HashMap();
    private HashMap<Integer ,String> sink = new HashMap();
    private HashMap<Integer, ArrayList> param = new HashMap();
    private ArrayList<String> errors = new ArrayList();



    public void run(HashMap script)
    {
        this.script = script;

        findSinks();
        findSqlMethod();
        findSqlStringVar();
        parameters();

    }

     public void findSinks()
    {
         for(int i=1; i<script.size()+1;i++)
        {
             ArrayList<String> sinkName = new ArrayList();
             pattern = Pattern.compile("(\\w+) *?=
*?\\w+?\\.(getParameter|getParameterValues|getHeader|getHeaders|getReq
uestedSessionId|getCookies|getValue|getAttribute)");
             String string = script.get(i).toString();
             matcher = pattern.matcher(string);
              while (matcher.find()) {


                 String group1 = matcher.group(1);
                 sink.put(i,group1);

            }
        }

    }


    public void findSqlMethod() throws java.lang.NullPointerException,
PatternSyntaxException
    {
        for(int i=1; i<script.size()+1;i++)
        {
             pattern =
Pattern.compile("(executeQuery\\(|executeUpdate\\(|execute\\()(.*)\\)"
);

             String string = script.get(i).toString();
             matcher = pattern.matcher(string);

             while (matcher.find()) {

                String group2 = matcher.group(2);

                if(SqlStatement(group2))
                {
                    sqlStatement.put(i, group2);
                }
                else
                {
                    sqlString.put(i, group2);
                }

            }
        }
    }



    public boolean SqlStatement(String code)
    {
          pattern =
Pattern.compile("\\\"((SELECT|INSERT|UPDATE|DROP)|(select|insert|updat
e|drop))");
          matcher = pattern.matcher(code);

          if(matcher.find())
            {
                return true;
            }
            else
            {
                return false;
            }
    }



    public void findSqlStringVar()
    {
        for(String value: sqlString.values())
        {


         for(int i=1; i<script.size()+1;i++)
            {
             pattern = Pattern.compile("(" + value + " *?= *?\\(
*?\\\" *?(SELECT|UPDATE|INSERT|DROP))|("+ value +" *?= *?\\\"
*?(SELECT|UPDATE|INSERT|DROP))");
             String string = script.get(i).toString();
             matcher = pattern.matcher(string);

             while (matcher.find()) {

                 sqlStatement.put(i, string);
            }
            }
        }
    }




    public void parameters()
    {
       Set<Integer> lines = sqlStatement.keySet();
         ArrayList<String> dangParam = new ArrayList();
        for(int i:lines)
        {

             ArrayList<String> dangVar = new ArrayList();

             pattern =
Pattern.compile("(getParameter|getParameterValues|getHeader|getHeaders
|getRequestedSessionId|getCookies|getValue|getAttribute)\\s*?\\(.+?\\)
*?\\+");
             matcher = pattern.matcher(script.get(i).toString());

             while (matcher.find()) {

                 dangVar.add(matcher.group());
                 dangLine.put(i, dangVar);
             }




//////////////////////////////////////////////////////////////////

             for(String s: sink.values())
            {
            pattern = Pattern.compile("\\+ *?(\\w+)? *?\\+");
            matcher = pattern.matcher(script.get(i).toString());

             while(matcher.find())
             {
                    String group1 = matcher.group(1);
                    if(group1.equals(s))
                    {
                    dangParam.add(group1);
                    param.put(i, dangParam);
                    }
             }
            }




        }






    }






    public ArrayList logError()
    {

        //printin sql vulnerabilities in user variables

        if(!(sink.isEmpty()))
        {
            errors.add("\n"+"Number of Sinks: " + sink.size() );

            Set<Integer> lines = sink.keySet();
            for(int l: lines)
            {
                errors.add("Found SQL vulnerability on Line: " + l +"
Variable : "+ sink.get(l).toString());
            }

        }


        //printin sql vulnerabilities in user variables

        if(!(dangLine.isEmpty()))
        {
            errors.add("\n"+"Number of Dangerous Variable found: " +
dangLine.size());

            Set<Integer> lines = dangLine.keySet();
            for(int l: lines)
            {
                errors.add("Found possible SQL vulnerability on Line:
" + l +" Variable : "+ dangLine.get(l).toString());
            }
        }


         if(!(param.isEmpty()))
        {
            errors.add("\n"+"Number of SQL statements using tainted
inputs: " + param.size());

