

Assignment 3 - A

SQL Injection

Atharva Vaidya
121942024

Contents

| | |
|---|---|
| What is difference between DBMS and RDBMS? | 2 |
| Before breaking in to the site for access, an attacker needs to know which SQL engine is at the back end, list out what are the ways to know which SQL server is on work? | 3 |
| What are some examples of web applications that connect to a database server to access data? | 4 |
| What is the impact of SQL injection vulnerabilities? | 5 |
| What is the main reason for the existence of SQL injection vulnerability? | 6 |
| SQL Injection Demo using WebGoat | 7 |

What is difference between DBMS and RDBMS?

| Parameter | DBMS | RDBMS |
|-----------------------|---|--|
| Storage | DBMS Stores data as a file. | Data is stored in the form of tables. |
| Database structure | DBMS system, stores data in either a navigational or hierarchical form. | RDBMS uses a tabular structure where the headers are the column names, and the rows contain corresponding values |
| Number of Users | DBMS supports single user only. | It supports multiple users. |
| ACID | In a regular database, the data may not be stored following the ACID model. This can develop inconsistencies in the database. | Relational databases are harder to construct, but they are consistent and well structured. They obey ACID (Atomicity, Consistency, Isolation, Durability). |
| Type of program | It is the program for managing the databases on the computer networks and the system hard disks. | It is the database systems which are used for maintaining the relationships among the tables. |
| Integrity constraints | DBMS does not support the integrity constants. The integrity constants are not imposed at the file level. | RDBMS supports the integrity constraints at the schema level. Values beyond a defined range cannot be stored into the particular RDMS column. |
| Normalization | DBMS does not support Normalization | RDBMS can be Normalized. |
| Distributed Databases | DBMS does not support distributed database. | RBMS offers support for distributed databases. |
| Ideally suited for | DBMS system mainly deals with small quantity of data. | RDMS is designed to handle a large amount of data. |
| Data Redundancy | Data redundancy is common in this model. | Keys and indexes do not allow Data redundancy. |
| Examples | Examples of DBMS are a file system, XML, Windows Registry, etc. | Example of RDBMS is MySQL, Oracle, SQL Server, etc. |

Before breaking in to the site for access, an attacker needs to know which SQL engine is at the back end, list out what are the ways to know which SQL server is on work?

If your site has web page extensions like .asp, .aspx etc. then there is a high chance of SQL Server or MS Access.

If the pages end as .jsp it could be an Oracle system at your end.

If the pages are ending as .php, it will probably be MySQL.

To identify the back end SQL server the minimum qualification required is beginner level experience with SQL. The general tool used to find out the SQL engine at the back end are the string concatenating characters and the comment characters.

Using the character used to end a SQL statement is also a good candidate for finding out the SQL engine at the backend.

We can not stop this kind of attacks because even showing an error page back to an attacker is like letting him know that the entered string was wrong. However the golden thumb of rule is in the error page don't show the full detailed error message back, instead just say some error has occurred, and you can thus be saved from further damage the attacker is going to cause. Because, SQL injection attacks generally use the returned error messages to go a bit deep into the attacking process.

What are some examples of web applications that connect to a database server to access data?

- Pandora
- Hulu
- Meebo
- Cooliris
- Facebook
- Youconvertit
- Peepel
- Jott
- Netvibes
- TiddlyWiki

What is the impact of SQL injection vulnerabilities?

With no mitigating controls, SQL injection can leave the application at a high-risk of compromise resulting in an impact to the confidentiality, and integrity of data as well as authentication and authorization aspects of the application. An adversary can steal sensitive information stored in databases used by vulnerable programs or applications such as user credentials, trade secrets, or transaction records.

SQL injection vulnerabilities should never be left open; they must be fixed in all circumstances. If the authentication or authorization aspects of an application is affected an attacker may be able login as any other user, such as an administrator which elevates their privileges.

What is the main reason for the existence of SQL injection vulnerability?

- **Design of SQL** : It is designed to allow people to access information and is therefore vulnerable, so every developer should know about SQL Injection.
- **Every single Input Counts** : A single input validation, if remained, may lead to loss of data.
- **Inexperienced Developers** : New Developers lack proper training for designing secure Applications.
- **Priority** : Organizations do not give priority for securing such vulnerabilities.
- **Time** : A lot of time is needed to search vulnerabilities like SQL Injection. Most organizations do not invest in such required amount of time.
- **Money** : A lot of capital is needed to invest in Pen-Testers for finding out vulnerabilities.

