P-3320

K C T
LIBRARY
COIMBATORE- 641 006

WEB APPLICATION SQL INJECTION PREVENTER

A PROJECT REPORT

Submitted by

SANGEETHA K.P.N 71206205040

SHARANYA K.K 71206205043

SUGANTHA PRIYA V 71206205051

in partial fulfillment for the award of the degree

of

BACHELOR OF TECHNOLOGY

IN

INFORMATION TECHNOLOGY

KUMARAGURU COLLEGE OF TECHNOLOGY, COIMBATORE

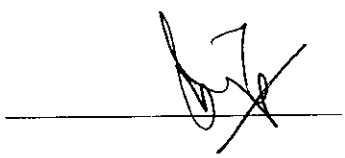
ANNA UNIVERSITY: CHENNAI 600 025

APRIL 2010

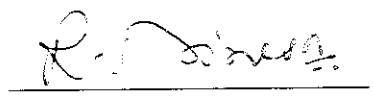
ANNA UNIVERSITY:: CHENNAI 600 025

BONAFIDE CERTIFICATE

Certified that this project report "**WEB APPLICATION SQL INJECTION PREVENTER**" is the bonafide work of **SANGEETHA K.P.N, SHARANYA K.K AND SUGANTHA PRIYA V** who carried out the project work under my supervision



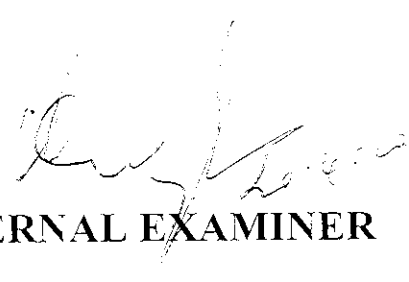
SIGNATURE
Dr.L.S.Jayashree M.E., Ph.D
HEAD OF THE DEPARTMENT
Department of
Information Technology,
Kumaraguru college of Technology,
Coimbatore – 641 006.



SIGNATURE
Mr.R.Dinesh,M.S.(USA)
SUPERVISOR AND GUIDE
Department of
Information Technology,
Kumaraguru college of Technology,
Coimbatore – 641 006.

The candidates with University Register No **71206205040**, **71206205043 & 71206205051 examined** by us in the project viva-voce examination held on 20-4-2010

INTERNAL EXAMINER



EXTERNAL EXAMINER

ACKNOWLEDGEMENT

We express our sincere thanks to our Chairman **Padmabhushaan Arutselvar Dr.N.Mahalingam B.Sc., F.I.E.,** Co-Chairman **Dr.V.K.KrishnaRajaVanavarayar B.E., M.S., M.I.E.,** Correspondent **Shri.M.Balasubramaniam and** joint Correspondent **Dr.A.Selvakumar** for all their support and ray of strengthening hope extended. We are immensely grateful to our Principal, **Prof.Ramachandran,** for his invaluable support to the outcome of this project.

We are deeply obliged to **Dr.S.Thangasamy**, BE (HONS), Ph.D., Dean, Department of Computer Science and Engineering for his valuable guidance and useful suggestions during the course of this project.

We also extend our heartfelt thanks to our project co-coordinator **Dr.L.S.Jayashree** M.E., Ph.D, Asst.Prof, Department of Information Technology for providing us her support which really helped us.

We are indebted to our project guide **Mr.R.Dinesh**, M.S., (USA),Asst.Prof Department of Information Technology for his helpful guidance and valuable support given to us throughout this project.

We thank the teaching and non-teaching staffs of our Department for providing us the technical support during the course of this project. We also thank all of our friends who helped us to complete this project successfully.

TABLE OF CONTENTS

ABSTRACT

Many software systems have evolved to include a Web-based component that makes them available to the public via the Internet and exposes them to a variety of Web-based attacks. One of these attacks is SQL injection, which can give attackers unrestricted access to the databases that underlie Web applications and has become increasingly frequent and serious.

This paper presents an overview of this concept and indicates the need for protecting Web applications against SQL injection that is both conceptual and has practical advantages over most existing techniques. From a conceptual standpoint, the approach is based on the novel idea of positive tainting and on the concept of syntax-aware evaluation. From a practical standpoint, our technique is indicative and true with minimal deployment requirements, and incurs a negligible performance overhead in most cases. We have implemented our techniques in the Web Application SQL-injection Preventer (WASP) tool, which we used to perform an example evaluation . WASP does show four of the successful attempted attacks and tries not to generate any false positives.

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVATIONS

JSP : Java Server Pages

HTML : Hyper Text Markup Language

SQL : Structured Query Language

ODBC : Open Database Connectivity

URL : Uniform Resource Locator

1. INTRODUCTION

1.1 GENERAL

Web applications are the applications that can be accessed over the Internet by using any compliant Web browser that runs on any operating system and architecture. They have become ubiquitous due to the convenience, flexibility, availability, and interoperability that they provide. Web applications interface with databases that contain information such as customer names, preferences, credit card numbers, purchase orders, and so on. Web applications build SQL queries to access these databases based, in part, on user-provided input. The intent is that Web applications will limit the kinds of queries that can be generated to a safe subset of all possible queries, regardless of what input users provide. However, inadequate input validation can enable attackers to gain complete access to such databases. One way in which this happens is that attackers can submit input strings that contain specially encoded database commands. When the Web application builds a query by using these strings and submits the query to its underlying database, the attacker's embedded commands are executed by the database and the attack succeeds.

The results of these attacks are often disastrous and could range from leaking of sensitive data (for example, customer data) to the destruction of database contents. we propose a new highly automated approach for dynamic detection and prevention of SQL injection attack. Intuitively, our approach works by identifying "trusted" strings in an application and allowing only these trusted strings to be used to create the semantically relevant parts of a SQL query such as keywords or operators.

1.2 PROBLEM DEFINITION

Web applications are vulnerable to a variety of security threats. SQL Injection Attacks (SQLIAs) are one of the most significant of such threats. Web applications build SQL queries to access the databases. However, inadequate input validation can enable attackers to gain complete access to such databases. One way in which this happens is that attackers can submit input strings that contain specially encoded database commands. When the Web application builds a query by using these strings and submits the query to its underlying database, the attacker's embedded commands are executed by the database and the attack succeeds. The results of these attacks are often disastrous and can range from leaking of sensitive data to the destruction of database contents.

This approach works by identifying "trusted" strings in an application and allowing only these trusted strings to be used to create the semantically relevant parts of a SQL query such as keywords or operators. Positive tainting identifies and tracks trusted data. In Web applications, sources of trusted data can more easily and accurately be identified than untrusted data sources. Therefore, the use of positive tainting leads to increased automation.

The second approach is the use of flexible syntax-aware evaluation. Syntax-aware evaluation lets us address security problems that are derived from mixing data and code while still allowing for this mixing to occur. it gives developers a mechanism for regulating the usage of string data based not only on its source but also on its syntactical role in a query string.

1.3 OBJECTIVE OF THE PROJECT

The objective of the project is to develop and implement a highly automated technique against SQL injection attacks that is able to detect, stop, and report injection attacks before they cause any vulnerable attack to the database. This is based on the concept of positive tainting and syntax aware evaluation

The positive tainting performs efficient dynamic tainting of java strings that precisely propagates trust markings while strings are manipulated at runtime. Positive tainting identifies and tracks trusted data. The tainted information is tracked at the character level hence while building the SQL queries, strings are constantly broken into substrings, manipulated and combined.

In syntax aware evaluation, to evaluate the query string, the technique uses a SQL parser to break the string into a sequence of tokens that correspond to SQL keywords, operators and literals. The technique then iterates through the tokens hand checks whether tokens other than literals contain only trusted data. If all such tokens pass this check, the query is considered to be safe or specified action can be invoked.

2. LITERATURE REVIEW

2.1 FEASIBILITY STUDY

2.1.1 CURRENT STATUS OF THE PROBLEM

Web applications are also vulnerable to a variety of new security threats. SQL Injection Attacks are one of the most significant of such threats. SQL Injection Attacks has become increasingly frequent and poses very serious security risks because they can give attackers unrestricted access to the databases that underlie Web applications. SQL injection takes advantage of the syntax of SQL to inject commands that can read or modify a database, or compromise the meaning of the original query. SQL Injection Attacks are a class of code injection attacks that take advantage of the lack of validation of user input. These attacks occur when developers combine hard-coded strings with user-provided input to create dynamic queries. Intuitively, if user input is not properly validated, attackers may be able to change the developer's intended SQL command by inserting new SQL keywords or operators through specially crafted input strings. The existing systems have several drawbacks like,

- False negatives
- Incompleteness
- They are expensive

2.1.2 PROPOSED SYSTEM AND ITS ADVANTAGES

The proposed system is based on the POSITIVE TAINTING AND SYNTAX-AWARE EVALUATION. This overcomes the disadvantages of the existing system by performing the following approaches,

1) Identifying trusted data sources and marking data coming from these sources as trusted.

2) Using dynamic tainting to track trusted data at runtime, and

3) Allowing only trusted data to form the semantically relevant parts of queries such as SQL keywords and operators. Unlike previous approaches based on dynamic tainting, our technique is based on positive tainting, which explicitly identifies trusted (rather than untrusted) data in a program. This way, we eliminate the problem of false negatives that may result from the incomplete identification of all untrusted data sources. The system has several advantages such as,

- It is fully-automated
- It has Minimal deployment requirements and
- Low overhead on the application

2.2 SYSTEM REQUIREMENTS

2.2.1 HARDWARE REQUIREMENTS

PROCESSOR	:	Pentium IV
HARD DISK	:	40 GB
RAM	:	256 MB
MONITOR	:	15" Color Monitor
KEYBOARD	:	104 keys Standard Keyboard
MOUSE	:	Standard 3 Button Mouse.

2.2.2 SOFTWARE REQUIREMENTS

OPERATING SYSTEM	:	Windows 9X/NT/XP
LANGUAGE USED	:	J2EE
WEB SERVER	:	Apache Tomcat 6.0
BACK END	:	SQL Server 2000

2.3 SOFTWARE OVERVIEW

2.3.1 JSP

JSP may be viewed as a high-level abstraction of java servlets. JSP pages are loaded in the server and operated from a structured special installed Java server packet called a Java EE Web Application, often packaged as a .war or .ear file archive.

JSP allows Java code and certain pre-defined actions to be interleaved with static web markup content, with the resulting page being compiled and executed on the server to deliver an HTML or XML document. The compiled pages and any dependent Java libraries use Java byte code rather than a native software format, and must therefore be executed within a java virtual machine that integrates with the host operating system to provide an abstract platform-neutral environment.

JSP syntax is a fluid mix of two basic content forms: scriptlet elements and markup. Markup is typically standard HTML or XML, while scriptlet elements are delimited blocks of Java code which may be intermixed with the markup. When the page is requested the Java code is executed and its output is added, in situ, with the surrounding markup to create the final page. Because Java is a compiled language, not a scripting language, JSP pages must be compiled to Java bytecode classes before they can be executed, but such compilation generally only occurs once each time a change to the source JSP file occurs.

Java code is not required to be complete (self contained) within its scriptlet element block, but can straddle markup content providing the page as a whole is syntactically correct (for example, any Java if/for/while blocks opened in one scriptlet element must be correctly closed in a later element for the page to successfully compile). This system of split inline coding sections is called step over scripting because it can wrap around the static markup by stepping over it.

Markup which falls inside a split block of code is subject to that code, so markup inside an if block will only appear in the output when the if condition evaluates to true; likewise markup inside a loop construct may appear multiple times in the output depending upon how many times the loop body runs.

The JSP syntax adds additional XML-like tags, called JSP actions, to invoke built-in functionality. Additionally, the technology allows for the creation of JSP tag libraries that act as extensions to the standard HTML or XML tags.

JSP not only enjoys cross-platform and cross-Web-server support, but effectively melds the power of server-side Java technology with the WYSIWYG features of static HTML pages. JSP pages typically comprise of:

- Static HTML/XML components.
- Special JSP tags
- Optionally, snippets of code written in the Java programming language called "script lets".

It can be used not only by developers, but also by page designers, who can now play a more direct role in the development life cycle. Another advantage of JSP is the inherent separation of presentation from content facilitated by the technology, due its reliance upon reusable component technologies like the JavaBeans TM component architecture and Enterprise JavaBeans TM technology.

2.3.2 SQL SERVER 2000

Microsoft SQL Server 2000 is a relational database management system, it is a complete database and analysis product that meets the scalability and reliability requirements of the most demanding enterprises. It is appropriate for a broad range of solution types, including e-commerce, data warehousing, and line-of-business applications. SQL Server 2000 maintains maximum application compatibility across platforms. SQL is a scripting language that allows you to access the information in the tables.

Unlike database management systems of the past, SQL Server is relatively easy to deploy and use, even for smaller organizations. It is not necessary to run multiple copies of the SQL server database engine to allow multiple users to access the databases on a server. An instance of the SQL Server Standard or Enterprise Edition is capable of handling thousands of users working in multiple databases at the same time. Each instance of SQL server makes all databases in the instance available to all users that connect to the instance, subject to the defined security permissions.

The database engine has two main parts storage engine and relational engine.

