

## HW 09: Security Testing

### Objectives:

In part 1, conduct SQL injection using WebGoat as vulnerable web application tool to demonstrate security testing of a website. In part 2, evaluate website security using ZAP security testing tool.

### Assignment:

Part 1 - Conduct SQL injection using WebGoat as vulnerable web application tool

Part 2 - Evaluate and generate report of website security using ZAP security testing tool

**Author:** Prateek Singh Chauhan Honor

**Pledge:** "I pledge on my honor that I have not given or received any unauthorized assistance on this assignment/examination. I further pledge that I have not copied any material from a book, article, the Internet, or any other source except where I have expressly cited the source."

## Part 1:

Step Performed for Part 1 of the assignment:

1. Install WebGoat 8.2.2 and pre requisite Java 15
2. Ran command `java -jar webgoat-server-8.2.2.jar`

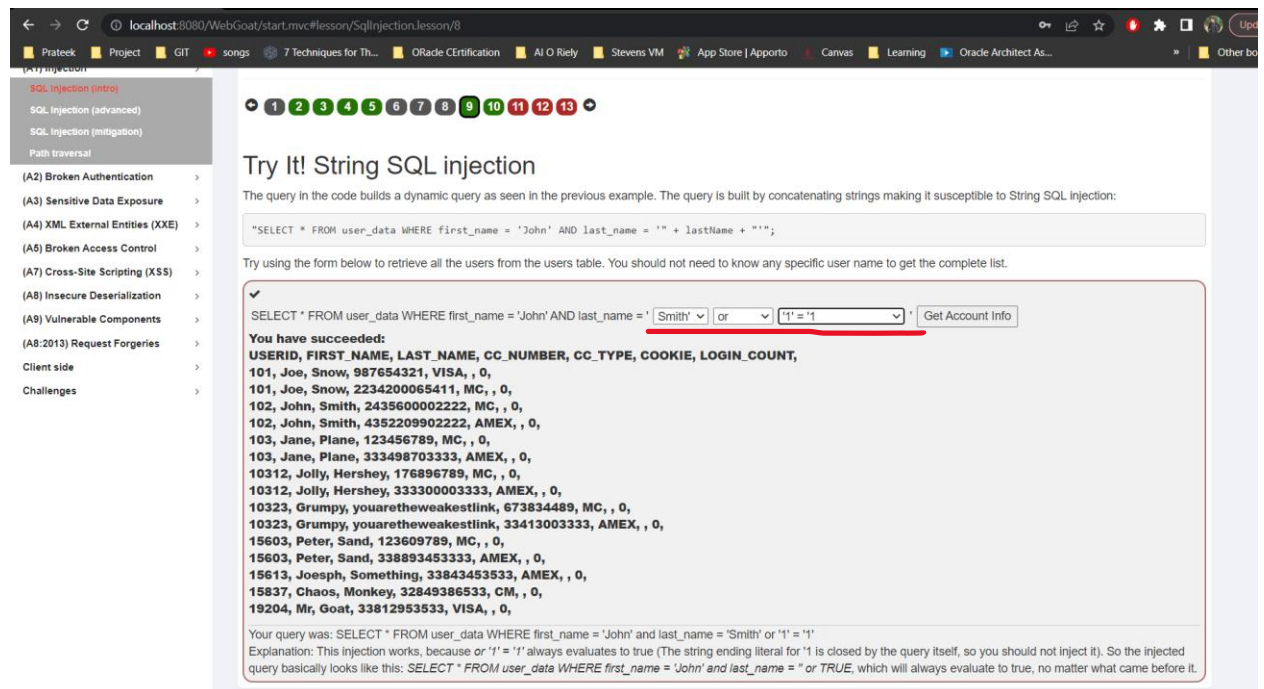
```
2022-11-09 15:05:26.832 INFO 42684 --- [main] org.jboss.threads : JBoss Threads version 3.1.0.Final
2022-11-09 15:05:26.881 INFO 42684 --- [main] o.s.b.w.e.undertow.UndertowWebServer : Undertow started on port(s) 8080 (http) with context path '/WebGoat'
2022-11-09 15:05:26.899 INFO 42684 --- [main] org.owasp.webgoat.StartWebGoat : Started StartWebGoat in 12.889 seconds (JVM running for 18.781)
```

3. In browser open `localhost:8080/WebGoat` and registered new user in WebGoat
4. Performed SQL Injection and SQL injection tutorials.