SQL Injection Demo using WebGoat

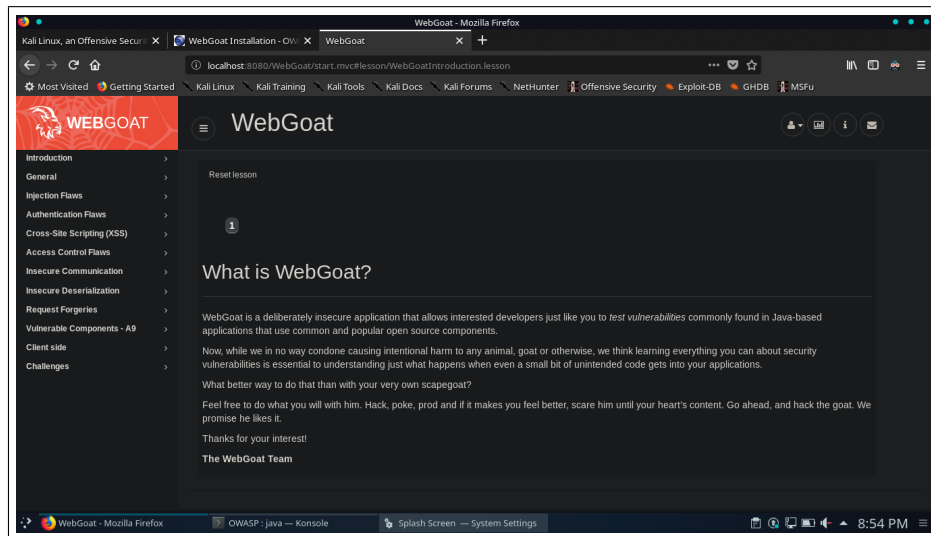


Figure 1: Introduction to WebGoat

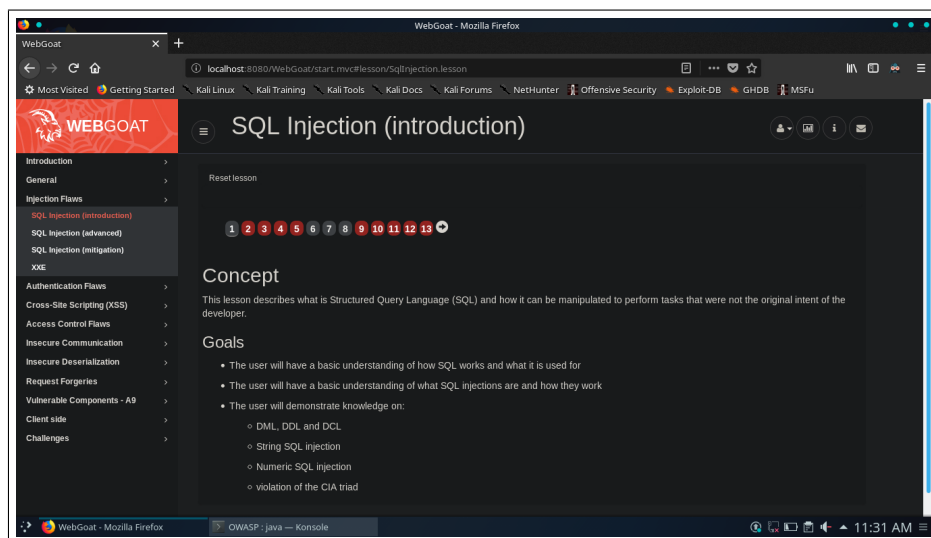


Figure 2: SQL Injection Introduction, Concepts and Goals