            Set<Integer> lines = param.keySet();
            for(Integer l: lines)
            {
                errors.add("Found possible SQL vulnerability on Line:
" + l +" "+ "Variables : "+ param.get(l));
            }

        }


        if(errors.isEmpty())
        {
            errors.add("No SQL Injections found");
        }

        return errors;
    }



}
```

12.1.4 PhpScanner.class

```
package squild.analyzer;

import java.io.*;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Set;
import java.util.regex.Pattern;
import java.util.regex.Matcher;
import java.util.regex.PatternSyntaxException;



public class PhpScanner implements Scanner {

    private Pattern pattern;
    private Matcher matcher;
    private HashMap script = new HashMap();
    private HashMap<Integer ,String> sqlStatement = new HashMap();
    private HashMap<Integer, String> sqlString = new HashMap();
    private HashMap<Integer ,ArrayList> userInputs = new HashMap();
    private HashMap<Integer ,String> sink = new HashMap();
    private HashMap<Integer ,ArrayList> param = new HashMap();
    private ArrayList<String> errors = new ArrayList();

    public void run(HashMap script)
    {
        this.script = script;

        findSinks();
        findSqlMethod();
        findSqlStringVar();
        parameters();


    }

    public void findSinks()
    {



        for(int i=1; i<script.size()+1;i++)
        {

             pattern = Pattern.compile("(\\$\\w+) *?= *?(' *?\\$_|\"
*?\\$_|\\$_)(GET|POST|REQUEST|COOKIE|SESSION)");
             String string = script.get(i).toString();
             matcher = pattern.matcher(string);
              while (matcher.find()) {


                String group1 = matcher.group(1);
                sink.put(i,group1);


            }

        }

    }


    public void findSqlMethod() throws
java.lang.NullPointerException, PatternSyntaxException
    {


        for(int i=1; i<script.size()+1;i++)
        {
             pattern =
Pattern.compile("(mysql|mysql_db)_query\\((.+?)\\)");
             String string = script.get(i).toString();
             matcher = pattern.matcher(string);
              while (matcher.find()) {

                String group2 = matcher.group(2);

                if(SqlStatement(group2))
                {
                    sqlStatement.put(i, group2);
                }
                else
                {
                    sqlString.put(i, group2);
                }
            }
        }
    }


    public boolean SqlStatement(String code)
    {
          pattern =
Pattern.compile("\\\"(SELECT|INSERT|UPDATE|DROP)");
          matcher = pattern.matcher(code);

          if(matcher.find())
            {
                return true;
            }
            else
            {
                return false;
            }
    }


    public void findSqlStringVar()
    {
        for(String value: sqlString.values())
        {


         for(int i=1; i<script.size()+1;i++)
            {
             pattern = Pattern.compile("\\"+value+" *?= *?\"
*?(SELECT|UPDATE|INSERT|DROP)");
             String string = script.get(i).toString();
             matcher = pattern.matcher(string);

             while (matcher.find()) {

                 sqlStatement.put(i, string);
            }
            }
        }
    }




    public void parameters()
    {

        String editString = null;
        Set<Integer> lines = sqlStatement.keySet();

        for(int i:lines)
        {
             ArrayList<String> dangVar = new ArrayList();
             ArrayList<String> dangParam = new ArrayList();
             editString = script.get(i).toString();

             pattern =
Pattern.compile("\\$\\_(POST|GET|REQUEST|SESSION|COOKIE) *?\\( *?'
*?\\$\\w*? *?' *?\\)");
             matcher = pattern.matcher(editString);

             while (matcher.find()) {

                 dangVar.add(matcher.group());
                 matcher.replaceAll("");
                 userInputs.put(i, dangVar);

             }




/////////////////////////////////////////////////////////////////////
///////////////////

             for(String s: sink.values())
             {
                pattern = Pattern.compile("\\"+s);
                matcher = pattern.matcher(editString);

             while(matcher.find())
             {
                    dangParam.add(matcher.group());
                    param.put(i, dangParam);
             }