**"SQL Injection" Challenge: Solve the challenge in steps 7 or 8 of the tutorial:**

What is the string that you entered into the name field to solve this challenge?

- Smith' or '1'='1
- 



What is the Hershey Jolly's AMEX credit card number?

- 333300003333
-

The screenshot shows the 'Try It! String SQL injection' section of WebGoat. The user has entered the payload: `"SELECT * FROM user_data WHERE first_name = 'John' AND last_name = '' + lastName + ''";`. The application has successfully retrieved all users from the `user_data` table. The output shows a list of users with their details, including `10312, Jolly, Hershey, 333300003333, AMEX, , 0`. The explanation states: "Your query was: SELECT \* FROM user\_data WHERE first\_name = 'John' and last\_name = 'Smith' or '1' = '1'. Explanation: This injection works, because or '1' = '1' always evaluates to true (The string ending literal for '1' is closed by the query itself, so you should not inject it). So the injected query basically looks like this: SELECT \* FROM user\_data WHERE first\_name = 'John' and last\_name = '' or TRUE, which will always evaluate to true, no matter what came before it."

### "SQL Injection (Advanced)" Challenge: Solve the challenge in Step 3 of "SQL Injection (Advanced)"

The screenshot shows the '6.a) Retrieve all data from the table' challenge in WebGoat. The user has entered the payload: `user' or '1'='1';Select * from user_system_data;--`. The application has successfully retrieved all data from the `user_system_data` table. The output shows a list of users with their details, including `105, dave, passW0rD, ,`. The explanation states: "Well done! Can you also figure out a solution, by using a UNION? Your query was: SELECT \* FROM user\_data WHERE last\_name = 'user' or '1'='1';Select \* from user\_system\_data;--"

What is the string that you entered into the name field to solve this challenge?

- `user' or '1'='1';Select * from user_system_data;--`

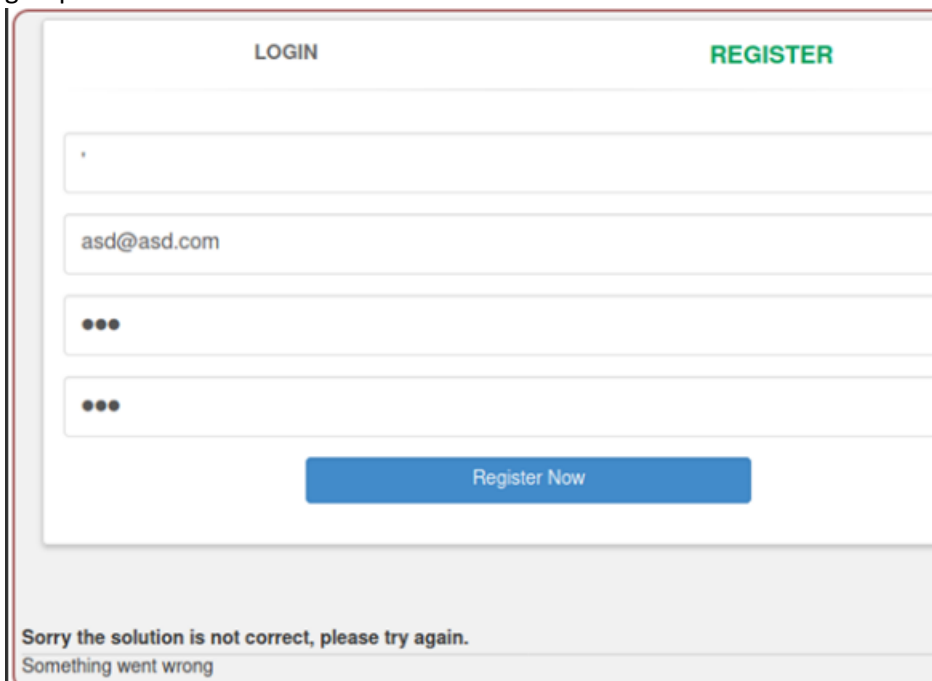
What is password of the user whose user\_name is 'dave'?