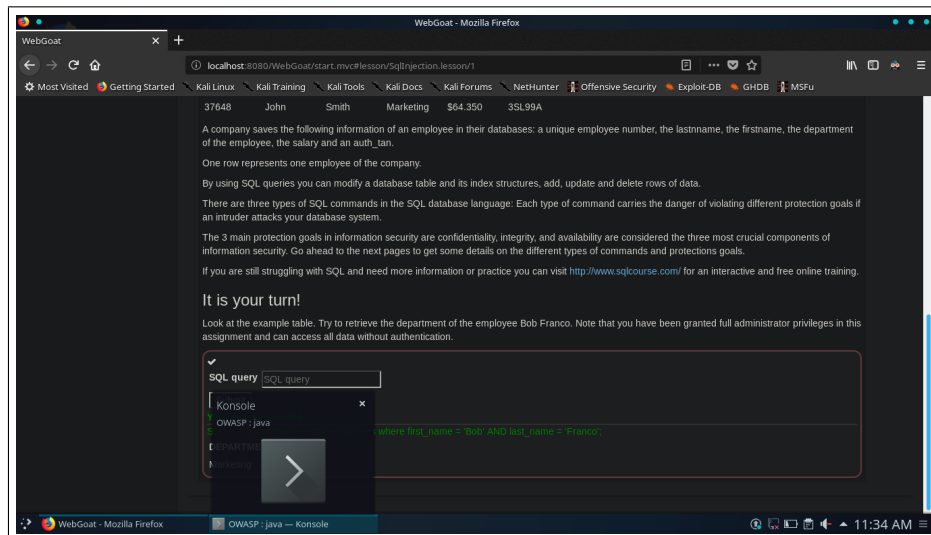


Figure 3: Basic SQL Select Query

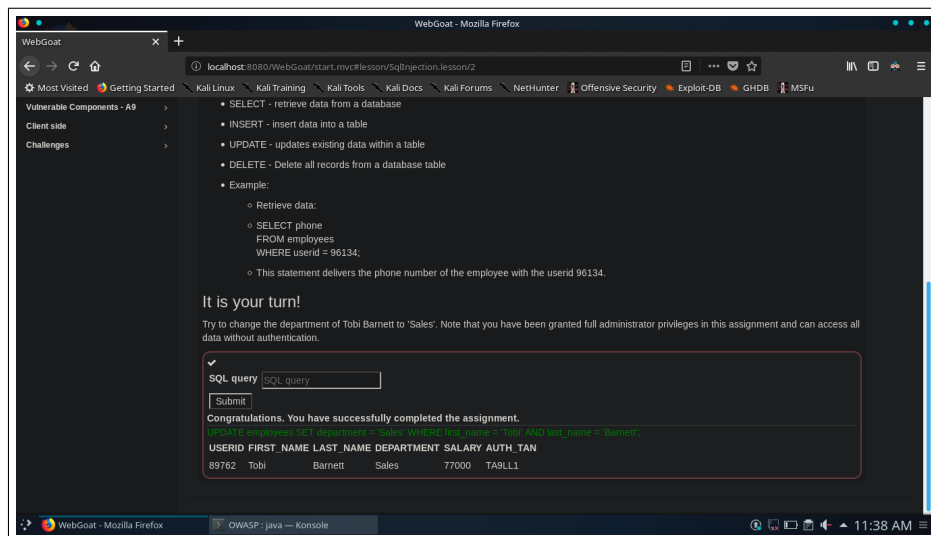


Figure 4: Basic SQL Update Query

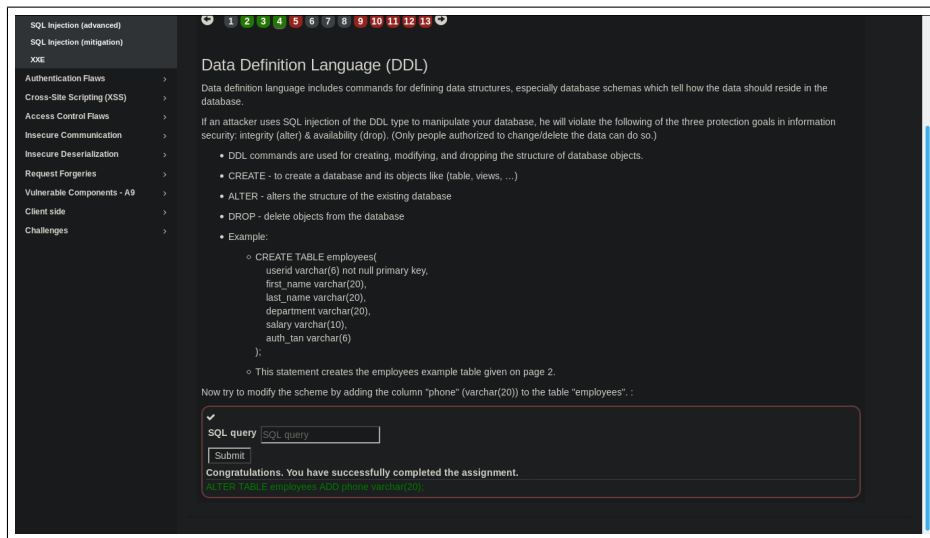


Figure 5: Basic DDL Query

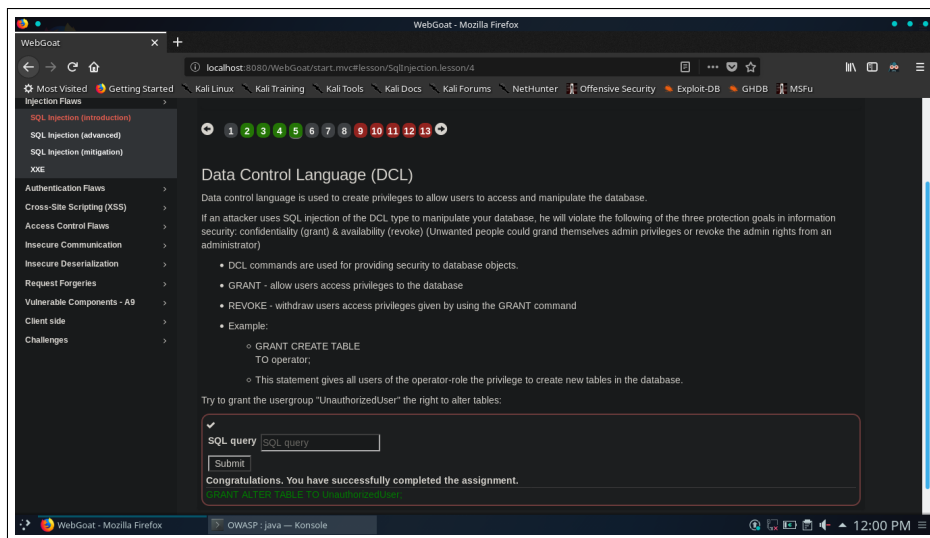


