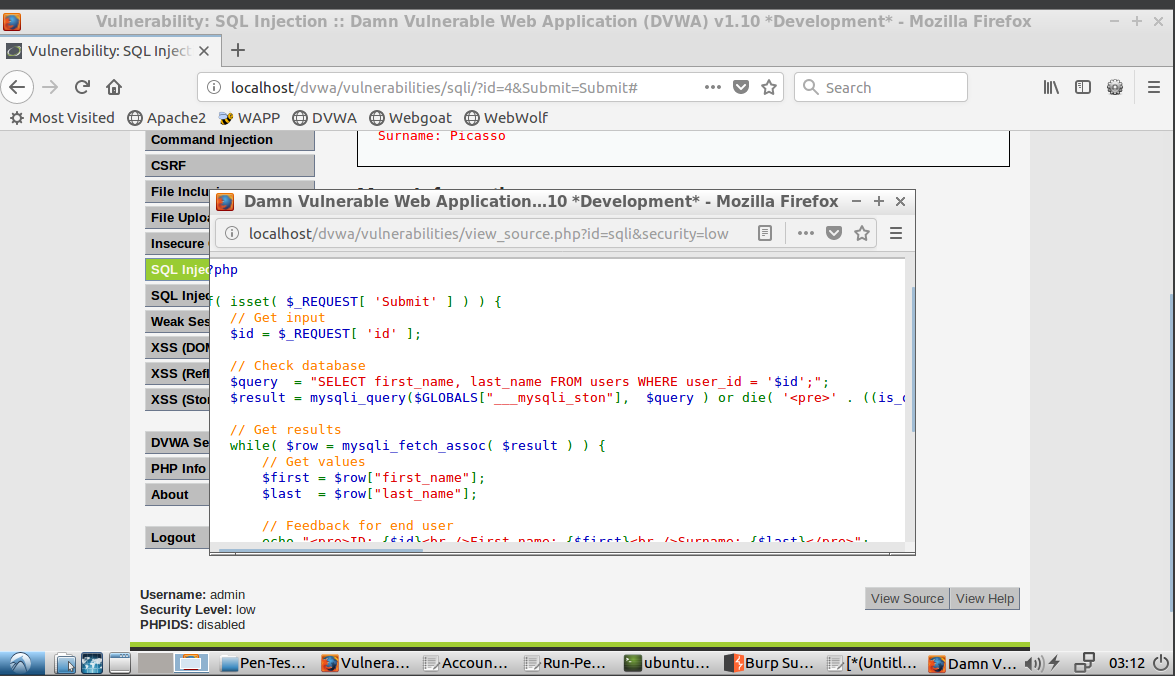
# Group Project 3

(Group 1)

Mario Amaro  
Sushruti Bansod  
Joshua Fishman  
Nick Forleo

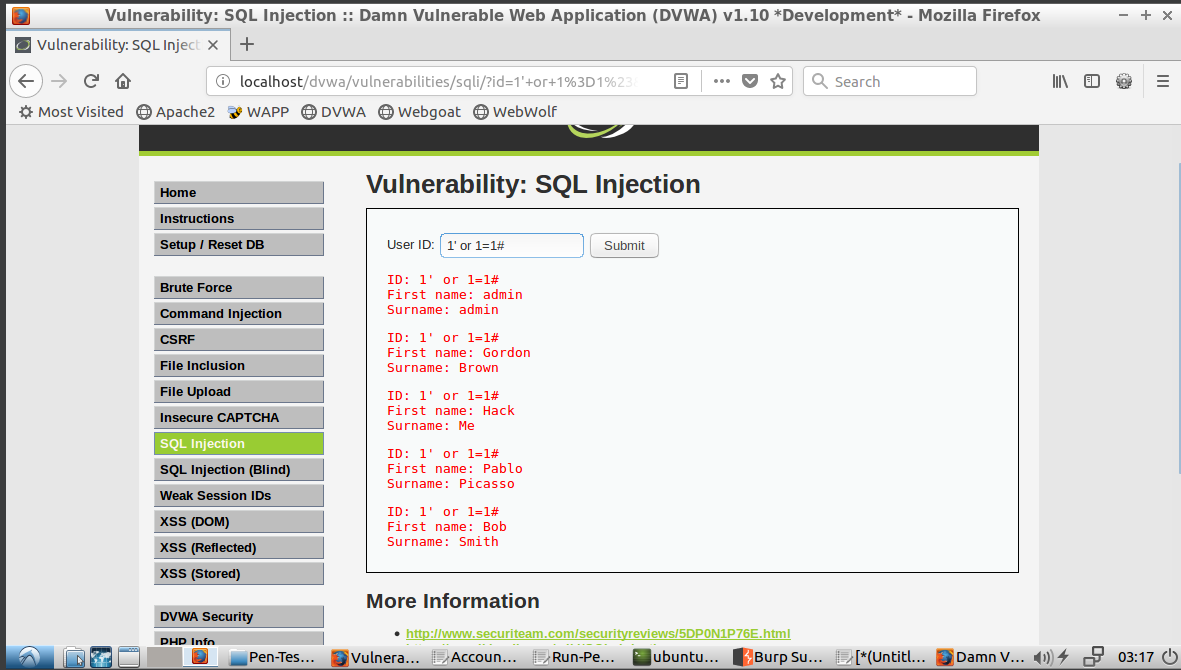
**Task 1:** Show how you can compromise the target application using a SQL injection