- `passW0rD`

## Reflection:

### Describe in your own words how you attacked this "site" to reveal the internal information

- I tried the Login form for a while with simple SQLi attempts first trying the password field as we did know that there is a user named Tom and when that failed, I tried to use the login field and comment out the password altogether. Responses were the same regardless of input and I was running out of ideas to try when I noticed the huge text next to the login. It said REGISTER and clicking it opened a new form.
- From ZAP tool, I understood that Register page is susceptible to SQLi and tried using register page get queries from tables.



- Then I tried using username as asd'—and when I clicked on register it created the user account.

**User asd'-- created, please proceed to the login page.**

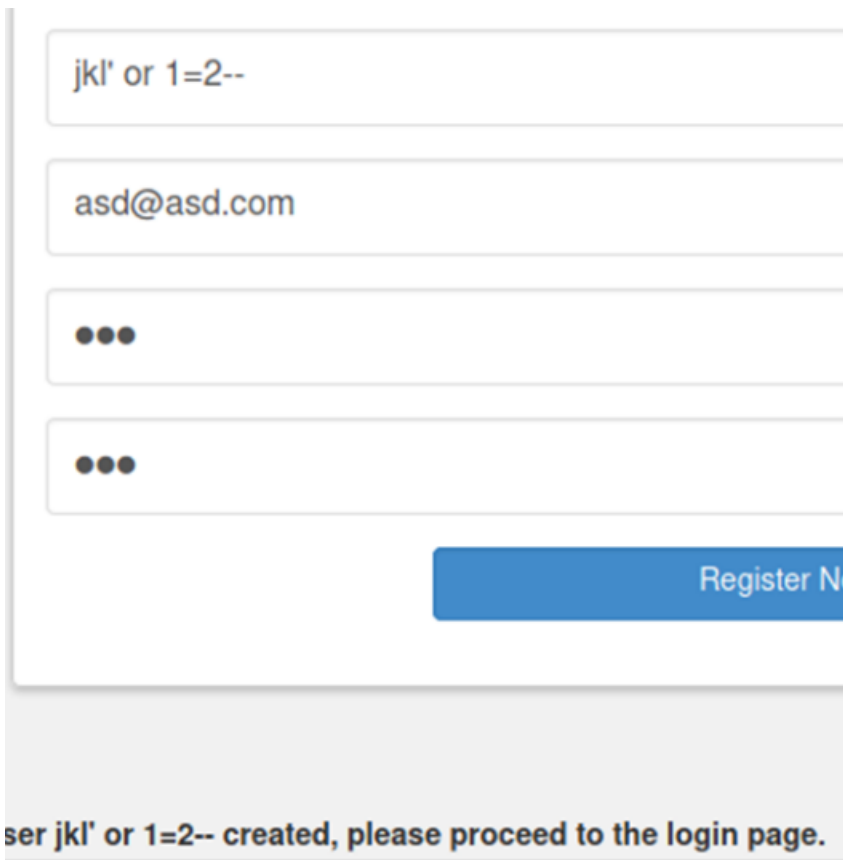
- After using the above username in login, it successfully logged in, meaning the login page works.
- The input has potential SQLi vulnerability as the single quote causes an error, but still the username is asd'-- instead of asd. Usually, services do not allow duplicates in usernames, so I wonder if there is a check for the duplicate. If I try to register again with the previous asd'-- things get weird. It got registered again. But when I try to with asd twice, it gave me an error.