Figure 6: Basic DCL Query

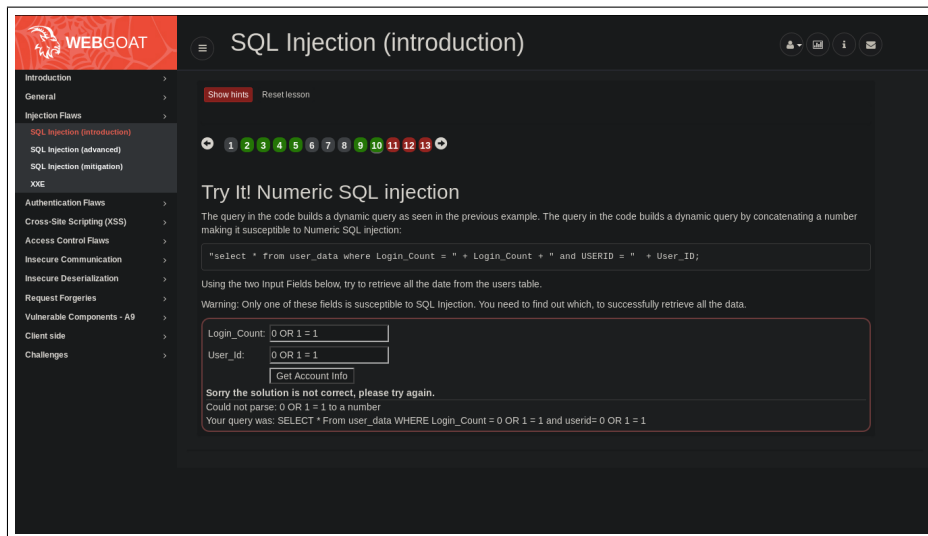


Figure 9: The Login-Count can only accept Numeric Values.

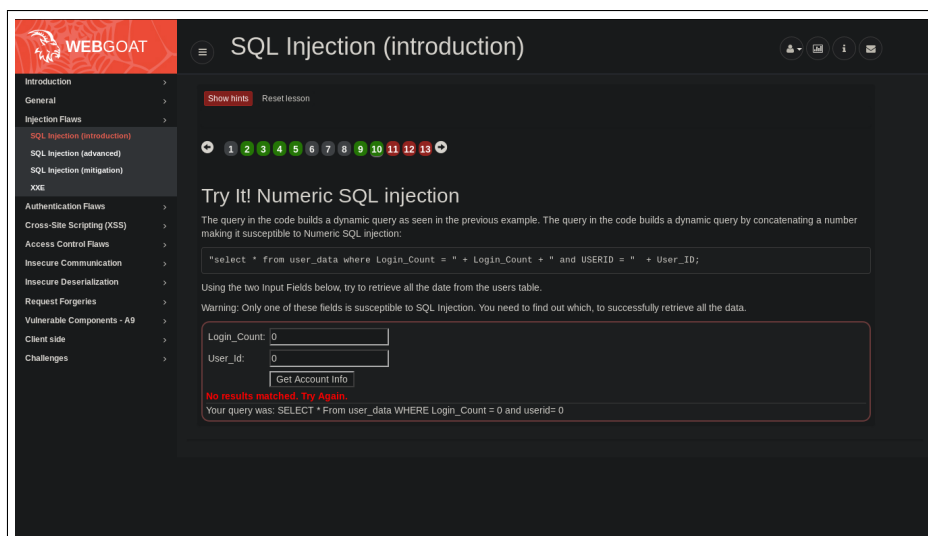


Figure 10: Checking the error message.