SQL Server Security

- Login Authentication.
  - Windows NT Authentication
  - SQL Server Authentication

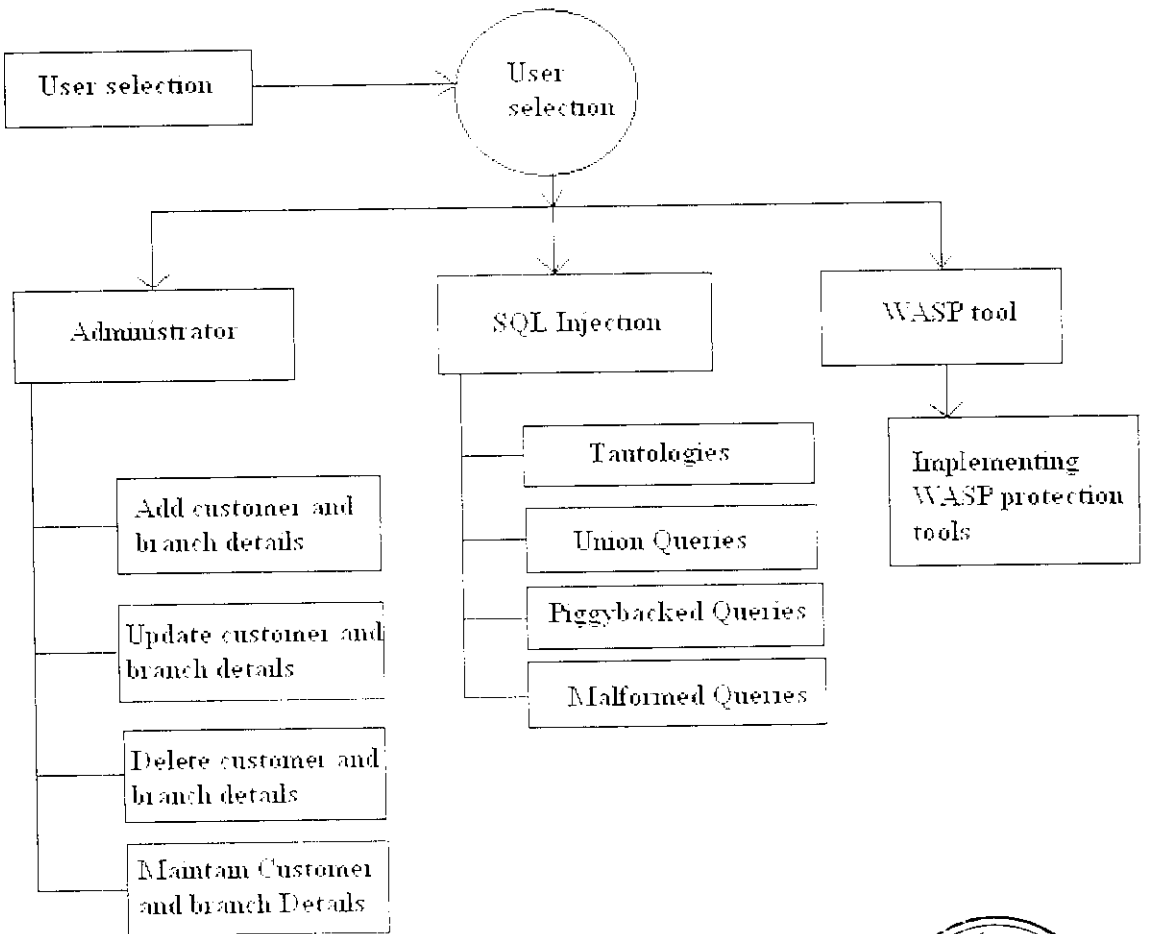
- Permissions validation on user database.
  - T-SQL statements sent to SQL Server.
  - SQL server checks user permissions on receipt of T-SQL statements

FEATURES

- Created by Microsoft and Sybase in the 80s.
- Is SQL Compliant - Uses ANSI SQL
- Supports SQL – 92 standards - Uses T-SQL
- Stores data in a central location and delivers it to clients on request
- New Server Architecture
- Graphic Administration Tools
- Maintains ANSI standards and 6.x Compatibility
- Data integrity means reliability and accuracy of data.
- Integrity rules keep data consistent.
- Supports Client/Server model.

3. DETAILS OF METHODOLOGY EMPLOYED

3.1 PROCESS FLOW CHART DIAGRAM:



LIBRARY
COIMBATORE- 641 006
F - 3325

3.2 MODULES

The following are the modules in the proposed system:

- Administrator
  - Customer Details Maintenance
  - Branch Details Maintenance
- SQL Injection
  - Tautologies
  - Union Queries
  - Piggybacked Queries
  - Malformed Queries
- WASP

3.2.1 ADMINISTRATOR

Admin process in mainly concentrated to maintain the customer details and the branch. The administrator has following functionality to perform and verify.

- ❖ Add Customer and Branch Details
- ❖ Update Customer and Branch Details
- ❖ Delete Customer and Branch Details
- ❖ Maintain Details

Add Customer Details and branch Details

The customer details contain the name, address, contact number, accounts details and the pin number. The branch details contain branch name, branch code, pin and area code. By using these details the user can access the account information and they can use in different application like the online purchasing and in various application.

Update & Delete Customer Details and Branch Details

Update customer details are necessary to the administrator to modify the customer details when they are in needed portion. Customer id is consider as the unique id by using that only the details of the customer is fetched and modified when they needed. Admin have the authorization power to delete the customer details.

Maintain Details

The administrator will maintain the customer details and branch details along with the branch details and they can view the information.

3.2.2 SQL INJECTION ATTACKS

SQL injection attack is a class of code injection attacks that take advantage of the lack of validation of user input.

Tautologies

The general goal of a tautology based attack is to inject SQL tokens that cause the query's conditional statement to always evaluate to true.In this type of injection, an attacker exploits a vulnerable input field that is used in the query's WHERE conditional.

Eg:

```
SELECT Branchcode FROM BranchDetails WHERE customerid='c01' OR
1=1 - -' pin=
```

Union Queries

Attackers inject a statement of the form "UNION < injected query >." By suitably defining < injected query >, attackers can retrieve information from a specified table. The outcome of this attack is that the database returns a data set that is the union of the results of the original query with the results of the injected query.

Eg:

```
SELECT cid FROM CustomerDetails WHERE login='' UNION
SELECT pin from BranchDetails where accountno=123 – AND pin=
```

Malformed Queries

Union queries and piggybacked queries let attackers perform specific queries or execute specific commands on a database, but require some prior knowledge of the database schema, which is often unknown. Malformed queries allow for overcoming this problem by taking advantage of overly descriptive error messages that are generated by the database when a malformed query is rejected. When these messages are directly returned to the user of the Web application, instead of being logged for debugging by developers, attackers can make use of the debugging information to identify vulnerable parameters and infer the schema of the underlying database.

Eg:

```
SELECT Branchcode FROM BranchDetails WHERE Customerid='c01'
AND pin=convert(int,select top 1 name from sysobjects where xtype='u')
```

Piggybacked queries

If the attack is successful, the database receives and executes a query string that contains multiple distinct queries. The first query is generally the original legitimate query, whereas subsequent queries are the injected malicious queries. This type of attack can be especially harmful because attackers can use it to inject virtually any type of SQL command.

Eg:

```
SELECT Branchcode FROM BranchDetails WHERE Customerid='c01'
OR 1=1 - -' pin=;DROP table CustomerDetails
```

3.2.3 WASP

This eliminate the problem of false negatives that may result from the incomplete identification of all un trusted data sources. False positives, although possible in some cases, can typically be easily eliminated during testing. Our approach also provides practical advantages over the many existing techniques whose application requires customized and complex runtime environments: It is defined at the application level, requires no modification of the runtime system, and imposes a low execution overhead.

Positive Tainting

Positive tainting is based on the identification, marking and tracking of trusted data, rather than untrusted data. In the case of negative tainting, incompleteness leads to trusting data that should not be trusted and ultimately, to false negatives.

Incompleteness may thus leave the application vulnerable to attacks and can be very difficult to detect, even after attacks actually occur, because they may go completely unnoticed. With positive tainting, incompleteness may lead to false positives, but it would never result in an SQLIA escaping detection. In positive tainting identifying trusted data is often straightforward and always less error prone.

To account for the incompleteness, positive tainting provides developers with a mechanism for specifying sources of external data that should be trusted. The data sources can be of various types such as files, network connections and server variables, uses this information to mark data that comes from these additional sources as trusted. Comparing the query with the already maintained trusted data it is marked as trusted or untrusted. The trusted query is then sent to the database for retrieving the information.

Accurate and efficient Taint Propagation

Taint propagation consists of tracking taint markings associated with the data while the data is used and manipulated at runtime Character-level tainting. We track taint information at the character level rather than at the string level. We do this because, for building SQL queries, strings are constantly broken into substrings, manipulated, and combined. By associating taint information to single characters, our approach can precisely model the effect of these string operations.

Syntax-Aware Evaluation

Syntax-aware evaluation of a query string is performed immediately before the string is sent to the database to be executed. To evaluate the query string, the technique first uses a SQL parser to break the string into a sequence of tokens that correspond to SQL keywords, operators, and literals. The technique then iterates through the tokens and checks whether tokens (that is, substrings) other than literals contain only trusted data. If all such tokens pass this check, the query is considered safe and is allowed to execute. If an attack is detected, a developer specified action can be invoked. This approach can also handle cases where developers use external query fragments to build SQL commands. In these cases, developers would specify which external data sources must be trusted, and our technique would mark and treat data that comes from these sources accordingly.

1) It considers only two kinds of data (trusted and untrusted) and

2) It allows only trusted data to form SQL keywords and operators, is adequate for most Web applications.

For example, it can handle applications where parts of a query are stored in external files or database records that were created by the developers. Nevertheless, to provide greater flexibility and support a wide range of development practices, our technique also allows developers to associate custom trust markings to different data sources. Trust policies are functions that take as input a sequence of SQL tokens and perform some type of check based on the trust markings associated with the tokens.

3.3 TABLE DESIGN

Customer Details:

Column Name	Data Type	Description
Customer Id	Varchar	Customer Id
First Name	Varchar	Customer First name
Last Name	Varchar	Customer's initial
Contact Number	Varchar	Contact Number
DOB	Varchar	Date Of Birth
E-Mailid	Varchar	E-Mail Id
Address	Varchar	Address

Branch Details:

Column Name	Data Type	Description
Customerid	Varchar	Customer Id
Branchcode	Varchar	Branch Code
Branchname	Varchar	Branch Name
Areacode	Varchar	Area code
pin	Varchar	Pin code

Login:

Column Name	Data Type	Description
Username	Varchar	User Name
pwd	Varchar	Password

4. PERFORMANCE EVALUATION

Testing is the process of executing a program with the intent of finding any errors.A good test of course has the high probability of finding a yet undiscovered error. A successful testing is the one that uncovers a yet undiscovered error.

A test is vital to the success of the system.. System test makes a logical assumption that if all parts of the system are correct, then goal will be successfully achieved. The candidate system is subjected to a variety of tests online like responsiveness, its value, stress and security. A series of tests are performed before the system is ready for user acceptance testing. The different types of testing performed here are:-

- Unit Testing
- Integration Testing
- Validation Testing
- Output Testing
- User Acceptance Testing

4.1 UNIT TESTING

Here we test each module individually and integrate the overall system. Unit testing focuses verification efforts even in the smallest unit of software design in each module. This is also known as "Module Testing". The modules of the system are tested separately. This testing is carried out in the programming style itself. In this testing each module is focused to work satisfactorily as regard to expected output from the module. There are some validation checks for the fields.

4.2 INTEGRATION TESTING

Data can be lost across an interface ,one module can have an adverse effect on the sub-functions,when combined may not produce the desired functions. Integrated testing is the systematic testing to uncover the errors within the interface. This testing is done with simple data and the developed system has run successfully with this simple.The need for integrated system is to find the overall system performance.

4.3 VALIDATION TESTING

At the culmination of the black box testing, software is completely assembled as a package. Interfacing errors have been uncovered and correct and final series of test,i.e.,validation test begins. Validation test van is defined with a simple defitnition that validation succeeds when the software functions in a manner that can be reasonably accepted by the customer.

4.4 USER ACCEPTANCE TESTING

User acceptance testing of the system is the key factor for the success of any system. The system under consideration is tested for user acceptance by constantly keeping in touch with prospective system at the time of development and making change whenever required. This is done with regard to the input screen design and output screen design.

4.5 OUTPUT TESTING

After performing validation testing,the next step is output testing of the proposed system. Since the system cannot be useful if it does not produce the required output. Asking the user about the format in which the system is required tests the output displayed or generated by the system under consideration. Here the output format is considered in two ways. One is on screen format and another one is printed format.The output format on the screen is found to be corrected as the format was designed in the system phase according to the user needs. As for the hard copy the output comes according to the specification requested by the user. Here the output testing does not result in any correction in the system.

Taking various kinds of data plays a vital role in system testing. After preparing the test data, system under study is tested using the test data. While testing, errors are again uncovered and corrected by using the above steps and corrections are also noted for future use. The system has been verified and validated by running test data and live data. First the system is tested with some sample test data that are generated with the knowledge of the possible range of values that are required to hold by the fields. The system runs successfully for the given test data and for the live data.

5. CONCLUSION

We have presented a novel highly automated approach for protecting Web applications from SQLIAs. Our approach consists of 1) identifying trusted data sources and marking data coming from these sources as trusted, 2) using dynamic tainting to track trusted data at runtime, and 3) allowing only trusted data to form the semantically relevant parts of queries such as SQL keywords and operators. Unlike previous approaches based on dynamic tainting, our technique is based on positive tainting, which explicitly identifies trusted (rather than untrusted) data in a program. This way, we eliminate the problem of false negatives that may result from the incomplete identification of all untrusted data sources. False positives, although possible in some cases, can typically be easily eliminated during testing. Our approach also provides practical advantages over the many existing techniques whose application requires customized and complex runtime environments: It is defined at the application level, requires no modification of the runtime system, and imposes a low execution overhead.

6. FUTURE ENHANCEMENTS

The continuous updation results to track taint information at the character level and use a syntax aware evaluation to examine tainted input. Raising to the level of granularity of strings, which introduces imprecision in modelling string operations and declassification rules, instead of syntax aware evaluation, helps to assess whether a query string contains an attack. SQL Guard is an approach similar to SQL Check. These approaches scan the application and leverage information flow analysis or heuristics to detect code that could be vulnerable to SQL Injection attacks.