For this task, we will attack DVWA. DVWA outputs information onto the web page that can be used to exploit their system, so using a toolkit isn’t mandatory. However, we used BurpSuite to verify that no information was hidden. The first thing we noticed was that we have access to the query being executed. This is shown in Figure 1.1

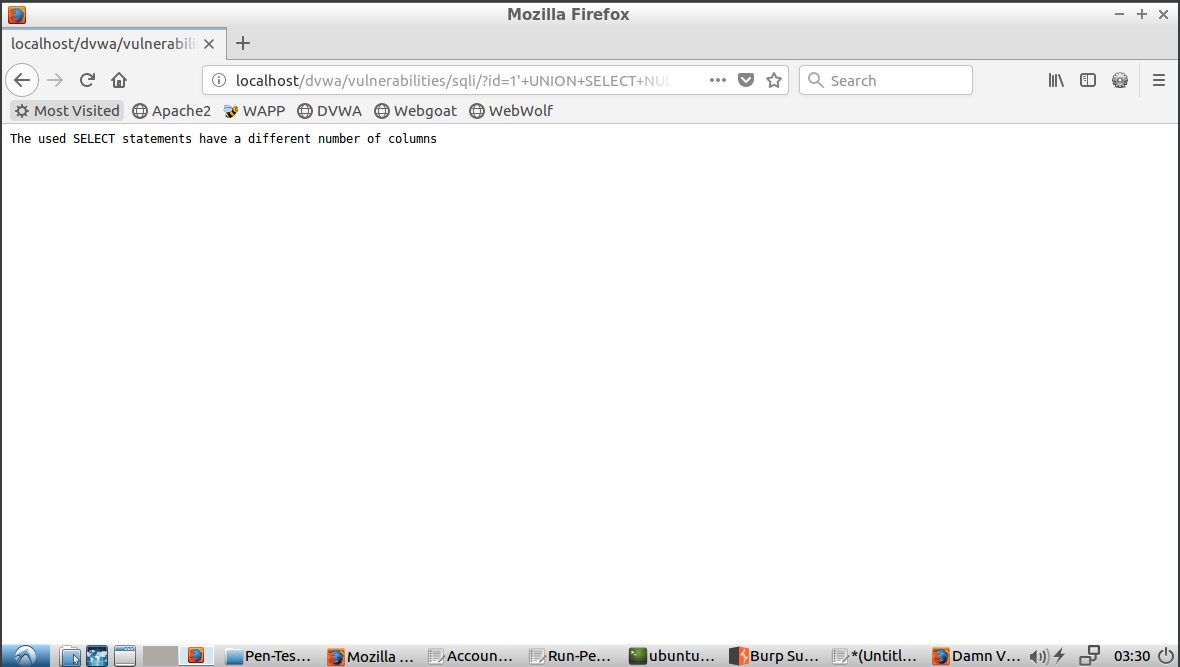


*Figure 1.1 - Source code of database query*

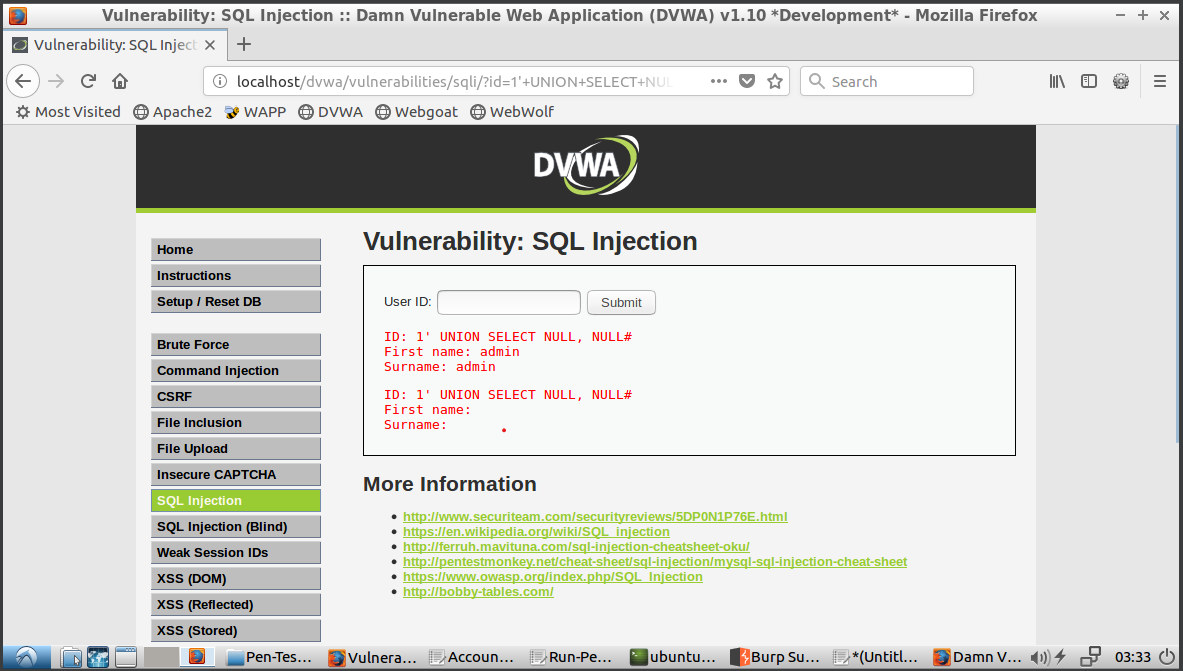
Based on this screenshot, we know some information. The important pieces to note are that the name of the table is `users` and that the user input is not validated. This gives us a head start to exploit this page. First, to verify that the user input is truly unvalidated, we can enter a SQL boolean value and check if it returns extra data. To do this, we entered `1’ or 1=1#`. The `1’` will end the trailing the quote while the `or 1=1#` is the boolean that evaluates to True. This means we should be able to get all results from the table. This is shown below in Figure 1.2

*Figure 1.2 - Query entered into form to return extra data*

Now that we see that the user input is not validated, we can begin to craft a query to extract data that should be hidden from the database. We can do this by doing a UNION attack. This could help us get data from other tables, if necessary. In our case will help determine the number of columns needed for a second select statement[1]. To do this, we can start by entering `1’ UNION SELECT NULL#`. This will try to union our provided query with the query executed by the application. For this to be done, the queries must have the same number of columns. When our input is submitted, the web page returns `The used SELECT statements have a different number of columns`, as seen in Figure 1.3.