**User asd already exists please try to register with a different username.**

- Giving me an idea that username is susceptible to SQLi. But to find out Tom's password, we need to know the table name.
- For this, I registered with username as asd'; SELECT \* FROM information\_schema.tables; --

- But the above username registration gave output as User already exists. It appears that adding a new user is not vulnerable, but the code that checks if there already exists a user is. Therefore we were able to register an infinite number of users named asd'--.
- I did not see what the query returns so we are dealing with a blind SQLi here. There most likely is no user with the name in the above image suggesting that maybe there is code that gives "user exists"-error we see above if SQL query comes back as not null. It does not matter to us what the code behind is if we get different responses on true and false.

The screenshot shows a web application's registration form. It has four input fields: a username field containing 'jkl' or 1=1--', an email field containing 'asd@asd.asd', and two password fields, each with three dots indicating masked text. A blue 'Register Now' button is positioned to the right of the password fields. Below the form, a message in red text states: 'User jkl' or 1=1-- already exists please try to register with a different username.'



The screenshot shows a web registration form with four input fields and a button. The first field contains the text "jkl' or 1=2--". The second field contains "asd@asd.com". The third and fourth fields are masked with three dots. The button is blue and labeled "Register Now". Below the form, a message reads: "User jkl' or 1=2-- created, please proceed to the login page."

- Confirming that the website only returns true, false or error as output so expecting a DB output would be wrong.
- Since we know that results are not going to be printed in output, to make things easier, I started Burp so I can intercept the request and send it to repeater.

```
username_reg=asd'+and+username%3D'asd&
```

- From this, I was trying to see whether there is a column named username containing "asd". The last and first quote comes from the code.

```
{  
  "lessonCompleted":false,  
  "feedback":"Sorry the solution is not correct, please try again.",  
  "output":"Something went wrong"  
}
```

- With this output, I was confirmed that there is no column name username.
- But I tried different column names and was successful with password:

```
username_reg=asd'+and+password%3D'asd&email_reg=asd%40asd.com&password_reg=asd&  
confirm_password_reg=asd
```

```
{  
  "lessonCompleted":false,  
  "feedback":"User asd' and password='asd already exists please try to register with a different username.",  
  "output":null  
}
```

- Meaning that there is a column named password. Then it gave me an idea that based on the input given I can hack by brute force using substring with column password by passing each alphabet one by one.
- Like this: `asd' AND substring(password,1,1)='a`
- Which gave me result as true
- `"feedback": "User asd' AND substring(password,1,1)='a already exists please try to register with a different usernar`
- So I kept on trying one by one and was able to get the complete password in 12 tries.
- That's how I hacked the Web Goat

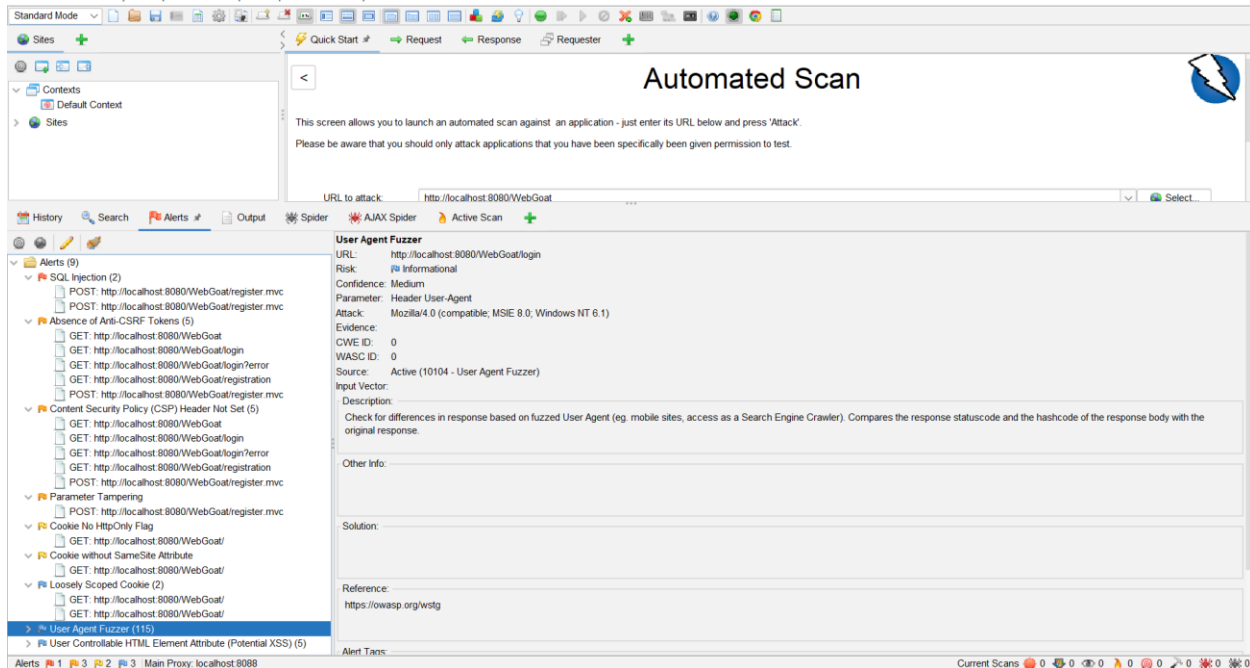
**What would you do to prevent this attack in your websites?**