APPENDIX I-SOURCE CODE

DATABASE CODING

```
package DbConnection;
import java.io.*;
import java.sql.*;
import java.io.*;
import java.util.ArrayList;
public class DbConnection
{

    private Connection conn;
    int i=0;
    int n=0;
    ArrayList<String> arr=new ArrayList<String>();
    File file = null;
    FileReader freader = null;
    LineNumberReader lnreader = null;
    String line = "";
    public DbConnection()
    {
        super();
    }

   public boolean connect() throws ClassNotFoundException,SQLException
   {

        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        conn=DriverManager.getConnection("jdbc:odbc:Driver={SQL Server};
        Server=(local);Database=WASP","sa","goodboy");
        return true;
   }
  public void close() throws SQLException
  {
      conn.close();
  }
```

```
public ResultSet exeSQL(String query) throws SQLException
{

      Statement stmt=conn.createStatement();
      System.out.println("_____"+i);
      ResultSet rs=stmt.executeQuery(query);
      return (rs==null)? null:rs;

}
public int exeUpdate(String query) throws SQLException
{
       Statement stmt1=conn.createStatement();
       i=stmt1.executeUpdate(query);
       return(i==0)? 0:i;
}

public int inputVerifier(String userName,String pwd)
{
      String data[]=userName.split(" ");
      String data1[]=pwd.split(" ");
      System.out.println("User Name::"+userName+"Password:::"+pwd);
      for(int m=0;m<data.length;m++)
        {
          if(data[m].equalsIgnoreCase("1=1")||data[m].equalsIgnoreCase("-
              -")||pwd.equalsIgnoreCase("--")||
          pwd.equalsIgnoreCase("1=1")||userName.equalsIgnoreCase("—
          ")||userName.equalsIgnoreCase("1=1"))
             {
                i=1;
                break;
             }

       else
            {
              i=2;
             System.out.println("Check Else Condition");
            }
     }
      return i;

}
```

```
public int WASP(String query)
{
    System.out.println("query"+query);
    String data[]=query.split(" ");
    Try
       {
            file = new File("C:\\Program Files\\Apache Software
                   Foundation\\Tomcat 5.5\\webapps\\WASP\\WEB-
                   INF\\classes\\DbConnection\\foo.txt");
            freader = new FileReader(file);
            lnreader = new LineNumberReader(freader);
           while ((line = lnreader.readLine()) != null){
           i=lnreader.getLineNumber();
                          arr.add(line);
       }
    freader.close();
    lnreader.close();

    for(int i=0;i<data.length;i++)
    {
       System.out.println("    "+data[i]);

       for(int k=0;k<arr.size();k++)
       {
          if(arr.get(k).equalsIgnoreCase(data[i]))
            {
              System.out.println("Matched::::::::::::::"+data[i]);
              n=1;
            }
         else if((data[i].equalsIgnoreCase("select"))||
                   (data[i].equalsIgnoreCase("from"))||
                   (data[i].equalsIgnoreCase("*"))||
                   (data[i].equalsIgnoreCase("where"))||
                   (data[i].equalsIgnoreCase("and"))||
                    (data[i].equalsIgnoreCase("values")))
           {

              System.out.println("Matched::::::::::::::"+data[i]);
               n= 1;
           }
```

```
        Elseif((data[i].equalsIgnoreCase("OR"))||
           (data[i].equalsIgnoreCase("union"))||
           (data[i].equalsIgnoreCase("drop")))
         {
          System.out.println("Not Matched");
          n=2;
          break;
      }

  } // Inner Loop

} // Outer Loop


}

   catch(Exception e){
        System.out.println(e);
      }

    return n;

 }

}
```

TAUTOLOGIES QUERY CODING

```
<%!
String button1,button2,query,customerID,pinNumber,textArea;
int i=0,v=5;
%>
<%
customerID=request.getParameter("T1");
pinNumber=request.getParameter("T2");
button1=request.getParameter("B1");
button2=request.getParameter("B2");
textArea=request.getParameter("S1");
textArea="";
if(customerID!=null&&pinNumber!=null)
{
  if(button1!=null)
    {
      if(button1.equals("View Query"))
        {
          query="select * from BranchDetails where customerid='"
                +customerID+"' and 1=1 --  pin='"+pinNumber+"'";
           i=1;
        }
    }
  else if(button2!=null)
    {
      if(button2.equals("SQL Result"))
        {
          System.out.println(query);
           response.sendRedirect("TautologiesResult.jsp?query="+query);
        }
    }
}
```

UNION QUERIES

```
<%@ page language="Java" import="java.sql.*" import ="java.io.*" %>
   <jsp:useBean id="db" scope="request"
    class="DbConnection.DbConnection" />
   <jsp:setProperty name="db" property="*" />

<%!
 String
query,customerId,branchCode,branegName,accountNumber,accountType,cc
Number,pinNumber;
   ResultSet rs;
%>

<%
    query=request.getParameter("query");
        try
          {
             db.connect();
              rs=db.exeSQL(query);

%>

<%@ page language="Java" import="java.sql.*" import ="java.io.*" %>
<jsp:useBean id="db" scope="request"
class="DbConnection.DbConnection" />
<jsp:setProperty name="db" property="*" />

<%!
String
query,customerId,branchCode,branegName,accountNumber,accountType,cc
Number,pinNumber,pin;
ResultSet rs;
%>

<%
query=request.getParameter("query");
System.out.print("Here the query");
try
{
```

```
db.connect();
rs=db.exeSQL(query);
        if(rs.next()){
                pin=rs.getString(1);
                System.out.println("PIN"+pin);
        }

}
catch(Exception e){
        System.out.println("Error"+e);
}
%>
```

PIGGYBACKED QUERIES

```
<%!
    String button1,button2,query,customerID,pinNumber,textArea;
    int i=0,v=5;
%>
<%

        customerID=request.getParameter("T1");
         pinNumber=request.getParameter("T2");
         button1=request.getParameter("B1");
         button2=request.getParameter("B2");
         textArea=request.getParameter("S1");

        textArea="";
if(pinNumber!=null)
   {
      if(button1!=null)
          {
            if(button1.equals("View Query"))
           {]
```

```
    query="select * from BranchDetails where customerid='"+customerID+"'
          or 1=1 or pin='"+pinNumber+"' drop table BranchDetails";
             i=1;
          }

        {

else if(button2!=null)
      {
         if(button2.equals("SQL Result"))
            {

              System.out.println(query);
            response.sendRedirect("PiggybackedResult.jsp?query="+query);
            }
       }

else
   {
       System.out.println("Text Area============================");
       query="";

   }
}
%>


<%@ page language="Java" import="java.sql.*" import ="java.io.*" %>
 <jsp:useBean id="db" scope="request"
  class="DbConnection.DbConnection" />
  <jsp:setProperty name="db" property="*" />

<%!
  String
query,customerId,branchCode,brancgName,accountNumber,accountType,cc
 Number,pinNumber;
   ResultSet rs;
 %>

<%
```

```
    query=request.getParameter("query");
     try
      {
         db.connect();
         rs=db.exeSQL(query);
%>
```

MALFORMED QUERIES

```
<%!
      String button1,button2,query,customerID,pinNumber,textArea;
       int i=0,v=5;
%>
<%

            customerID=request.getParameter("T1");
             pinNumber=request.getParameter("T2");
             button1=request.getParameter("B1");
             button2=request.getParameter("B2");
             textArea=request.getParameter("S1");

       textArea="";
if(pinNumber!=null)
  {
    if(button1!=null)
        {
          if(button1.equals("View Query"))
              {

                query="select Branchname from BranchDetails where
                customerid='"+customerID+"' AND
                pin=convert(int,(select top 1 name from sysobjects where
type='u'))";
                  i=1;
              }

        }
```

```
 else if(button2!=null)
   {
     if(button2.equals("SQL Result"))
     {

        System.out.println(query);
        response.sendRedirect("MalformedResult.jsp?query="+query);
     }
}

else
   {
      System.out.println("Text Area==========================");
       query="";

   }
}
%>

<%@ page language="Java" import="java.sql.*" import ="java.io.*" %>
<jsp:useBean id="db" scope="request"
class="DbConnection.DbConnection" />
<jsp:setProperty name="db" property="*" />

<%!
     String
query,customerId,branchCode,brancgName,accountNumber,accountType,cc
Number,pinNumber,pin,error;
      ResultSet rs;
     int i;
%>

<%
         query=request.getParameter("query");
          System.out.print("Here the query");
```

```
 try
    {
          db.connect();
          qrs=db.exeSQL(query);
       while(rs.next()){
             pin=rs.getString(1);
             System.out.println("PIN"+pin);
    }
}
    catch(Exception e){
       error=""+e;
       System.out.println("Error"+e);%>

<%}%>
```

WASP RESULT

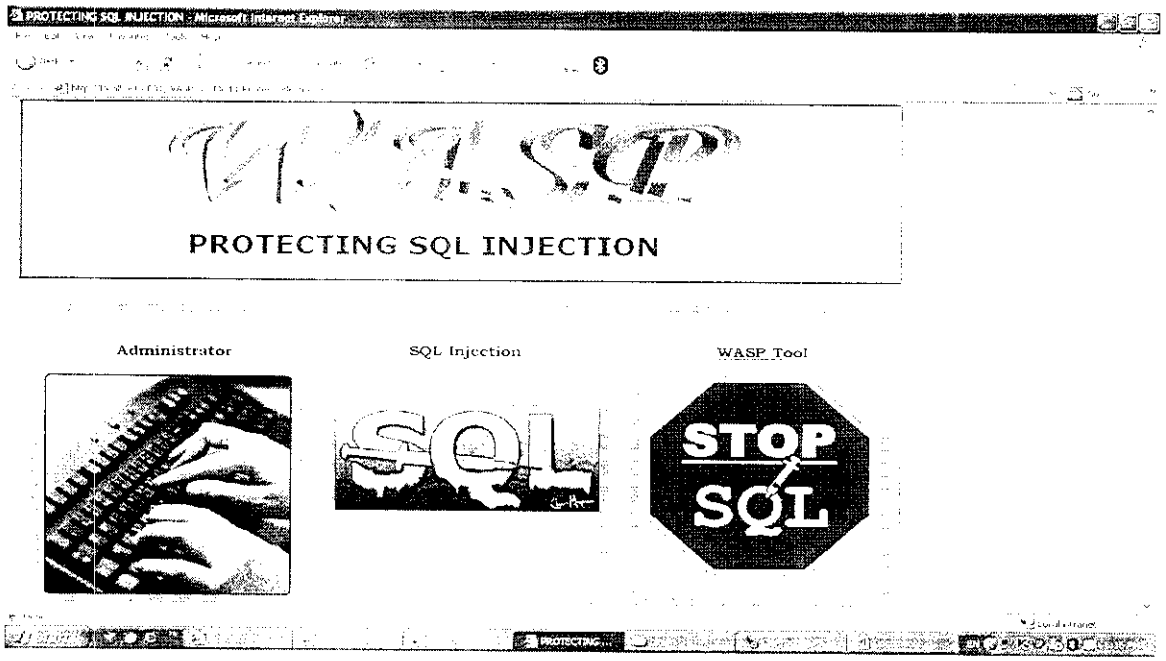
```
<%@ page language="Java" import="java.sql.*" import ="java.io.*" %>
<jsp:useBean id="db" scope="request"
class="DbConnection.DbConnection" />
<jsp:setProperty name="db" property="*" />

<%!
    String
query,customerId,branchCode,branegName,accountNumber,accountType,cc
Number,pinNumber;
    ResultSet rs;
    int i;
%>

<%
    query=request.getParameter("query");
    System.out.println(query);
    try
        {
            db.connect();
            i=db.WASP(query);
```

```
 System.out.println("WASP III::::"+i);
                if(i==1)
                  {
                       rs=db.exeSQL(query);
                       System.out.println("Query Excuted Successful!");

%>


<html>
<head>
 <meta http-equiv="Content-Language" content="en-us">
 <meta http-equiv="Content-Type" content="text/html; charset=windows-
1252">
  <title>New Page 1</title>
  <!--mstheme--><link rel="stylesheet" href="comp1011.css">
 <meta name="Microsoft Theme" content="compass 1011">
</head>
<body>
 <form method="POST" action="WASPResult.jsp">
<a href="UserLogin.jsp">
<img border="0" src="back.gif" width="60" height="45"></a></div>
<tr>
<td width="111" align="center"><b>Customer Id</b></td>
<td width="116" align="center"><b>Branch Name</b></td>
<td width="123" align="center"><b>Branch Code</b></td>
<td width="123" align="center"><b>Area Code</b></td>
td align="center"><b>Pin Number</b></td>
                       </tr>
                       <%
                       while(rs.next())
                       {
                       customerId=rs.getString(1);

                       System.out.println(customerId);
                       branchCode=rs.getString(2);
                       branegName=rs.getString(3);
                       accountNumber=rs.getString(4);;
                       pinNumber=rs.getString(5);
                       System.out.println(""+branchCode);
                       %>
```

```
<tr>
                    <%
                    }
                    }
                    else if(i==2)
                    {
                    System.out.println("ERROR ERROR ERROR ");
            response.sendRedirect("ErrorPage.jsp");
            System.out.println("ERROR Query");
            db.close();
                    }
                    }
                    catch(Exception e)
                    {
                    System.out.println(e);
                    }
                    %>
            </table>
        </div>
        <p> </div>
</form>
</body>
</html>
```

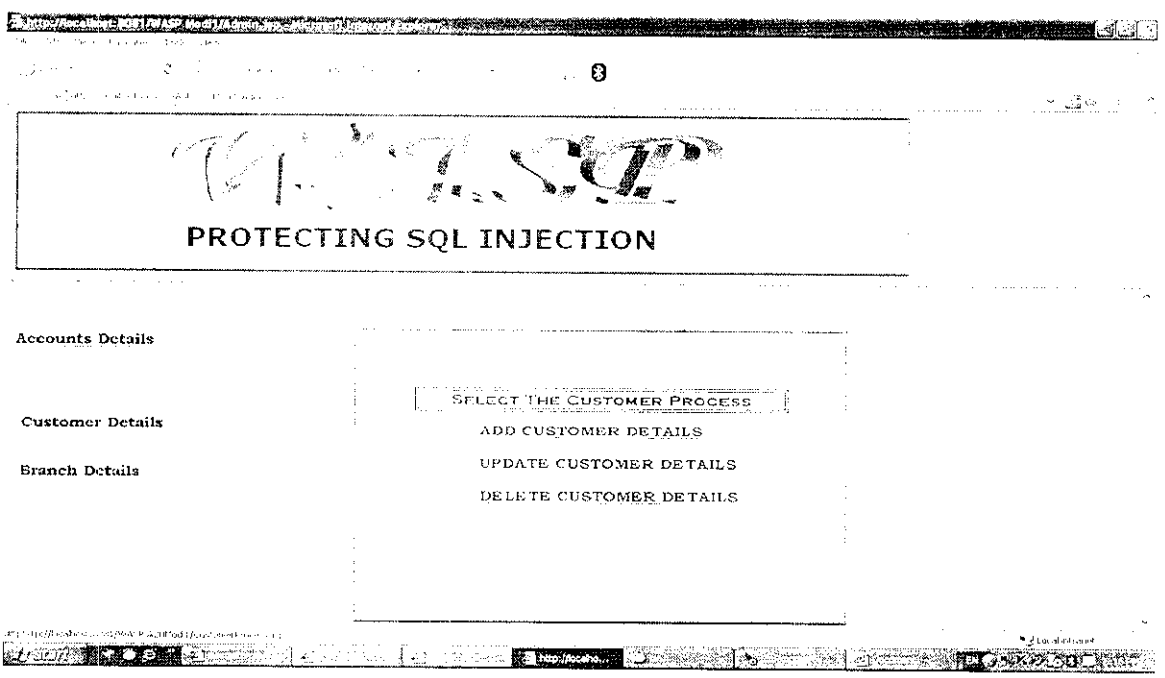
APPENDIX II - SCREEN SHOTS:

LOGIN PAGE:

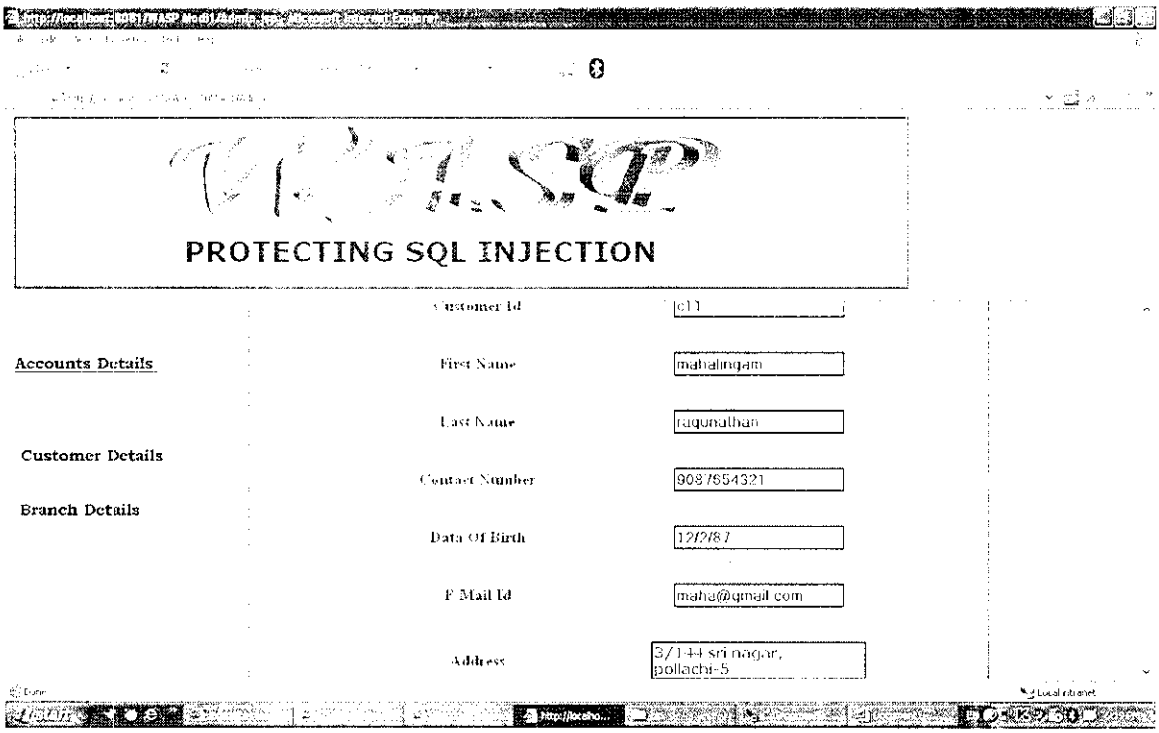


MODULE 1: ADMINISTRATOR

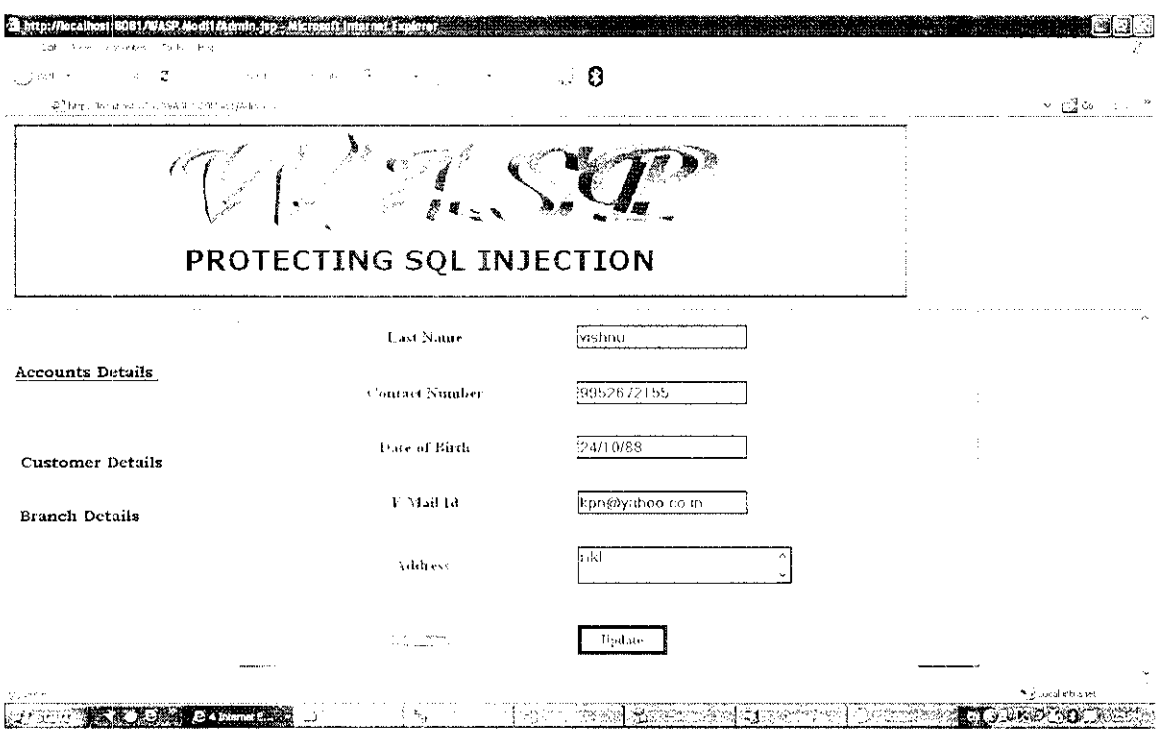
CUSTOMER SELECTION PROCESS:



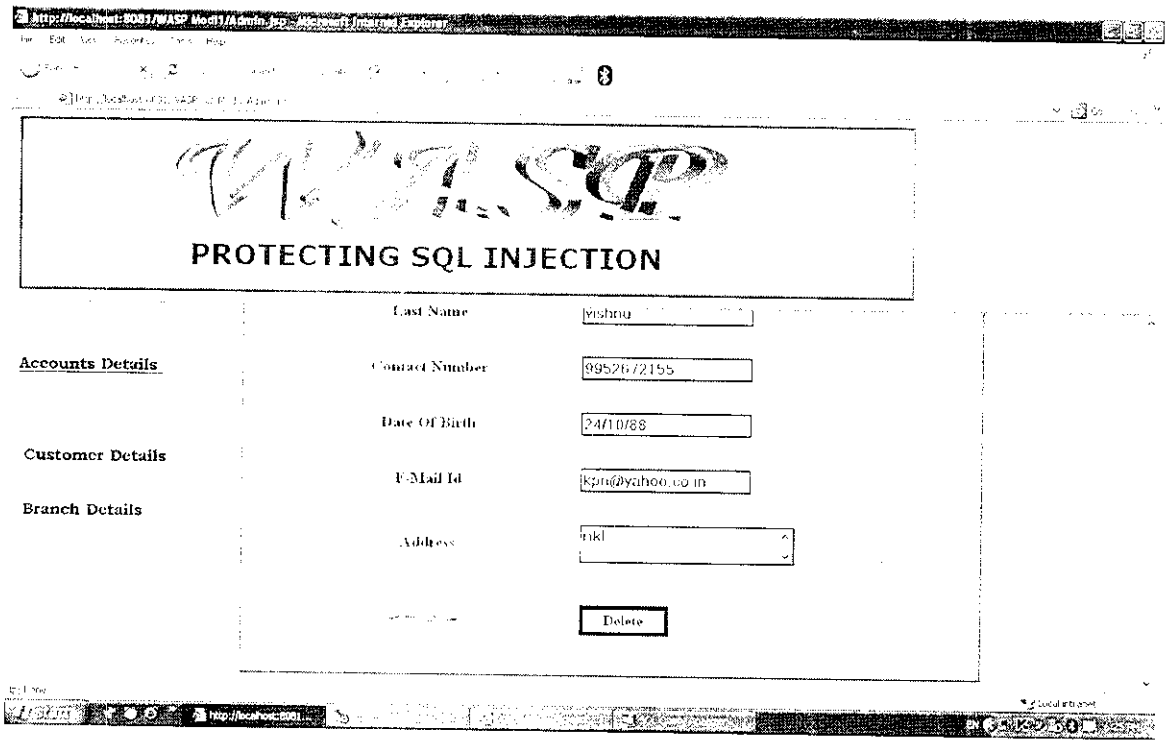
ADD CUSTOMER DETAILS:



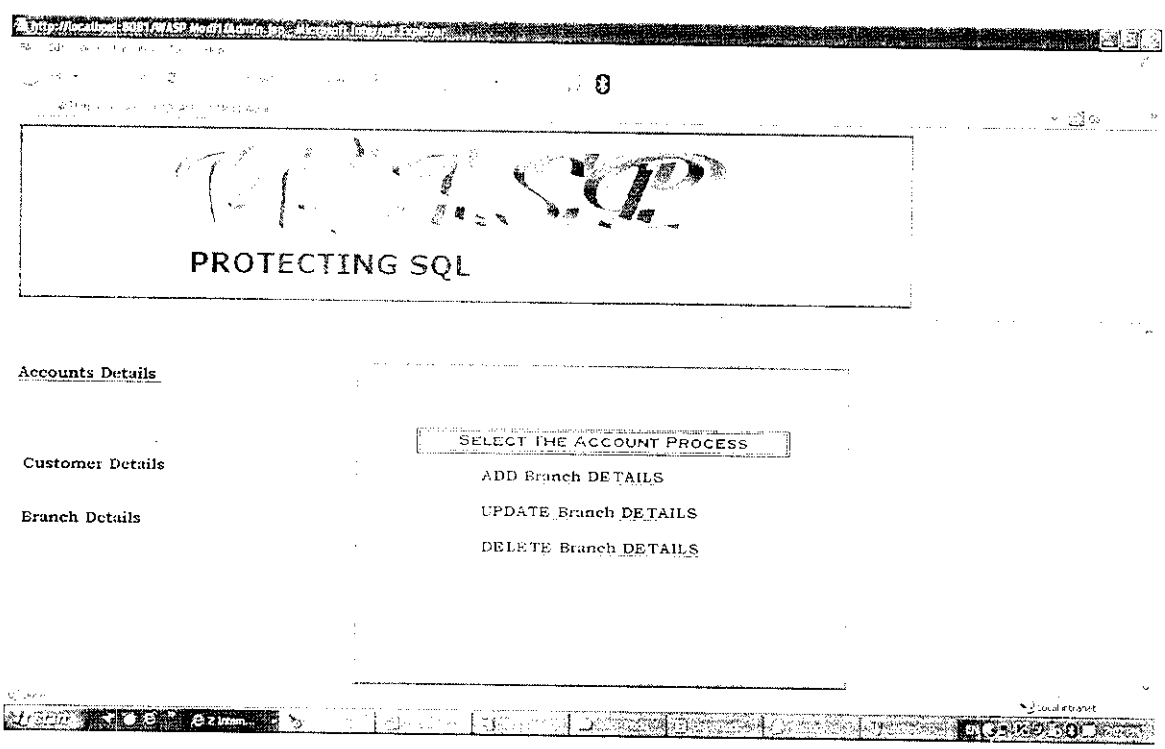
UPDATE CUSTOMER DETAILS:



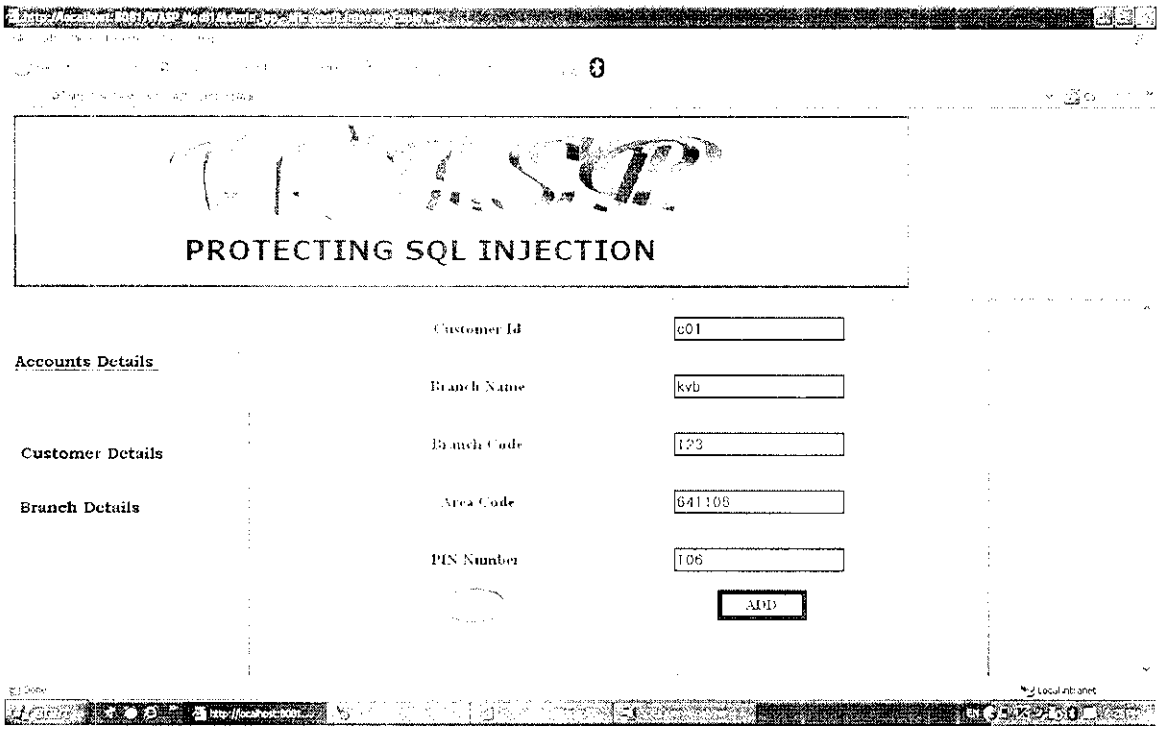
DELETE CUSTOMER DETAILS:



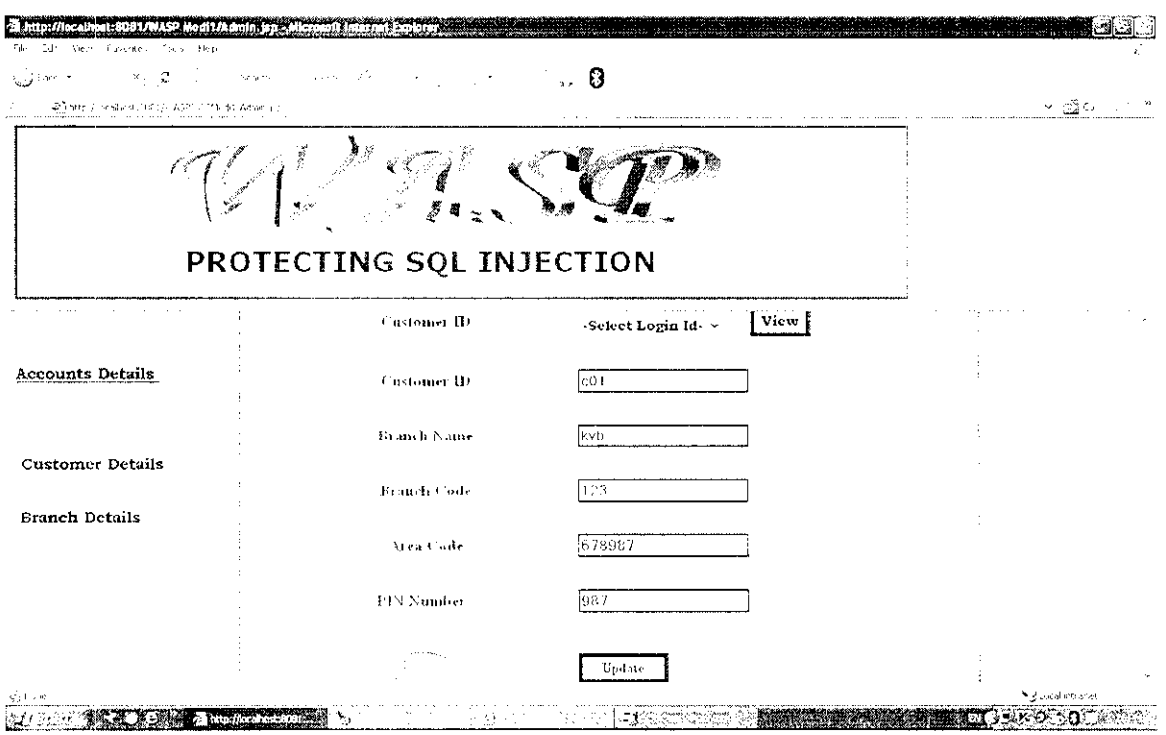
ACCOUNT SELECTION PROCESS:



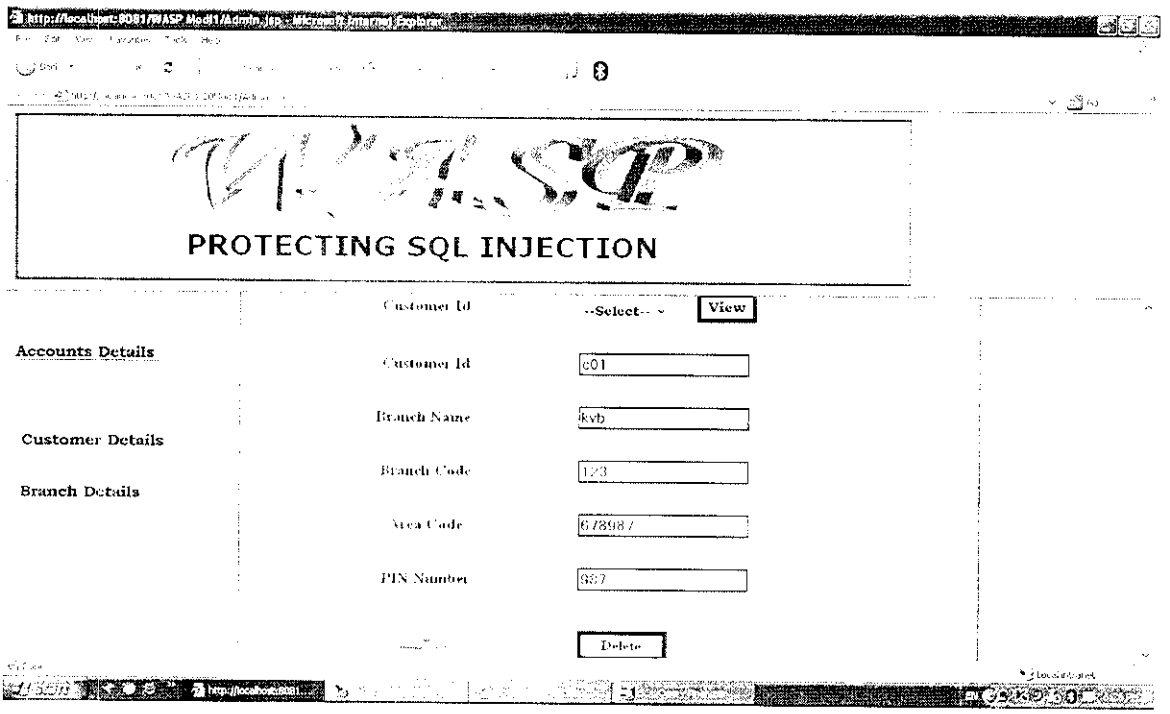
ADD BRANCH DETAILS:



UPDATE BRANCH DETAILS:

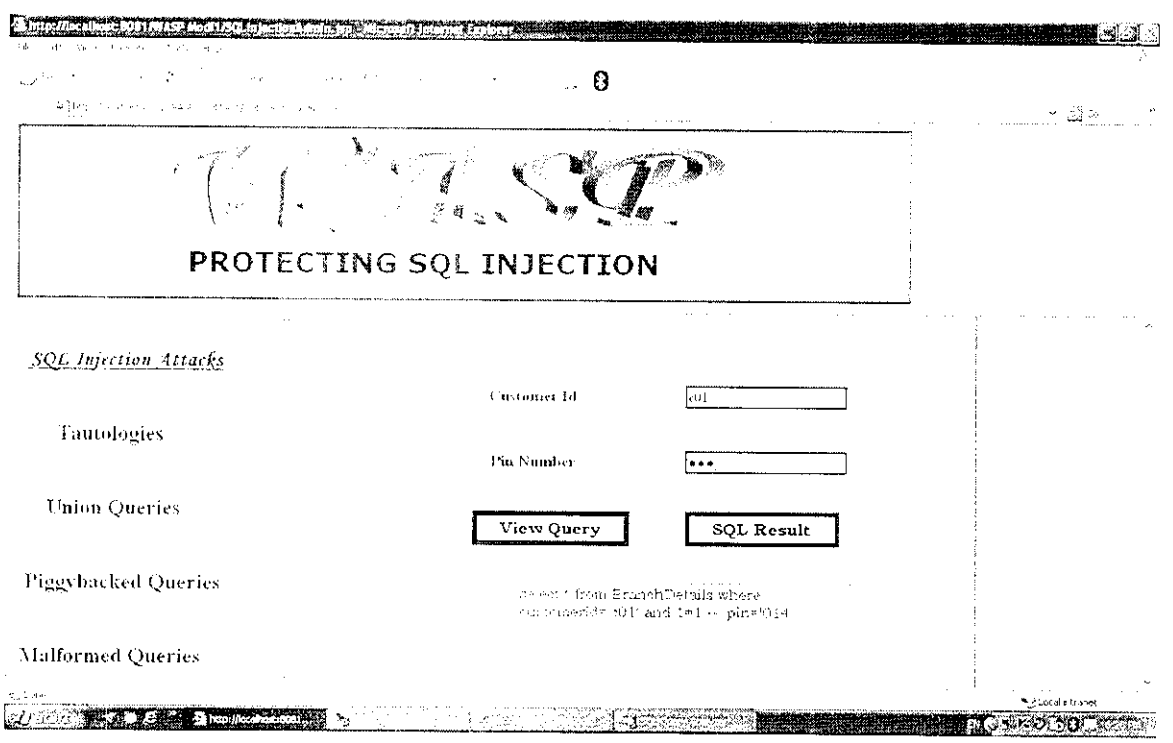


DELETE BRANCH DETAILS:

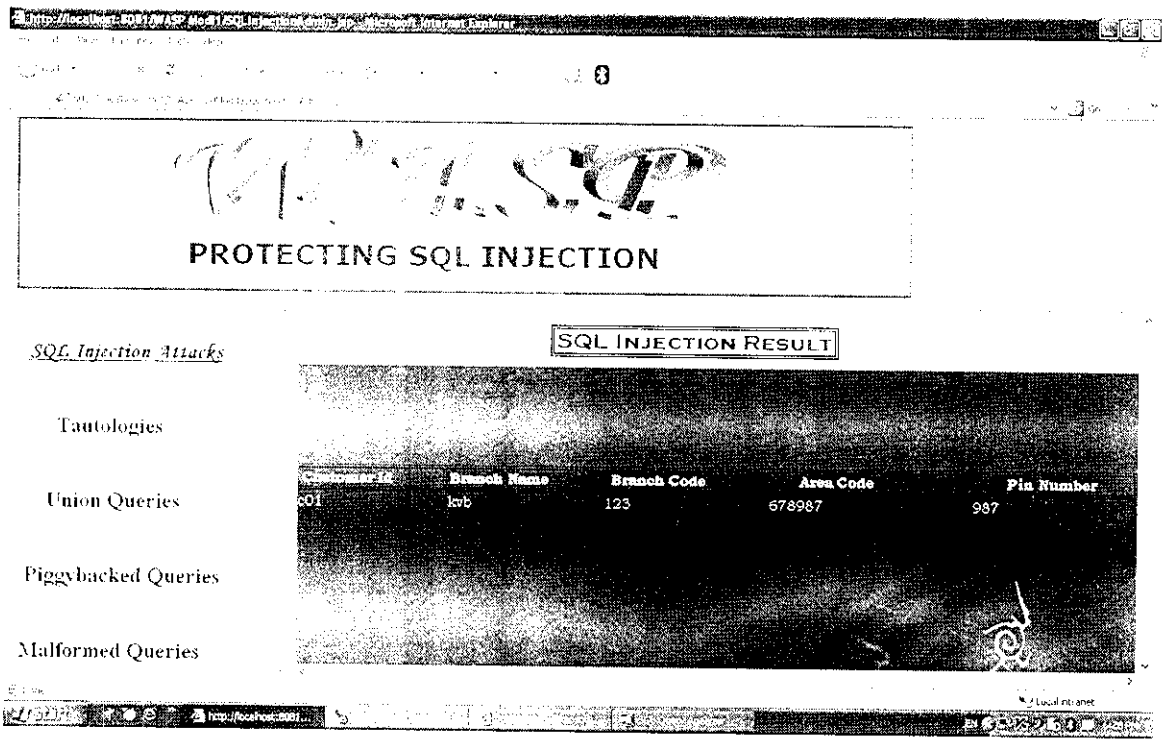


MODULE 2: SQL INJECTION ATTACKS

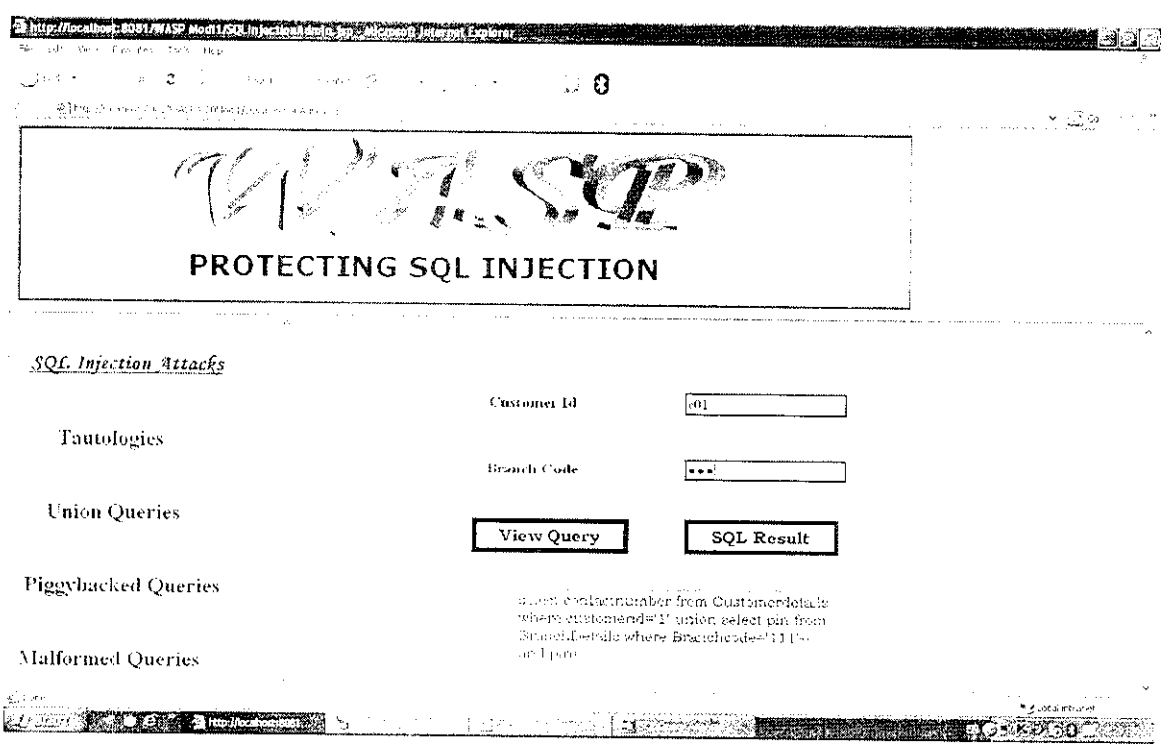
SQL INJECTION ATTACK 1::TAUTOLOGIES



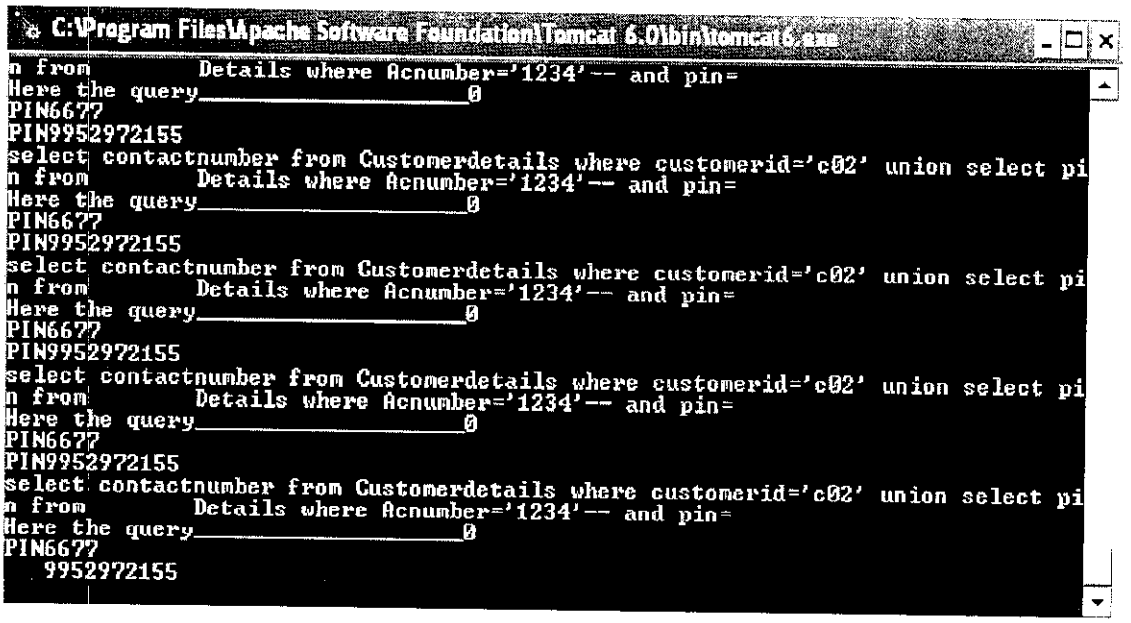
RESULT: Leakage of Customer data



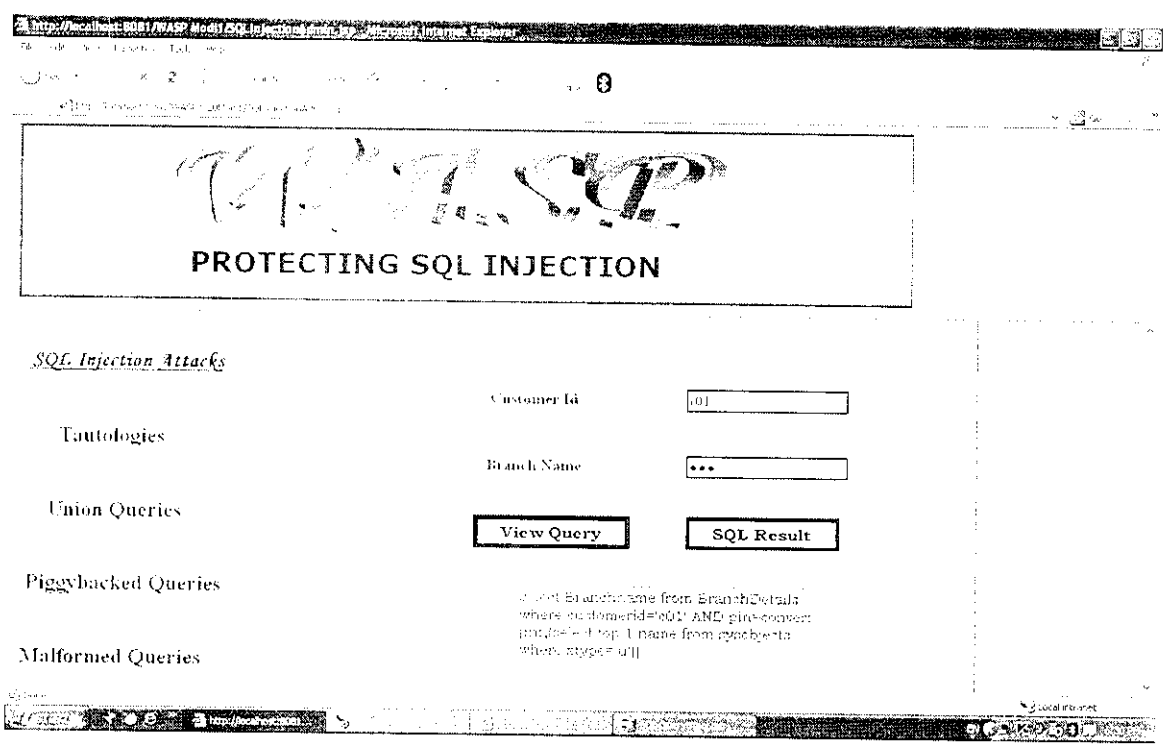
SQL INJECTION ATTACK 2::UNION QUERIES



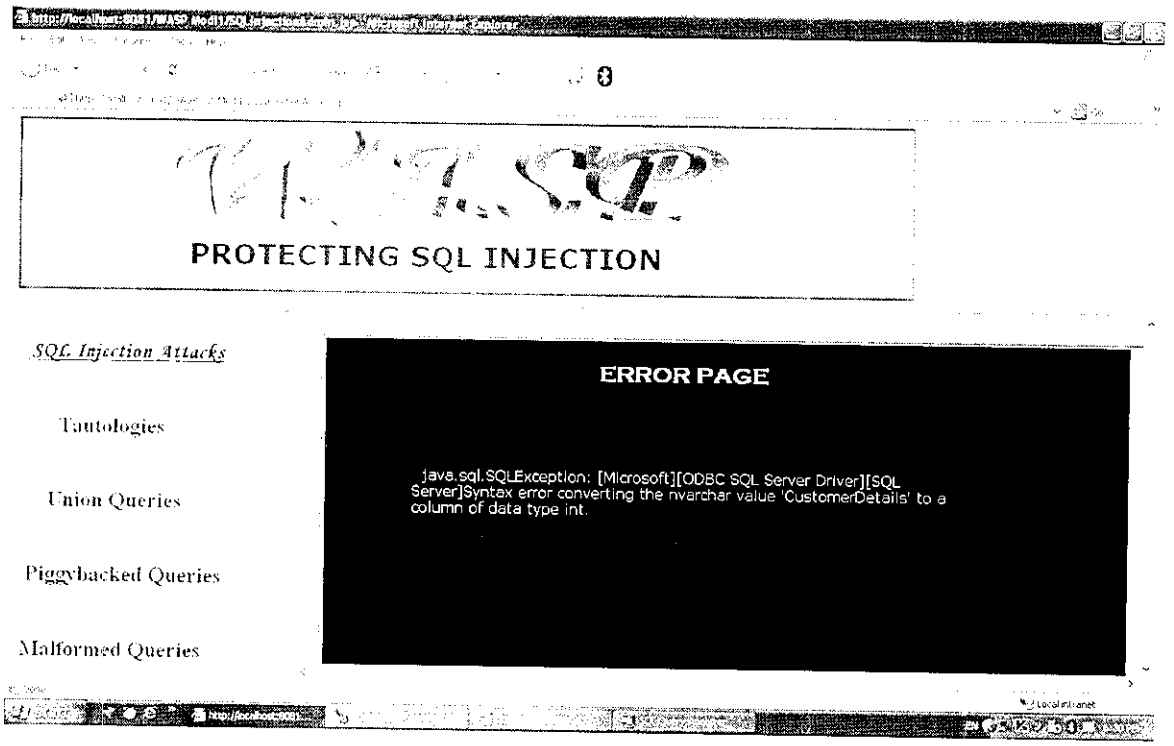
RESULT: Pin number and Contact number are retrieved.



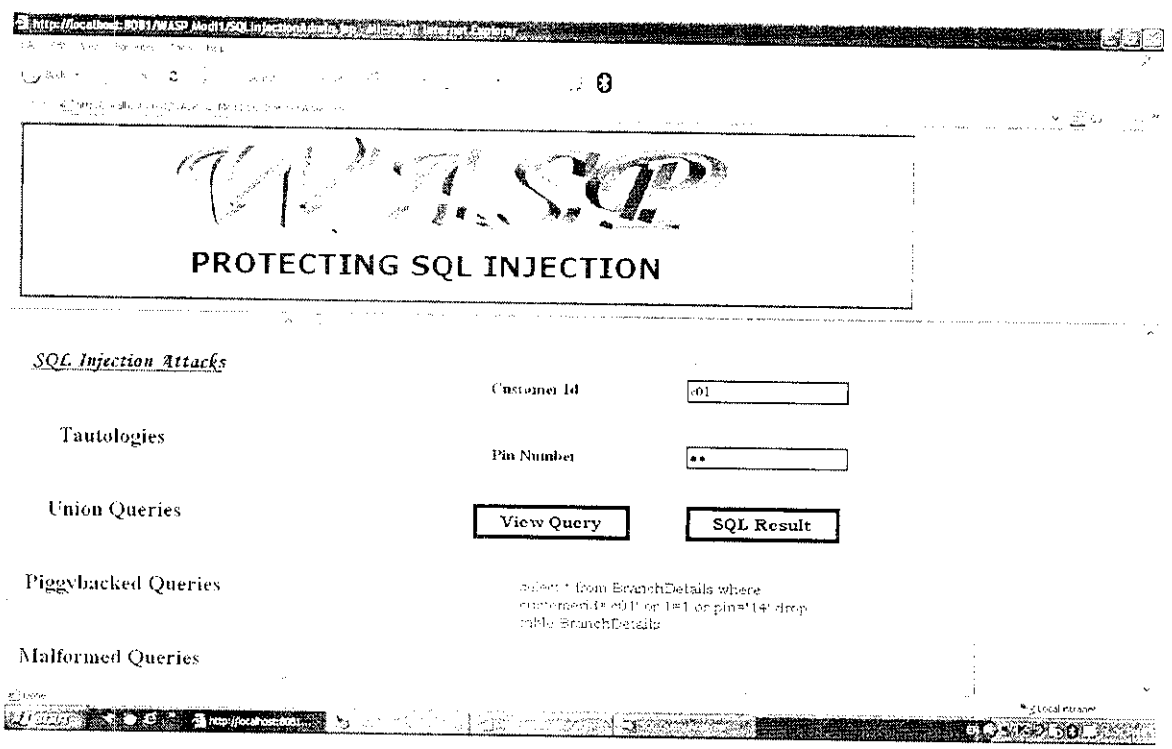
SQL INJECTION ATTACK 3::MALFORMED QUERIES



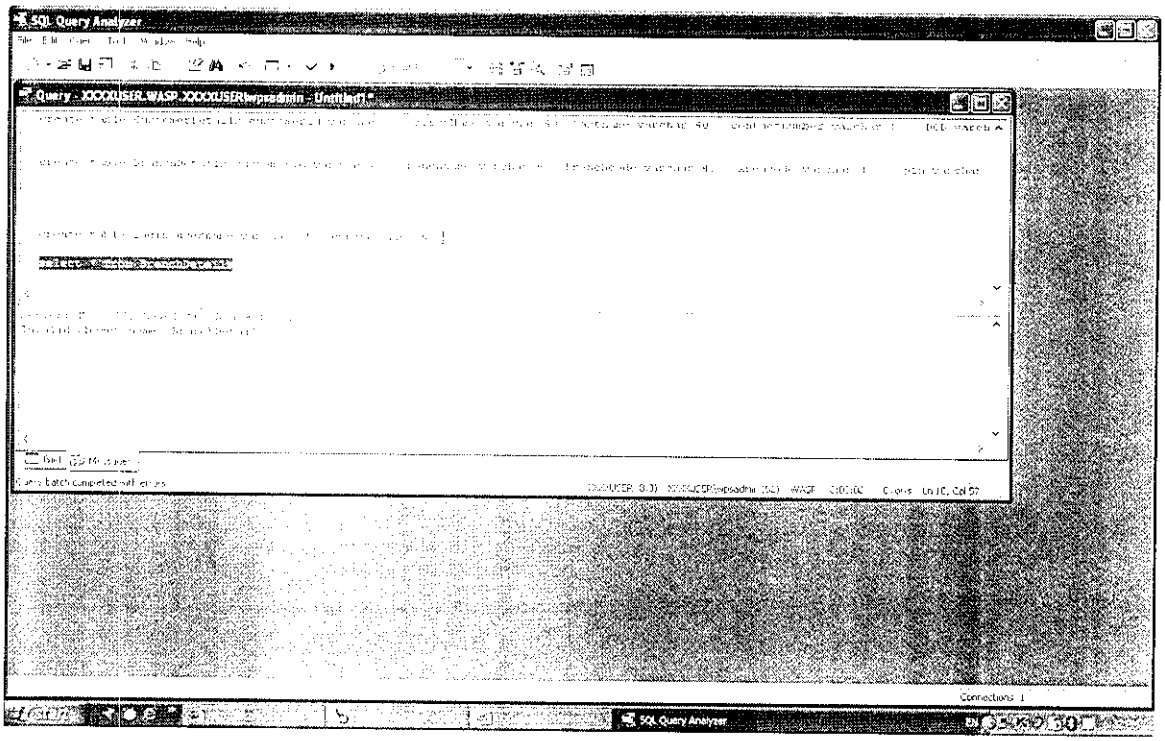
RESULT: Exception is Raised.



SQL INJECTION ATTACK 4::PIGGYBACKED QUERIES

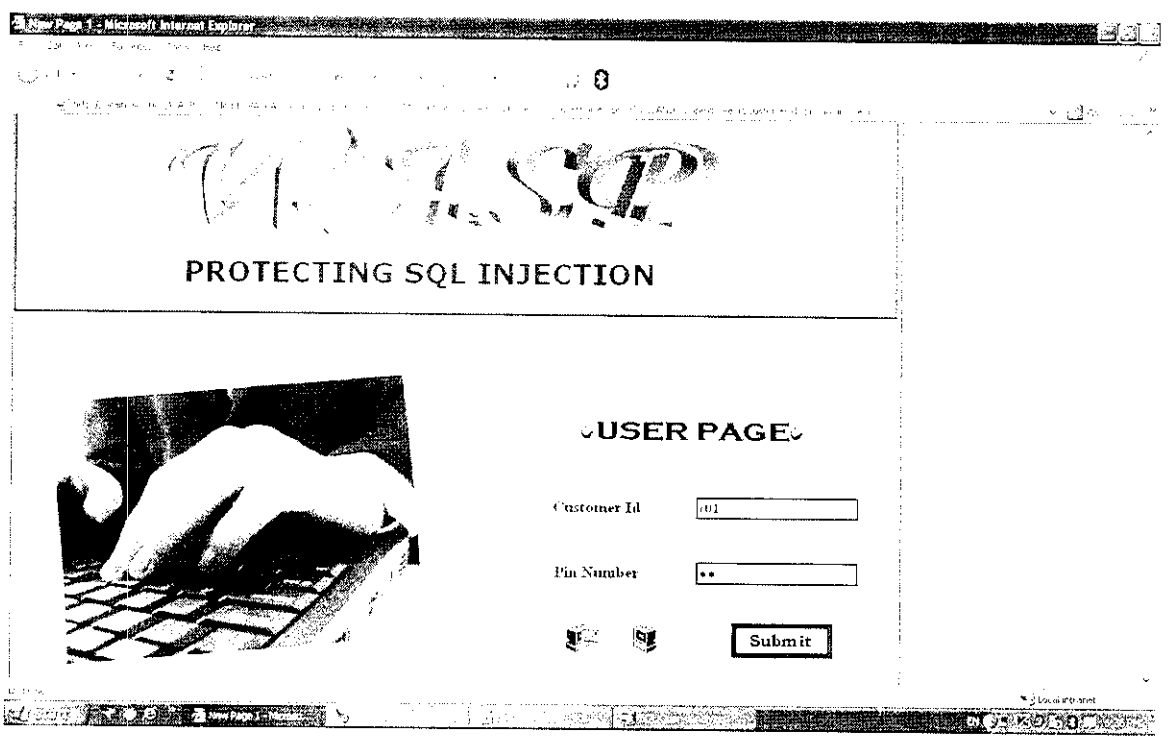


SQL SERVER RESULT: Loss of BranchDetails table.