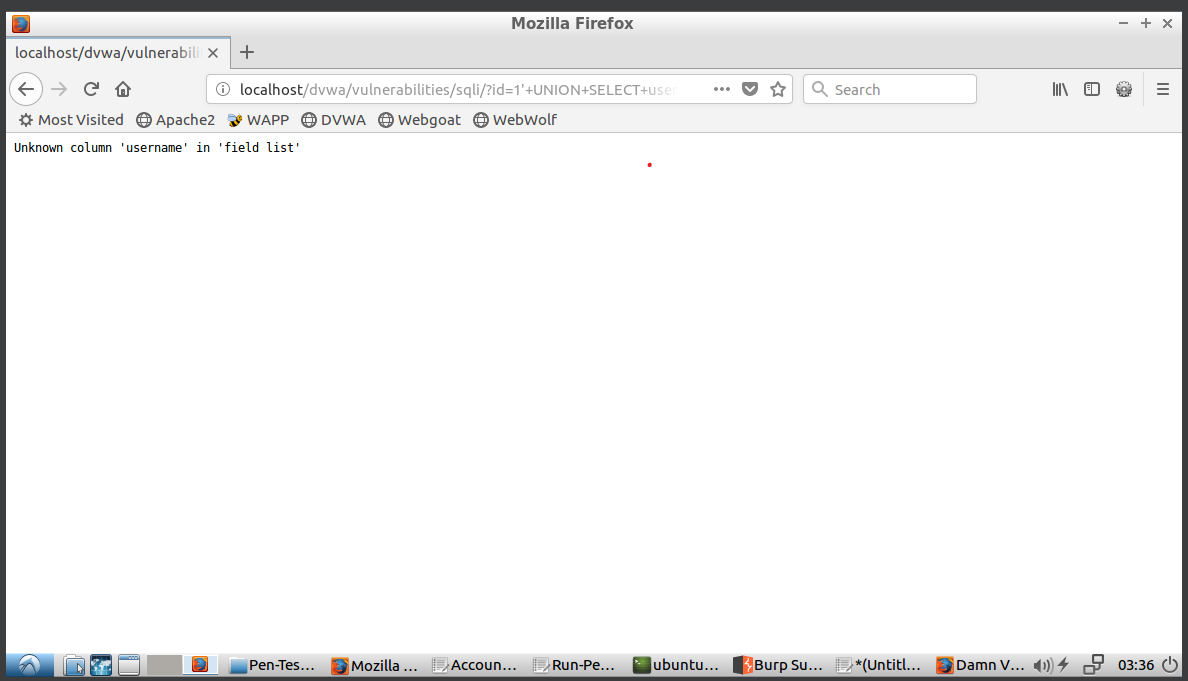
*Figure 1.3 - Results of executed query with incorrect number of columns*

Now we can increase the number of times that NULL is repeated in the query until we find out how many columns are necessary. So, the second time we enter `1’ UNION SELECT NULL, NULL#`. This time, the query is executed without errors and returns some data (see Figure 1.4). We now know that we are looking for two columns.



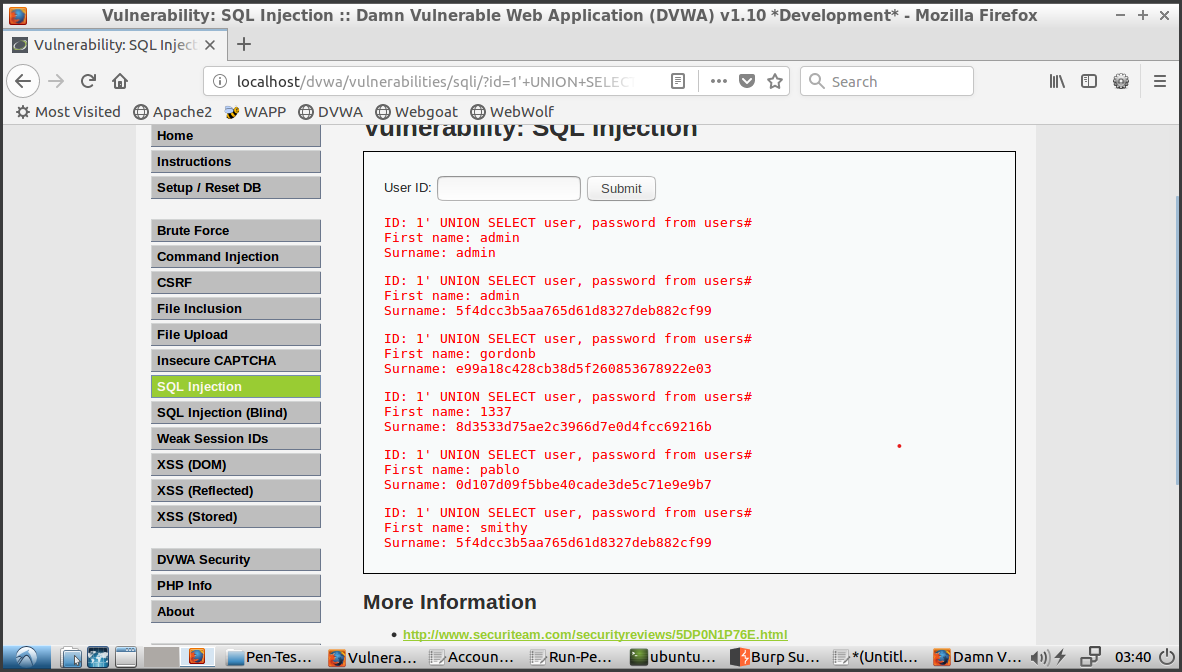
*Figure 1.4 -* `1’ UNION SELECT NULL, NULL#` results