             }

        }
    }




    public ArrayList logError()
    {
        //printin sql vulnerabilities in user variables

        if(!(sink.isEmpty()))
        {
            errors.add("\n"+"Number of Tainted Variables :
"+sink.size());

            Set<Integer> lines = sink.keySet();
            for(int l: lines)
            {
                errors.add("Found possible SQL vulnerability on Line:
" + l +"    Variable :"+ sink.get(l));
            }

        }


        //printin sql vulnerabilities in user variables

        if(!(userInputs.isEmpty()))
        {
            errors.add("\n"+"Number of Dangerous user input found:
"+userInputs.size());

            Set<Integer> lines = userInputs.keySet();
            for(Integer l: lines)
            {
                errors.add("Found possible SQL vulnerability on Line:
" + l +"     User Input :"+ userInputs.get(l));
            }

        }


        //sinks are being used in the SQL statement

        if(!(param.isEmpty()))
        {
            errors.add("\n"+"Number of SQL statements using tainted 
variables: " + param.size());

            Set<Integer> lines = param.keySet();
            for(Integer l: lines)
            {
                errors.add("Found possible SQL vulnerability on Line: 
" + l +" "+ "Variables :"+ param.get(l));
            }

        }




        //If no errors are found then print no sql injections found
        if(errors.isEmpty())
        {
            errors.add("No SQL Injections foound");
        }









        return errors;
    }

}
```

12.2 Simulator Packages

12.2.1 Simulate.class

```
package squild.simulator;


import java.io.IOException;
import java.lang.reflect.Array;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Locale;


import org.jsoup.*;
import org.jsoup.nodes.*;
import org.jsoup.select.Elements;



public class Simulate {

    ArrayList<Form> forms = new ArrayList();

    AnalyseResponses analyResp = new AnalyseResponses();

    String url = null;


    public Simulate() {}

    public void Start(String url) throws IOException
    {
        this.url = url;
        getweb();
        send();
        analyResp.compareResponse();
    }

    public void getweb() throws IOException
    {
        Document doc = Jsoup.connect(url).userAgent("Mozilla").get();

        Elements formlist = doc.getElementsByTag("form");



        for(int i=0; i<formlist.size();i++)
        {
            Element form = formlist.get(i);
            Form innerform = new Form(form);
            forms.add(innerform);
        }

    }

    public void send() throws IOException
    {
     for(int i=0; i<forms.size();i++)
     {

       Form f = forms.get(i);

if((!(f.getAction().equals("")))|(!(f.getAction().equals(null))))
       {

           System.out.println(f.getData().size());
           for(int a = 0; a<f.getData().size(); a++)
           {


                HashMap vulData = (HashMap) f.getData().get(a);

                Connection conn =
Jsoup.connect(f.getAction()).userAgent("Mozilla");
                System.out.println(f.getMethod());
                if(f.getMethod().toLowerCase().equals("post"))
                {


                    conn.post();

                    System.out.println(conn.response().body());
                    analyResp.add(conn.response().statusCode(),
conn.response().statusMessage(), conn.response().body(), vulData);

                }

                if(f.getMethod().toLowerCase().equals("get"))
                {

                    conn.get();

                    System.out.println(conn.response().body());

                    analyResp.add(conn.response().statusCode(),
conn.response().statusMessage(), conn.response().body(), vulData);
                }

            }
       }
     }
    }
    public ArrayList print()
    {
        return analyResp.getVuln();
    }
}
```

12.2.2 Form.class

```
package squild.simulator;

import java.util.*;

import org.jsoup.nodes.Element;
import org.jsoup.select.Elements;


public class Form
    {
        private Element element = null;
        private ArrayList<Element> inputElements = new ArrayList();
        private HashMap<String,ArrayList> selectElements = new
HashMap();
        private HashMap<String,String> urlData = new HashMap();
        private ArrayList<HashMap> injectedFormData = new
ArrayList();
        private SqlSignatures iv = new SqlSignatures();

        public Form(Element element)
        {
            this.element = element;
            extractInput();
            setInputElements();
            injectForm();

        }

        public Element getElement()
        {
            return element;
        }

        public String getAction()
        {
            return element.attr("abs:action");
        }

        public String getMethod()
        {
            return element.attr("method");
        }

        public void extractInput()
        {
            Elements inputs = element.select("input");

            for(Element e: inputs)
            {
                inputElements.add(e);
            }