- By using Parameterized Queries.
- A parameterized query is a query in which placeholders are used for parameters and the parameter values are supplied at execution time.
- The most important reason to use parameterized queries is to avoid SQL injection attacks

## Part 2:

Followed all the steps mentioned in the problem statement.

Screen dump of the automated scan:



High Risk Alerts:

- SQL Injection (2 Instances)
  - o Instance 1: SQL injection in the register page using Boolean expression in request body
    - Request Body:

**username=ZAP&password=ZAP&matchingPassword=ZAP&agree=agree+AND+1%3D1+--+**

**SQL Injection**

URL: http://localhost:8080/WebGoat/register.mvc

Risk: High

Confidence: Medium

Parameter: agree

Attack: agree AND 1=1 --

Evidence:

CWE ID: 89

WASC ID: 19

Source: Active (40018 - SQL Injection)

Input Vector: Form Query

Description: SQL injection may be possible

Other Info: The page results were successfully manipulated using the boolean conditions [agree AND 1=1 --] and [agree AND 1=2 --]. Data was returned for the original parameter. The vulnerability was detected by successfully restricting the data originally returned, by manipulating the parameter

Solution: Do not trust client side input, even if there is client side validation in place. In general, type check all data on the server side. If the application uses JDBC, use PreparedStatement or CallableStatement, with parameters passed by '?'




- Instance 2: SQL injection in the register page using Boolean expression in request body
  - Request Body:

username=ZAP&password=ZAP&matchingPassword=ZAP&agree=agree%27+AND+%271%27%3D%271%27+- --+

#### SQL Injection

URL: http://localhost:8080/WebGoat/register.mvc

Risk:  High

Confidence: Medium

Parameter: agree

Attack: agree' AND '1'='1' --

Evidence:

CWE ID: 89

WASC ID: 19

Source: Active (40018 - SQL Injection)

Input Vector: Form Query

#### Description:

SQL injection may be possible

#### Other Info:

The page results were successfully manipulated using the boolean conditions [agree' AND '1'='1' -- ] and [agree' AND '1'='2' -- ]

Data was returned for the original parameter.

The vulnerability was detected by successfully restricting the data originally returned, by manipulating the parameter

#### Solution:

Do not trust client side input, even if there is client side validation in place.

In general, type check all data on the server side.

If the application uses JDBC, use PreparedStatement or CallableStatement, with parameters passed by '?'

#### Reflection:

- Learned how to check the vulnerabilities using ZAP. Also using ZAP reports, I was able to hack Web Goat using SQLi.
- Using ZAP, one can find the vulnerabilities of the website and can fix them before the application goes to production.
- The security tool provides the report in ranking form. Ranking from High to low, based on which the teams can prioritize the fix and prevent security holes in their system.