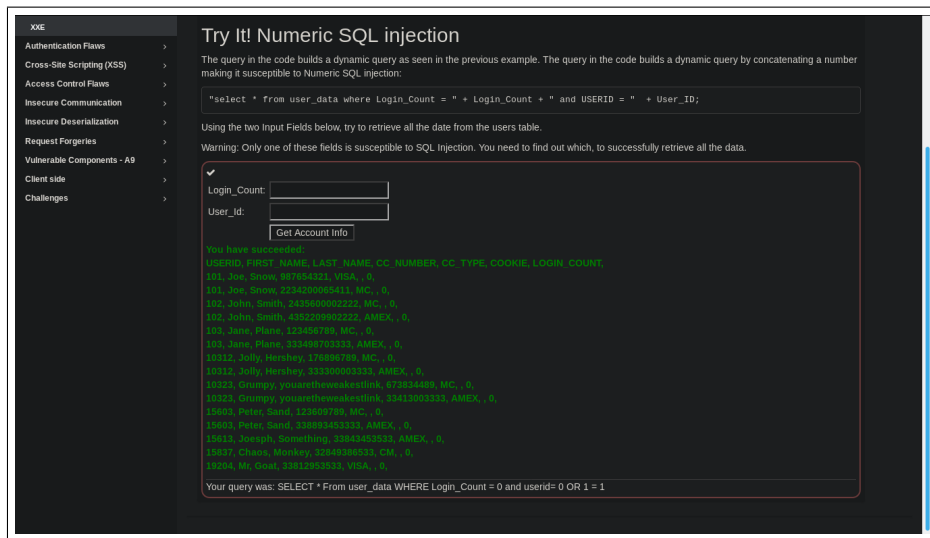


Figure 11: Identified the vulnerable field and applied the appropriate SQL Injection Query.

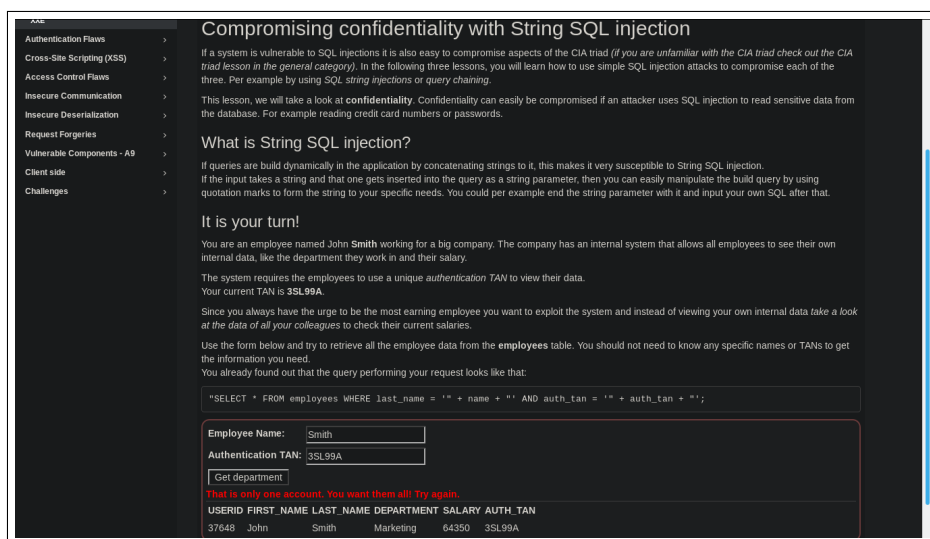


Figure 12: Applied the Query with known values.

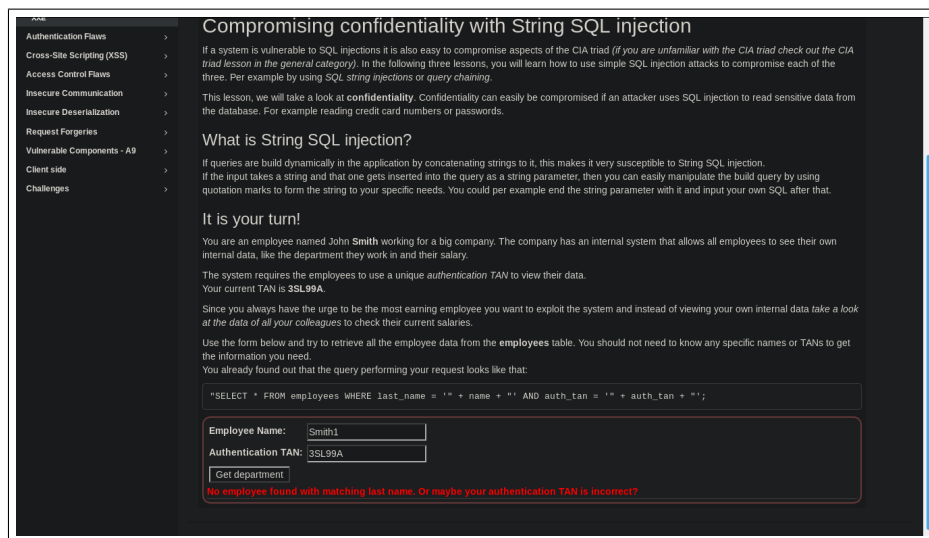


Figure 13: Checking the error message by feeding wrong username and password.