        }
        public void extractSelect()
        {
            Elements select = element.select("select");
            ArrayList<String> options = new ArrayList();
            String name = null;

            for(Element e:select)
            {
                Elements optionlist = e.select("option");

                for(Element o: optionlist)
                {
                    options.add(o.attr("value"));
                }

                if((e.attr("id").equals(null)) ||
(e.attr("id").equals("")))
                {
                   name = e.attr("name");

                }else
                {
                   name = e.attr("id");
                }

                selectElements.put(name, options);
            }
        }

        public void setSelectValue()
        {
            if(!(selectElements.isEmpty()))
            {
                Set<String> keys = selectElements.keySet();
                for(String key:keys)
                {
                  if(!(selectElements.get(key).isEmpty()))
                  {
                      String val =
selectElements.get(key).get(0).toString();
                      urlData.put(key, val);
                  }
                }
            }
        }



        public void setValue(Element e)
        {
            SqlSignatures iv = new SqlSignatures();
            String type = e.attr("type");
            String name = null;
            if((e.attr("id").equals(null)) ||
(e.attr("id").equals("")))
            {
                name = e.attr("name");
            }else
            {
                name = e.attr("id");
            }

            if(type.toLowerCase().equals("text"))
            {
                urlData.put(name, "text");
            }

            if(type.equals("password"))
            {
                urlData.put(name,"password");
            }

            if(type.equals("checkbox"))
            {
                urlData.put(name,e.val());
            }

            if(type.equals("hidden"))
            {
                urlData.put(name,e.val());
            }

            if(type.equals("radio"))
            {
                urlData.put(name,e.val());
            }
        }

        public void setInputElements()
        {
            for(Element e: inputElements)
            {
                setValue(e);
            }
        }



        public void injectForm()
        {
            HashMap copy = new HashMap();
            copy.putAll(urlData);
            Set<String> keys = urlData.keySet();




                for(String s: keys)
                {
                    for(int i = 0; i<iv.getSignaturesSize(); i++)
                    {
                        copy.put(s, iv.getInject(i));

                        extractSelect();
                        setSelectValue();

                        HashMap h = new HashMap();
                        h.putAll(copy);
                        injectedFormData.add(h);


                        copy.clear();
                        copy.putAll(urlData);



                    }


                }
        }

        public ArrayList getData()
        {

            return injectedFormData;

        }

}
```

12.2.3 SqlSignatures.class

```
package squild.simulator;

import java.io.*;
import java.util.ArrayList;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;


public class SqlSignatures{

    private ArrayList<Signature> signatures = new ArrayList();


    public SqlSignatures()
    {
            signatures();
    }

    public int getSignaturesSize()
    {
        return signatures.size();
    }

    public String getInject(int i)
    {
        return signatures.get(i).getSignature();
    }

    public void signatures()
    {
       Signature sign1 = new Signature("inject1", "'", " ");
       Signature sign2 = new Signature("inject2", "1' or '1'='1",
"");
       Signature sign3 = new Signature("inject3", "1') or ('1'='1",
"");
       Signature sign4 = new Signature("inject4", "value' or '1'='2",
"");
       Signature sign5 = new Signature("inject5", "value') or
('1'='2", "");
       Signature sign6 = new Signature("inject6", "1' and '1'='2",
"");
       Signature sign7 = new Signature("inject7", "1') and ('1'='2",
"");
       Signature sign8 = new Signature("inject8", "1' or
'ab'='a'+'b'", "");
       Signature sign9 = new Signature("inject9", "1') or
('ab'='a'+'b'", "");
       Signature sign10 = new Signature("inject10", "1' or 'ab'='a'
'b'", "");
       Signature sign11 = new Signature("inject11", "1') or ('ab'='a'
'b'", "");
       Signature sign12 = new Signature("inject12", "1' or
'ab'='a'||'b'", "");
```

```
        Signature sign13 = new Signature("inject13", "1') or
('ab'='a'||'b", "");

        signatures.add(sign1);
        signatures.add(sign2);
        signatures.add(sign3);
        signatures.add(sign4);
        signatures.add(sign5);
        signatures.add(sign6);
        signatures.add(sign7);
        signatures.add(sign8);
        signatures.add(sign9);
        signatures.add(sign10);
        signatures.add(sign11);
        signatures.add(sign12);
        signatures.add(sign13);