Recall from the beginning that we know the table is called `users`. We can use this with some common attribute names to attempt to extract data from the database. For example, we can enter `1’ UNION SELECT username, password from users#`. In our case, this failed with the message `Unknown column ‘username’ in ‘field list’`, as seen in Figure 1.5.



*Figure 1.5 -* `1’ UNION SELECT username, password from users#` error message

Now we know that username is incorrect. We will change this to `user` instead. Finally, we see that the SQL command was executed successfully and that the user names and password were output onto the screen (Figure 1.6). Most of the passwords are encrypted, but it seems to be fairly basic and with a little bit more work, can be decrypted into plain text. We were able to successfully expose protected data from the database.



*Figure 1.6 - Successfully exposed protected data from database*

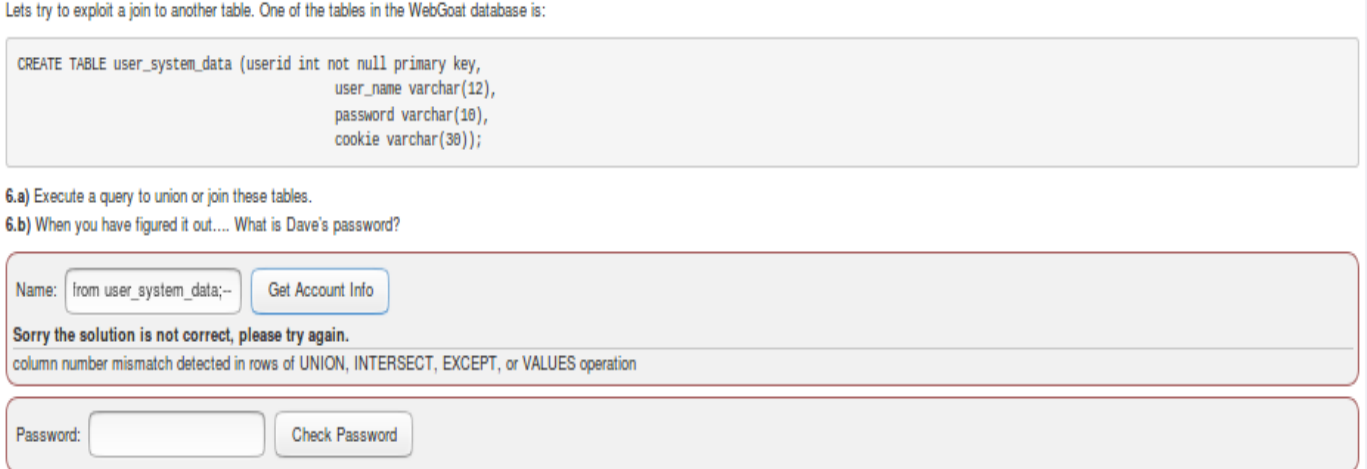
**Task 2:** Show how you can compromise its back-end database using a SQL injection

For this scenario we attacked Webgoat using SQL Injection that would comprise the database. We needed to pull data to get us Daves password. We know for a fact that one user has a data table with the following attributes: userid(PK), user\_name, password, and cookie.