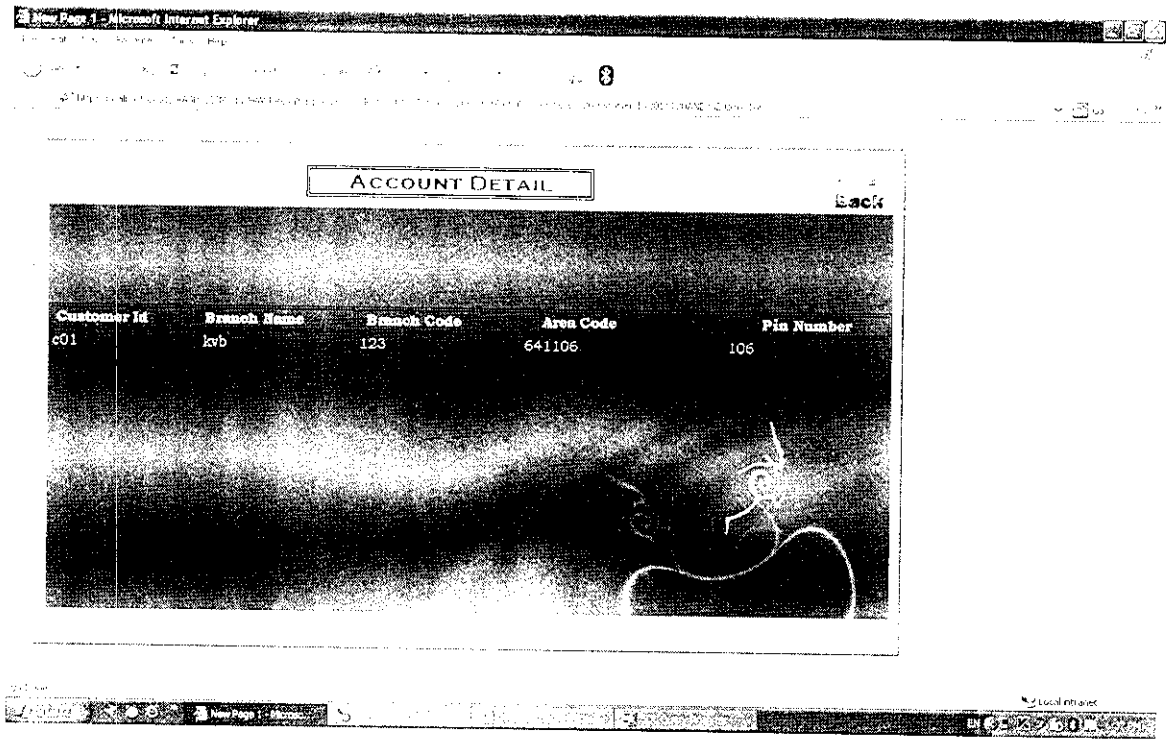


MODULE 3: WEB APPLICATION SQL INJECTION PREVENTER

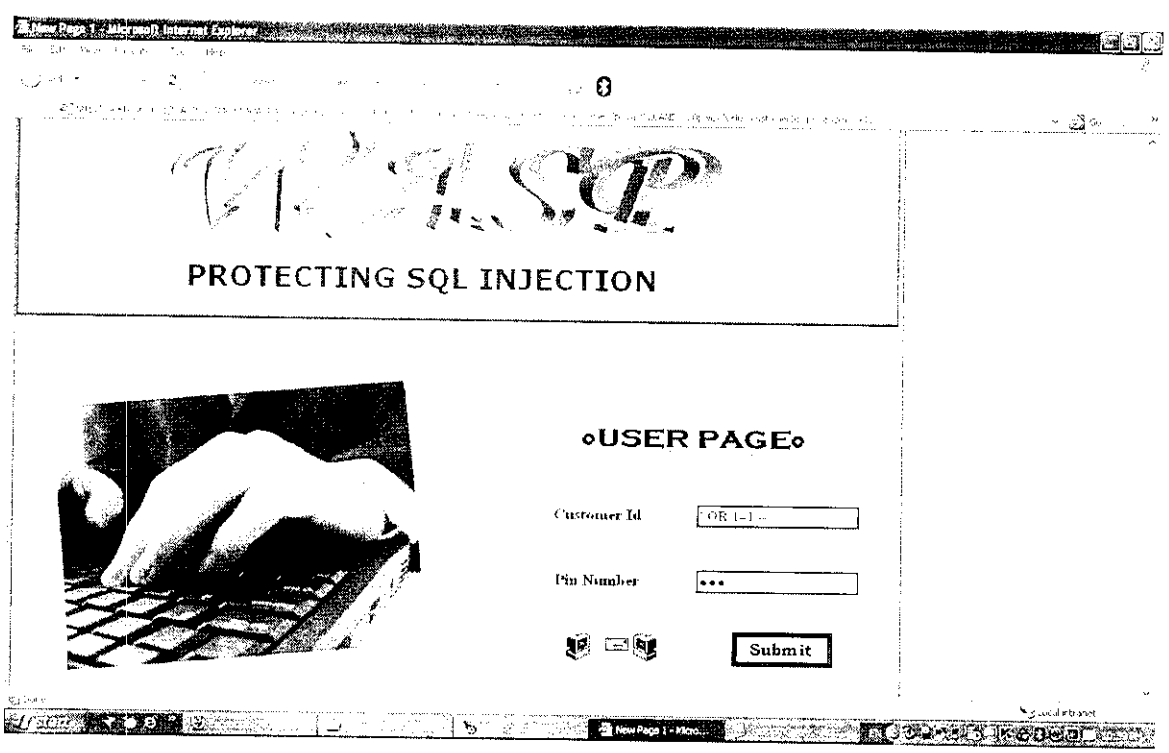
CUSTOMER LOGIN:



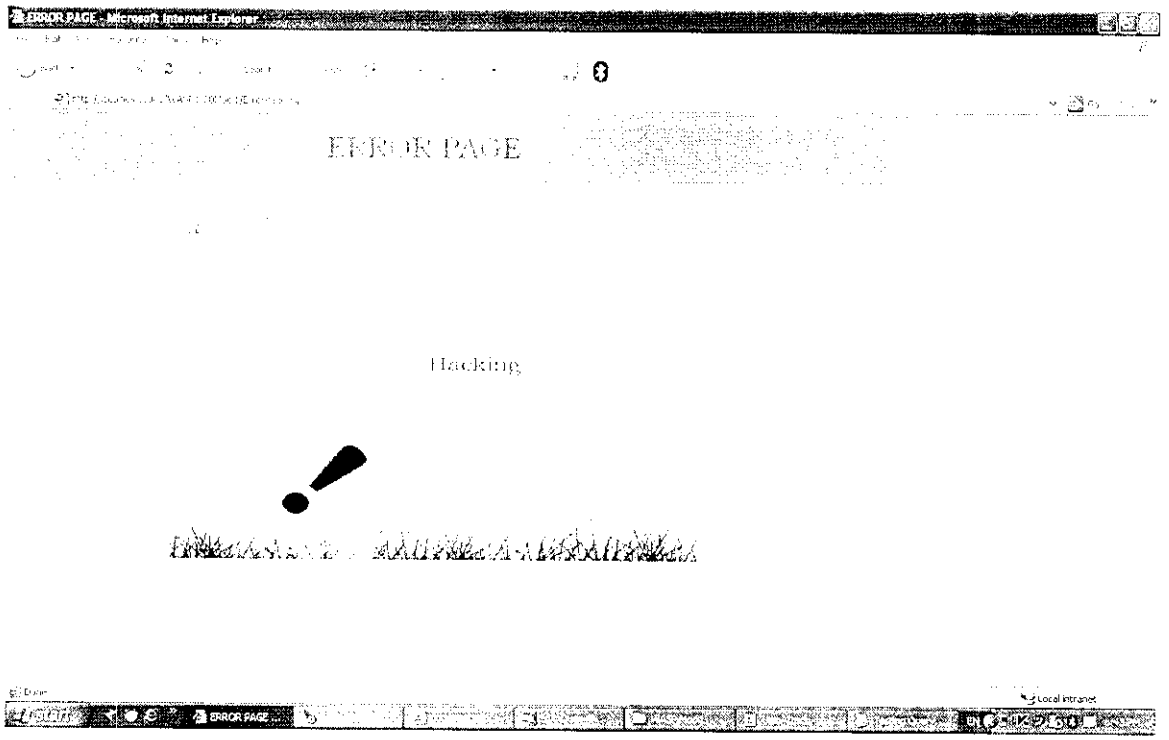
RESULT: Retrieval of data from the database.



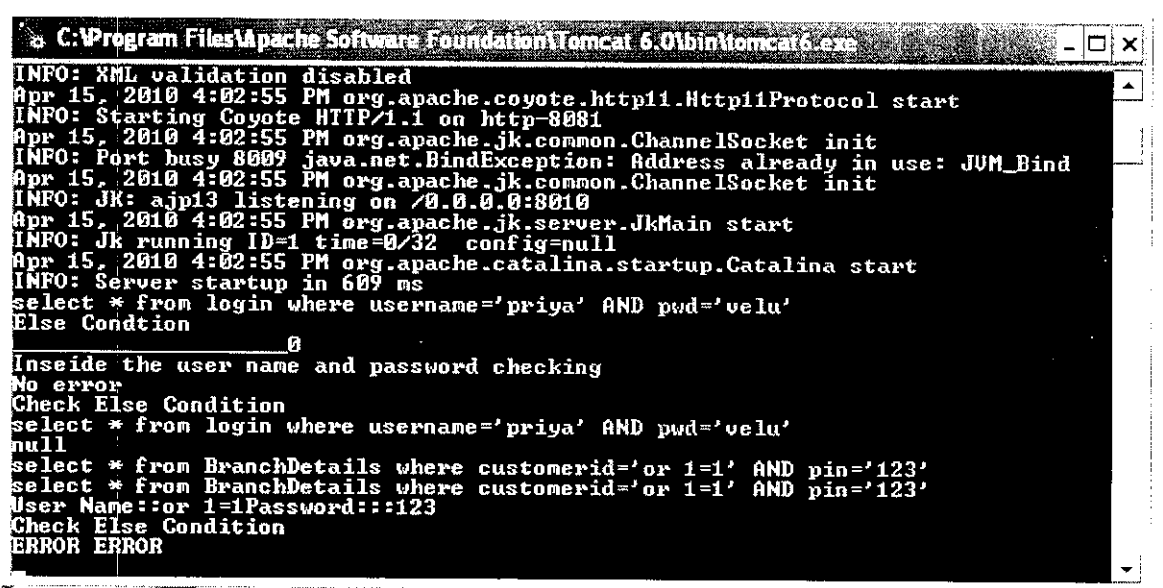
SQL INJECTION PREVENTER AGAINST TAUTOLOGIES:



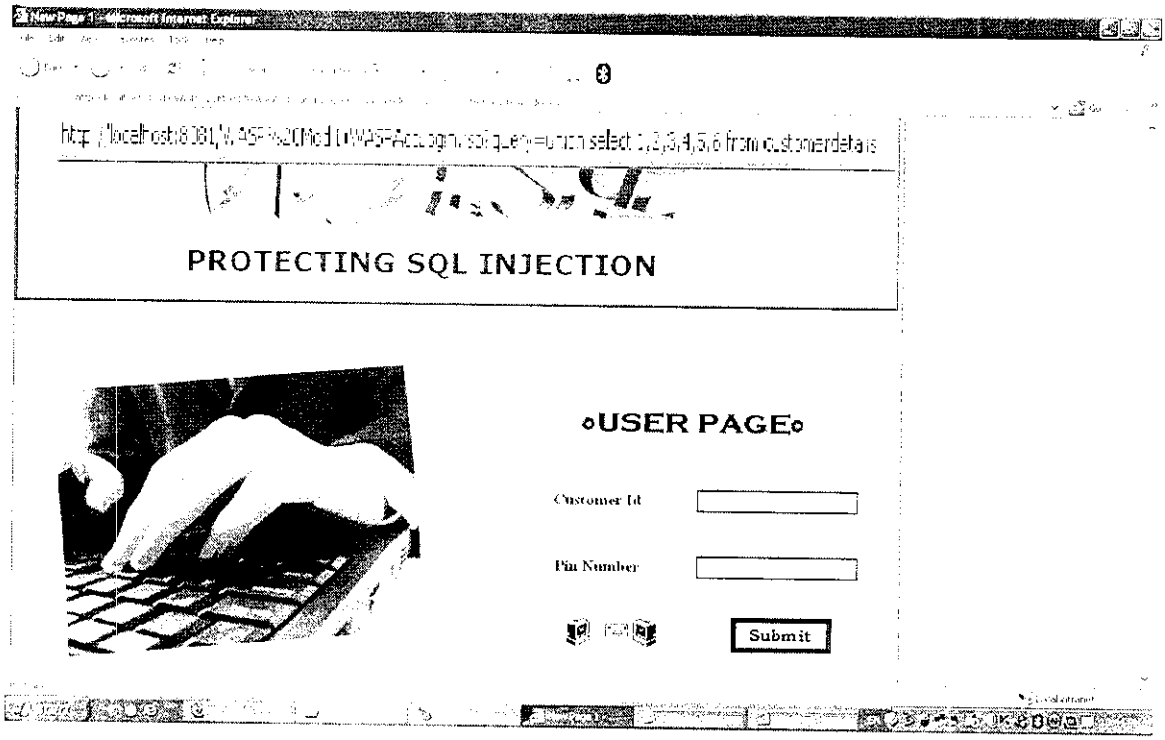
ERROR PAGE: CLIENTSIDE



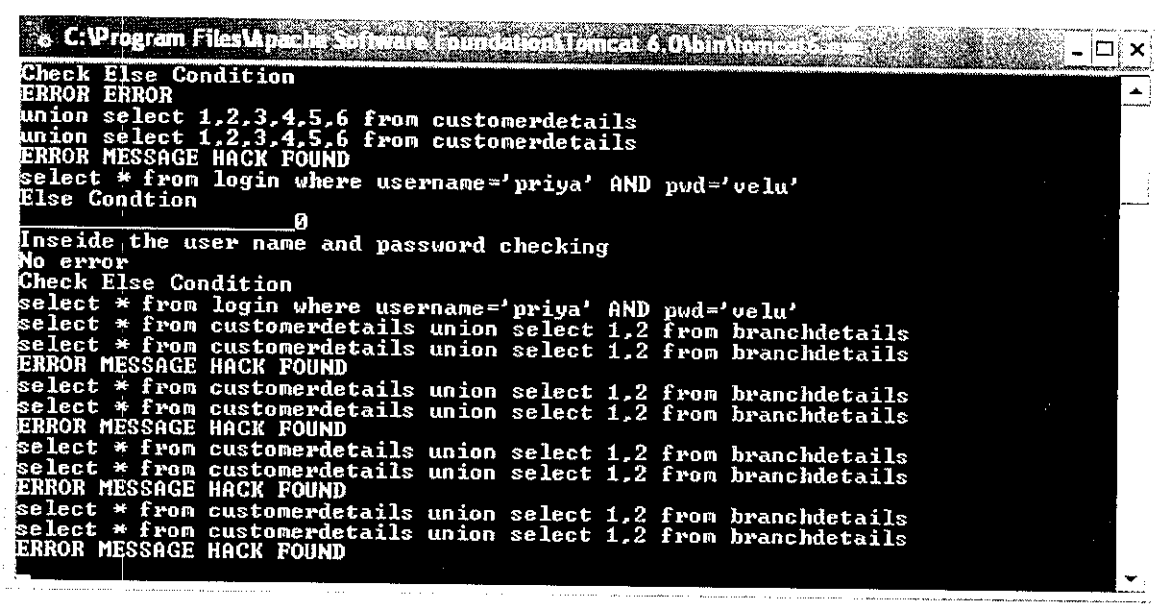
ERROR PAGE: SERVERSIDE



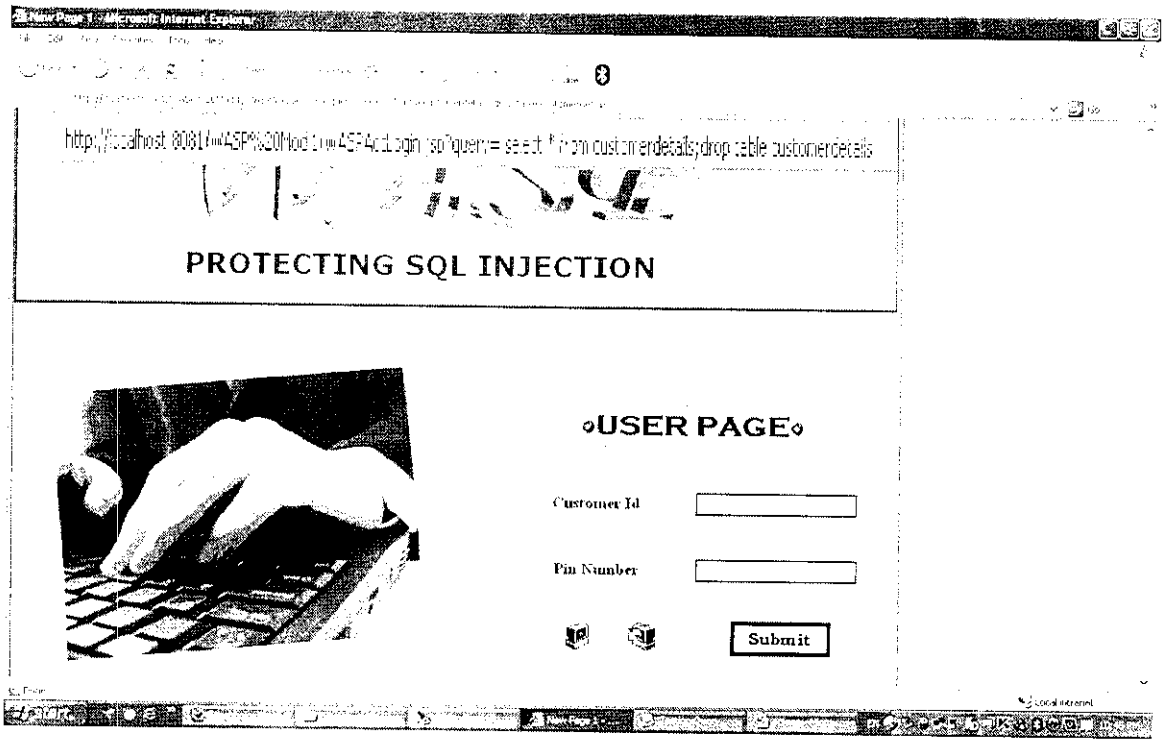
SQL INJECTION PREVENTER AGAINST UNIONQUERIES:



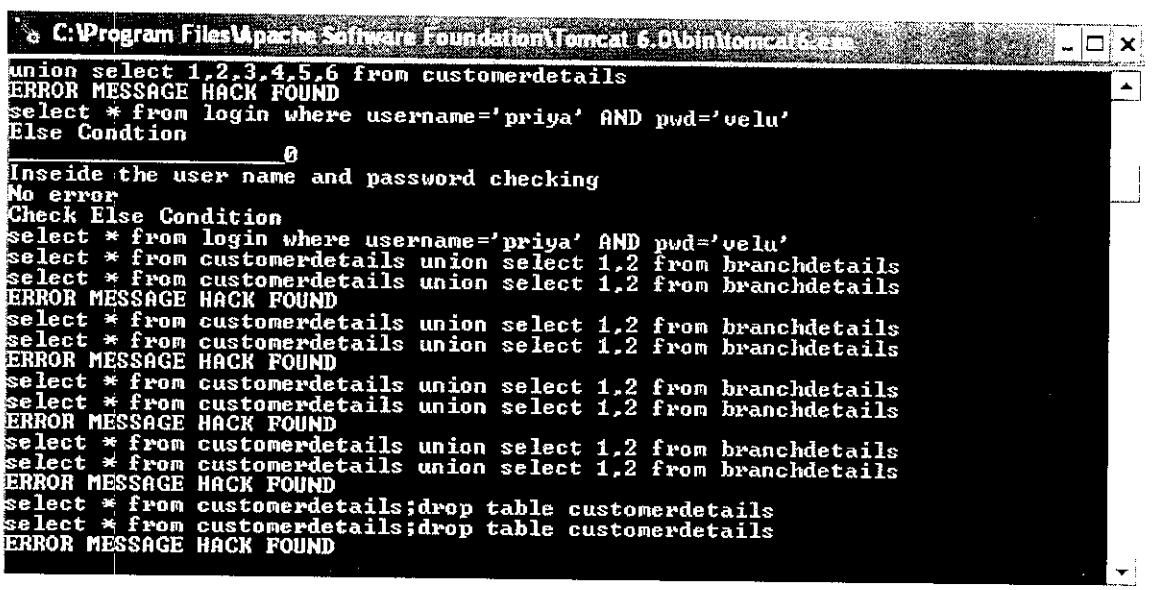
ERROR PAGE: SERVERSIDE



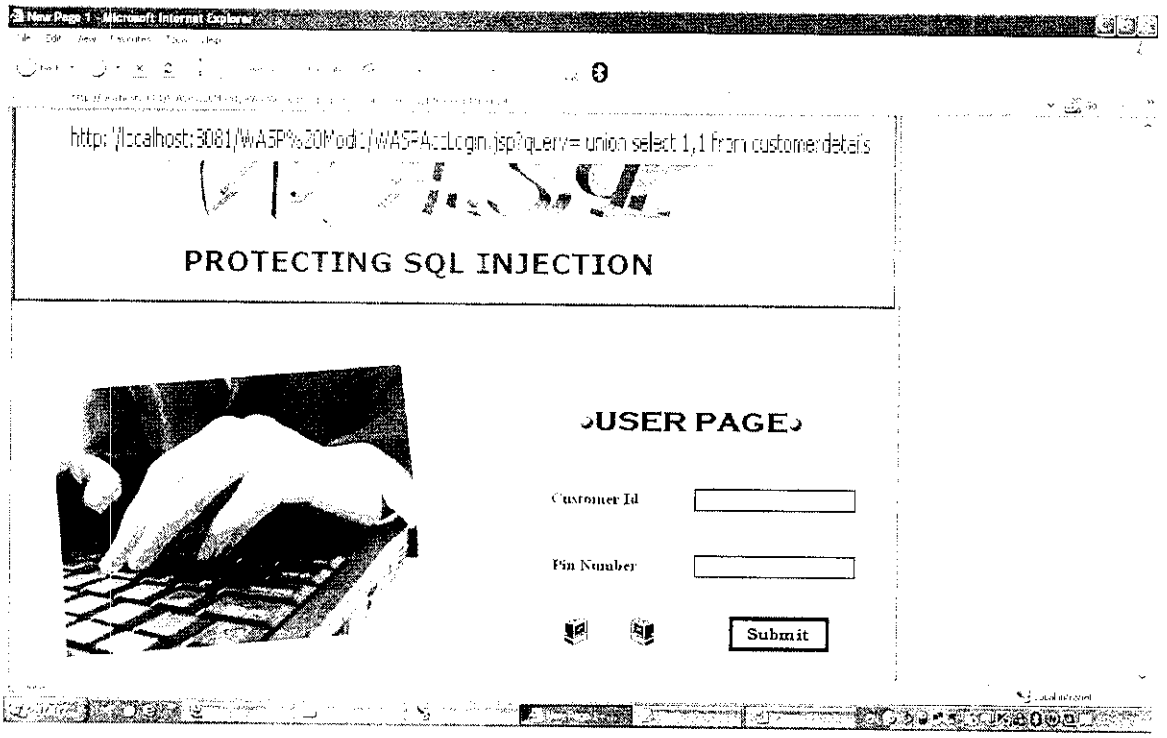
SQL INJECTION PREVENTER AGAINST PIGGYBACKEDQUERIES:



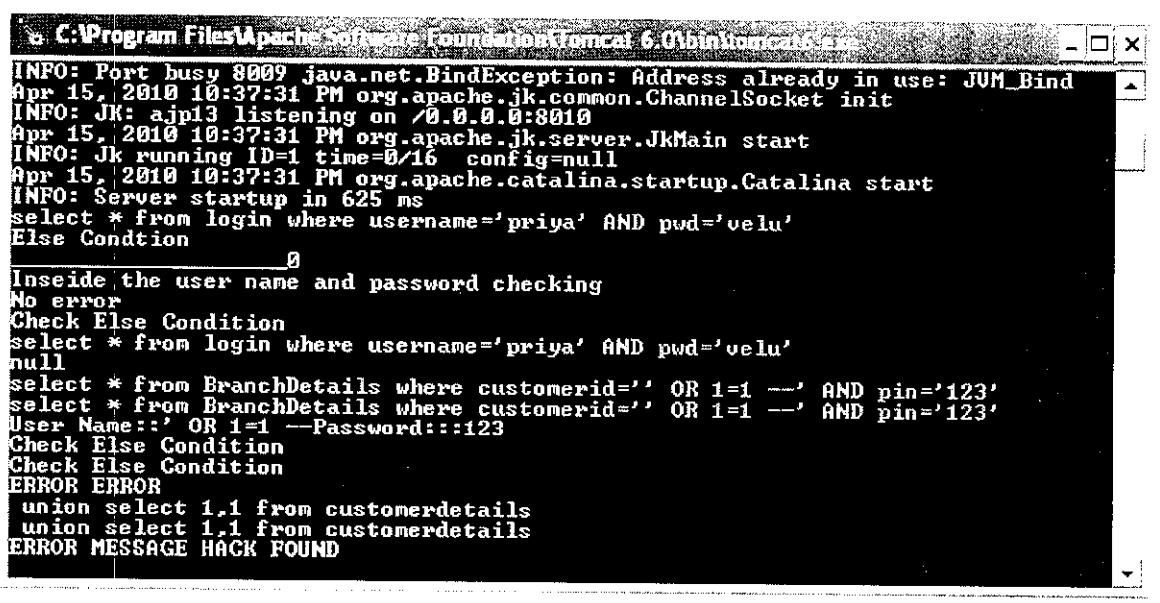
ERROR PAGE: SERVERSIDE



SQL INJECTION PREVENTER AGAINST MALFORMEDQUERIES:



ERROR PAGE: SERVERSIDE



REFERENCES

[1]WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation William G.J. Halfond, Alessandro Orso, Member, IEEE Computer Society, and Panagiotis Manolios, Member, IEEE Computer Society. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 34, NO. 1, JANUARY/FEBRUARY 2008

[2] C. Anley, "Advanced SQL Injection In SQL Server Applications,"white paper, Next Generation Security Software, 2002.

[3] S.W. Boyd and A.D. Keromytis, "SQLrand: Preventing SQL InjectionAttacks," Proc. Second Int'l Conf. Applied Cryptography and Network Security, pp. 292-302, June 2004.

[4] "Top Ten Most Critical Web Application Vulnerabilities," OWASP Foundation, http://www.owasp.org/documentation/topten.html, 2005.

[5]. D. Scott and R. Sharp, "Abstracting Application-Level Web Security," Proc. 11th Int'l Conf. World Wide Web, pp. 396-407,May 2002.

[6]. W. G. Halfond, J. Viegas, and A. Orso. A Classification of SQL-Injection Attacks and Countermeasures. In Proc. of the Intl. Symposium on Secure Software Engineering, Mar. 2006.

[7]. J. Saltzer and M. Schroeder. The Protection of Information in Computer Systems. In Proceedings of the IEEE, Sep 1975.

[8]. G. Wassermann and Z. Su. An Analysis Framework for Security in Web Applications. In Proc. of the FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2004), pages 70--78, Oct. 2004.

[9]. W. Xu, S. Bhatkar, and R. Sekar, "Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks," Proc. 15th Usenix Security Symp., Aug. 2006.

[10]. W. Halfond, A. Orso, and P. Manolios, "Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks,"Proc. ACM SIGSOFT Symp. Foundations of Software Eng., pp. 175- 185, Nov. 2006.

[11]. W.R. Cook and S. Rai, "Safe Query Objects: Statically Typed Objects as Remotely Executable Queries," Proc. 27th Int'l Conf.Software Eng., pp. 97-106, May 2005.

[12]. O. Maor and A. Shulman, "SQL Injection Signatures Evasion,"white paper, Imperva,http://www.imperva.com/application_defense_center/white_papers/sql_injection_signatures_evasion.html, Apr. 2004.

[13]. Anyi Liu , Yi Yuan , Duminda Wijesekera , Angelos Stavrou, SQLProb: a proxy-based architecture towards preventing SQL injection attacks, Proceedings of the 2009 ACM symposium on Applied Computing, March 08-12, 2009, Honolulu, Hawaii

[14]. Michael Emmi , Rupak Majumdar , Koushik Sen, Dynamic test input generation for database applications, Proceedings of the 2007 international symposium on Software testing and analysis, July 09-12, 2007, London, United Kingdom