    }


    public boolean isSqlValue(String s)
    {
        for(Signature sig:signatures)
        {

            if(s.trim().equals(sig.getSignature()))
            {
                System.out.println("found me");
                return true;
            }
        }

        return false;
    }


////////////////////////////////////////////////////////////////////
///////
    private class Signature
    {

        private String name;
        private String signature;
        private String variation;

        private Signature(String name, String signature, String
variation)
        {
            this.name = name;
            this.signature = signature;
            this.variation = variation;
        }

        private String getName()
        {
            return name;
        }
        private String getSignature()
        {
            return signature;
        }

        private String getVariation()
        {
            return variation;
        }

    }

//////////////////////////////////////////////////////////////////////
///////
}
```

12.2.4 Response.class

```
package squild.simulator;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Set;

public class Response {

    private int statusCode;
    private String statusMessage = null;
    private String response = null;
    private SqlSignatures iv = new SqlSignatures();
    private String vulnerableInput = null;
    private boolean vulnerable;
    private String injectionSig = null;



    public Response(int statusCode, String statusMessage, String
response, HashMap injectedData)
    {
        this.statusCode = statusCode;
        this.statusMessage = statusMessage;
        this.response = response;
        set(injectedData);


    }


   public int getStatusCode()
   {
       return statusCode;
   }

   public String getStatusMessage()
   {
       return statusMessage;
   }

   public String getRespose()
   {
       return response;
   }

   public String getInjecSig()
   {
       return injectionSig;
   }
```

```
    public String getVulnInput()
    {
        return vulnerableInput;
    }

    public boolean isVulnearble()
    {
        return vulnerable;
    }



     public boolean checkStatusCode()
     {
     if(statusCode == 500)
     {
         return true;
     }

     if(statusCode == 200)
     {
         return false;
     }

     if(statusCode == 301)
     {
         return true;
     }

     return false;
     }

    public void set(HashMap injectedData)
    {

        if(checkStatusCode())
        {
            vulnerable = true;

        }else{
            vulnerable = false;
        }


         Set<String> keys = injectedData.keySet();
         for(String s:keys)
         {
             if(iv.isSqlValue(injectedData.get(s).toString()))
             {
injectionSig = injectedData.get(s).toString();
                 vulnerableInput = s;

             }
         }
    }
}
```

12.2.5 AnalyseResponses.class

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package squild.simulator;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Set;
import java.util.regex.Matcher;
import java.util.regex.Pattern;



public class AnalyseResponses {

    private ArrayList<Response> response = new ArrayList();
    private ArrayList<String> vulnerabilities = new ArrayList();

    public AnalyseResponses()
    {

    }

    public void add(int statusCode, String statusMessage, String 
response, HashMap injectedData)
    {
        Response resp = new Response(statusCode, statusMessage, 
response, injectedData);

        this.response.add(resp);

    }

    public void compareResponse()
    {


        for(Response r:response)
        {
            if(r.isVulnearble())
            {
            String res = r.getRespose();
            String vi = r.getVulnInput();


            for(Response r2:response)
            {
                if(!(res.equals(r2.getRespose())))
                {

                        System.out.println("Found SQL injection 
vulnerability : "+vi);
                        vulnerabilities.add("Found SQL injection
vulnerability user input vi: "+vi+""+r.getInjecSig());




                }

            }
            }
            else
            {
                vulnerabilities.add("Found SQL injection
vulnerability : "+r.getVulnInput()+ "   Signature  : "+
r.getInjecSig());

            }



             Pattern pattern =
Pattern.compile("mysql|MySQL|MYSQL|MySql|Error|Incorrect
database|Can't find record|Error on close|Access denied|Incorrect
table|Cross dependency|Can't DROP|Query was empty|Unknown table");
             Matcher matcher = pattern.matcher(r.getRespose());

             while(matcher.find())
             {
                   vulnerabilities.add("Found SQL injection
vulnerability : "+r.getVulnInput()+ "   Signature  : "+
r.getInjecSig());
             }

        }




    }

    public ArrayList<String> getVuln()
    {
        return vulnerabilities;
    }






}
```