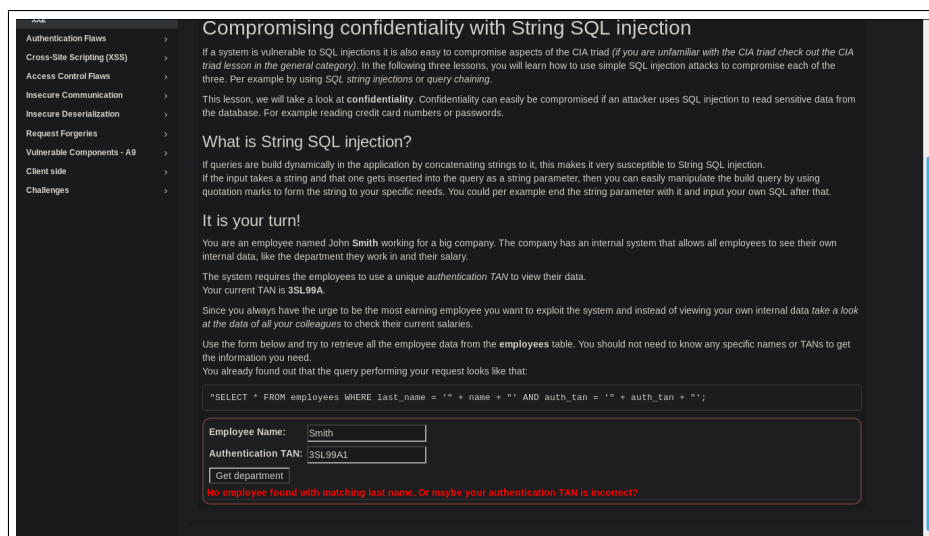


Figure 14: Checking the error message by feeding wrong password only.

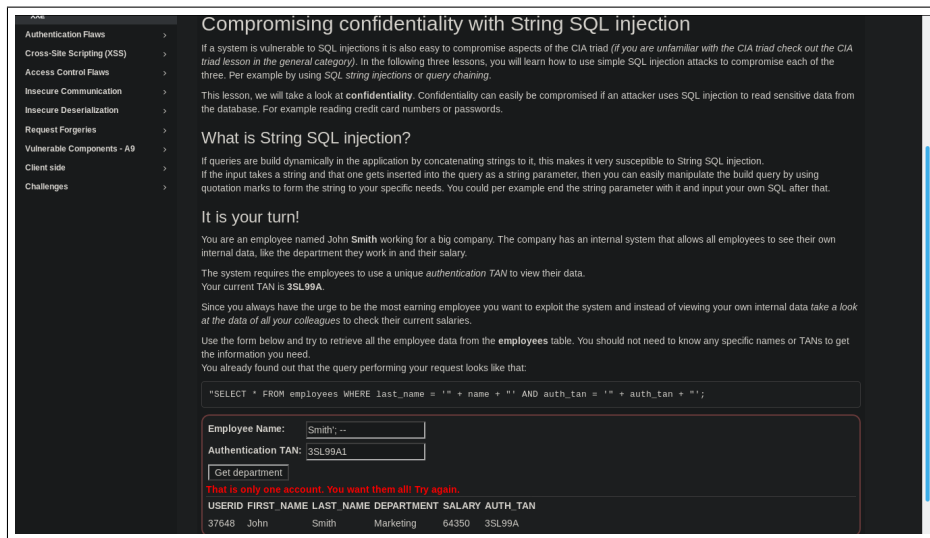


Figure 15: Trying SQLi by commenting the password field.

SELECT Query

Here the Query can be implemented in 2 Ways:

- Username - Smith
Password - ' OR '1'='1
- Username - Smith'; SELECT * FROM employees; –
Password - BLANK