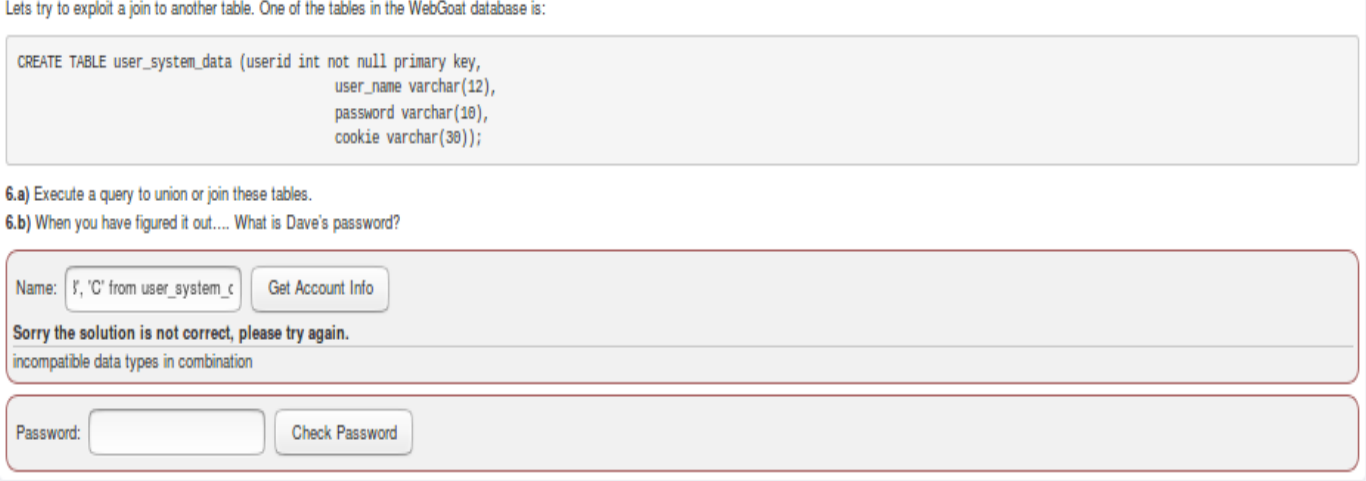
*Figure 2.1 – The login page*

We decided to use an SQL inject via a UNION statement that will look to overlap identical tables to get us the information we needed. Since we knew that we have userid that could not be null, user\_name, password, and cookie, we decided to use that to start the statement | ‘ UNION SELECT 1, user\_name, password, cookie from user\_system\_data;-- | where user\_system\_data is the location of the users data and “—” is the end comment. We received the error columns number mismatch detected in rows UNION, INTERSECT, EXCEPT, or VALUES operation. For a UNION the columns between all the tables have the match, so we were missing some columns. The next statement we tried was adding an extra varchar column ‘A’ to the statement | ‘ UNION SELECT 1, user\_name, password, cookie, ‘A’ from user\_system\_data;-- |. Again this came up with the same error and the same when we added another column ‘B’.



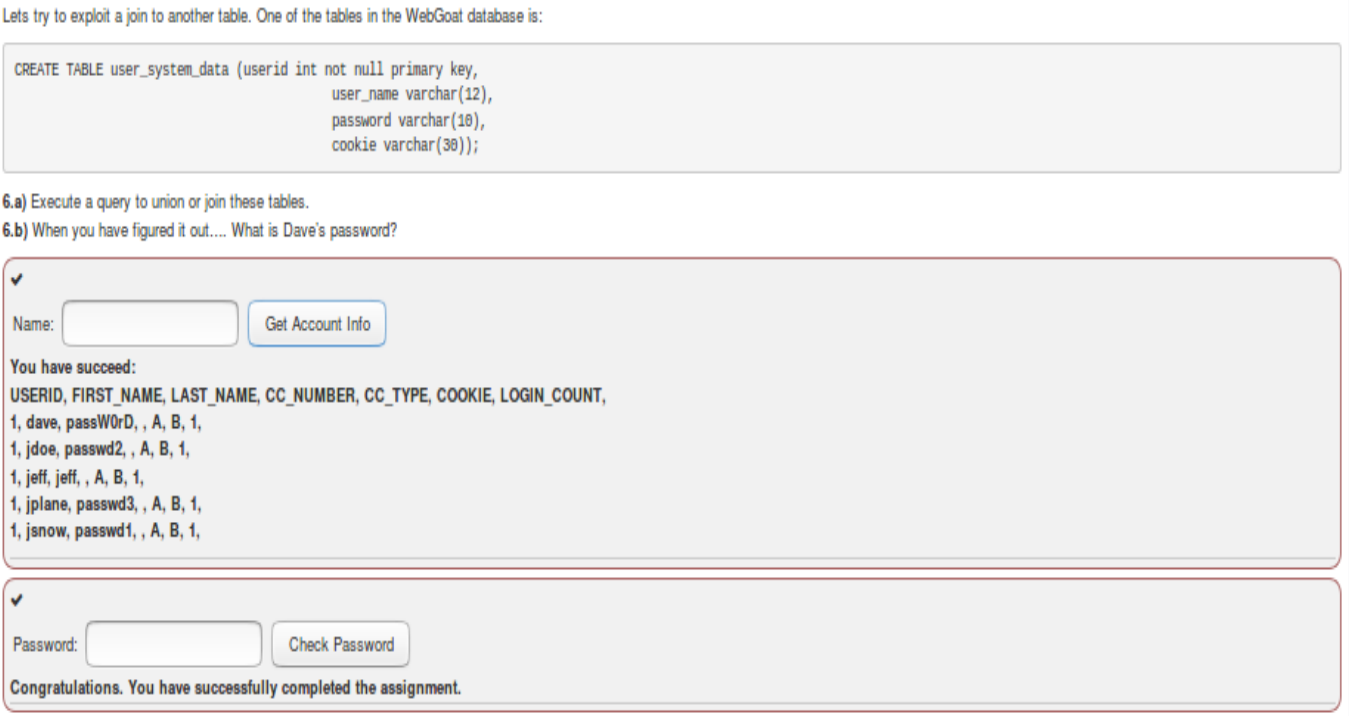
*Figure 1.2 – First couple of attempts for SQL Injection*