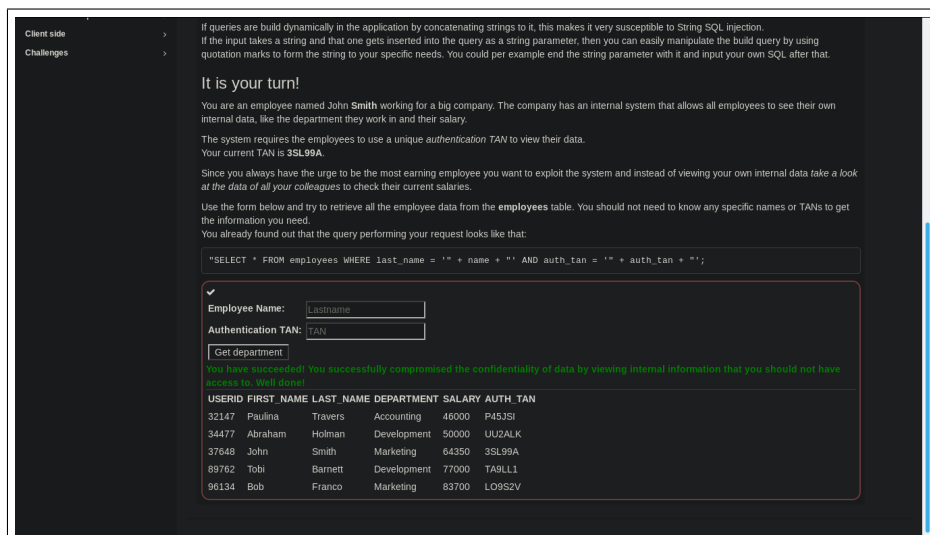


Figure 16: Successfully accessed the whole table

UPDATE Query

Employee Name - Smith'; UPDATE employees SET salary = '90000' WHERE last-name = 'Smith'; --
Auth-TAN - BLANK

SQL Injection (mitigation)

XXE

Authentication Flaws

Cross-Site Scripting (XSS)

Access Control Flaws

Insecure Communication

Insecure Deserialization

Request Forgeries

Vulnerable Components - A9

Client side

Challenges

Compromising Integrity with Query chaining

After compromising the confidentiality of data in the previous lesson, this time we are gonna compromise the **integrity** of data by using **SQL query chaining**.

The integrity of any data can be compromised, if an attacker per example changes information that he should not even be able to access.

What is SQL query chaining?

Query chaining is exactly what it sounds like. When query chaining, you try to append one or more queries to the end of the actual query. You can do this by using the ; metacharacter which marks the end of a query and that way allows to start another one right after it within the same line.

It is your turn!

You just found out that Tobi and Bob both seem to earn more money than you! Of course you cannot leave it at that. Better go and *change your own salary so you are earning the most!*

Remember: Your name is John **Smith** and your current TAN is **3SL99A**.

✓

Employee Name:

Authentication TAN:

Well done! Now you are earning the most money. And at the same time you successfully compromised the integrity of data by changing the salary!

| USERID | FIRST_NAME | LAST_NAME | DEPARTMENT | SALARY | AUTH_TAN |
|--------|------------|-----------|-------------|--------|----------|
| 37648 | John | Smith | Marketing | 90000 | 3SL99A |
| 96134 | Bob | Franco | Marketing | 83700 | LO952V |
| 89762 | Tobi | Barnett | Development | 77000 | TA9LL1 |
| 34477 | Abraham | Holman | Development | 50000 | UU2ALK |
| 32147 | Paulina | Travers | Accounting | 46000 | P45JSL |

Figure 17: Successfully compromised the Integrity of the Table by chaining Update Query.

WEBGOAT

SQL Injection (introduction)

Introduction

General

Injection Flaws

SQL Injection (introduction)

SQL Injection (advanced)

SQL Injection (mitigation)

XXE

Authentication Flaws

Cross-Site Scripting (XSS)

Access Control Flaws

Insecure Communication

Insecure Deserialization

Request Forgeries

Vulnerable Components - A9

Client side

Challenges

Compromising Availability

After successfully compromising confidentiality and integrity in the previous lessons, we now are going to compromise the third element of the CIA triad: **availability**.

There are many different ways to violate availability. If an account is deleted or the password gets changed, the actual owner cannot access it anymore. Attackers could also try to delete parts of the database making it useless or even dropping the whole database. Another way to compromise availability would be to per example revoke access-rights from admins or any other users, so that nobody gets access to (specific parts of) the database.

It is your turn!

Now you are the top earner in your company. But do you see that? There seems to be a **access_log** table, where all your actions have been logged to! Better go and *delete it* completely before anyone notices.

Action contains:

There is still evidence of what you did. Better remove the whole table.

| ID | TIME | ACTION |
|----|---------------------|---|
| 0 | 2019-10-08 18:31:26 | SELECT * FROM employees WHERE last_name = "" AND auth_tan = "" |
| 1 | 2019-10-08 18:31:46 | SELECT * FROM employees WHERE last_name = "Smith" AND auth_tan = "3SL99A" |
| 2 | 2019-10-08 18:31:53 | SELECT * FROM employees WHERE last_name = "Smith" AND auth_tan = "3SL99A" |

Figure 18: Searching with Empty field.