When the statement then became | ‘ UNION SELECT 1, user\_name, password, cookie, ‘A’, ‘B’, ‘C’ from user\_system\_data;-- |, a new error arrived stating incompatible data types in combination. So now we knew that the column for ‘C’ was not a varchar.



*Figure 1.3 – 7th column not a varchar*

We then decided to change ‘C’ into an integer ‘1’. The following statement was used. | ‘ UNION SELECT 1, user\_name, password, cookie, ‘A’, ‘B’, 1 from user\_system\_data;-- | And success, we were able to UNION the data tables from the database and gather the information we needed.



*Figure 1.6 - Successful UNION*

**Task 3:** Based on the found vulnerable issues, what should you do to prevent attackers from hacking your web application using a SQL injection?

To prevent the SQL injection attacks described in the scenario, we can implement several security measures:

1. **Input Validation**: Validate and sanitize all user inputs on the server-side before using them in SQL queries. Use whitelisting to allow only expected input formats and reject everything else.
2. **Parameterized Queries**: Use parameterized queries or prepared statements in the database queries. Parameterized queries separate SQL code from user data, preventing attackers from injecting malicious SQL commands.
3. **Error Handling**: Implement proper error handling and custom error messages. Do not expose detailed error messages, such as SQL syntax errors, to users. Log these errors securely for internal review and debugging.
4. **Database Firewall**: Consider using a database firewall or intrusion detection/prevention system (IDPS) that can detect and block suspicious SQL injection attempts.
5. **Regular Security Audits**: Conduct regular security audits and vulnerability assessments of the web applications and databases. Use tools like BurpSuite or OWASP ZAP to identify and mitigate potential security weaknesses.
6. **Use database account with least privileged access:** In these scenarios, the application retrieved data from other unrelated tables in the database. If the application can be designed so that it doesn’t need access to that table, permissions to that table should be removed from the database account to ensure least privileged access.
7. **Use external IAM (Identity and Access Management) service:** These applications stored username and password combinations in the same database as the as application data, so a SQL injection vulnerability of the application became a SQL injection vulnerability of the access control service, which is usually a more critical issue. Using an industry standard IAM service that is logically from the application (i.e. utilizing a different database) will prevent credentials from being stolen due to an SQL injection attack on the application.

By implementing these preventive measures, the risk of SQL injection attacks can be significantly reduced.

**Team peer-review table**

|  |  |
| --- | --- |
| Name | Contributing Efforts in this project (0 ~ 100%) |
| Nick | 100% |
| Mario | 100% |
| Sush | 100% |
| Josh | 100% |

**References:**

1. <https://portswigger.net/web-security/sql-injection/union-attacks>
2. <https://courses.worldcampus.psu.edu/canvas/sp24/22411--12724/content/09_lesson/printlesson.html>
3. <https://aws.amazon.com/iam/>
4. <https://www.keycloak.org/>
5. <https://learn.microsoft.com/en-us/windows-server/identity/identity-and-access>