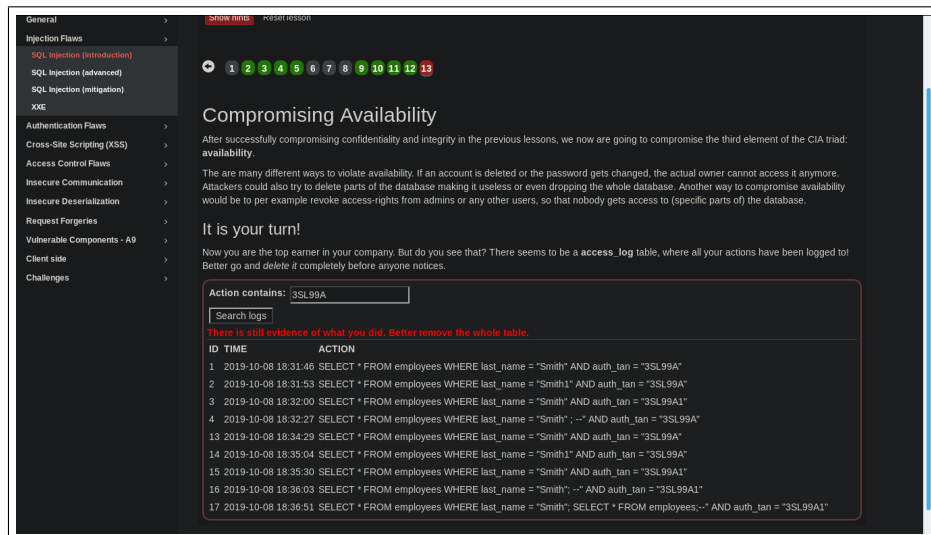


Figure 19: Searching the table with the Specified value.

DROP Query

}; DROP TABLE access-log; --

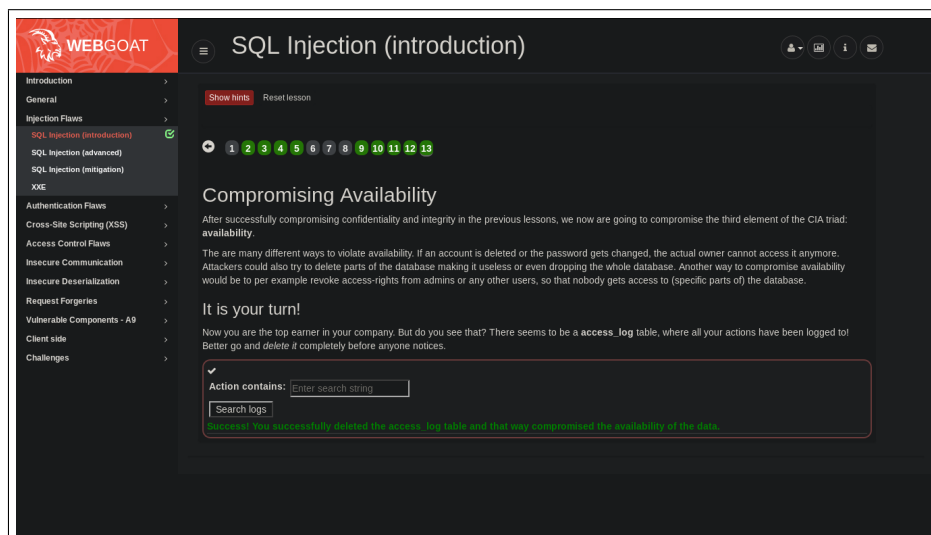


Figure 20: Successfully deleted the access-log table by chaining DROP Query.



Figure 21: Advanced SQL Injection

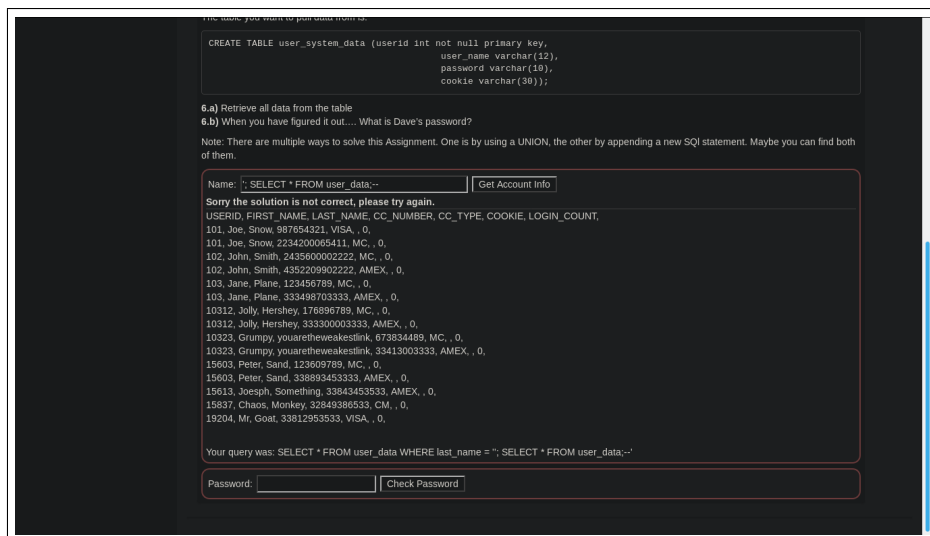


Figure 22: SELECT Query using Chaining.

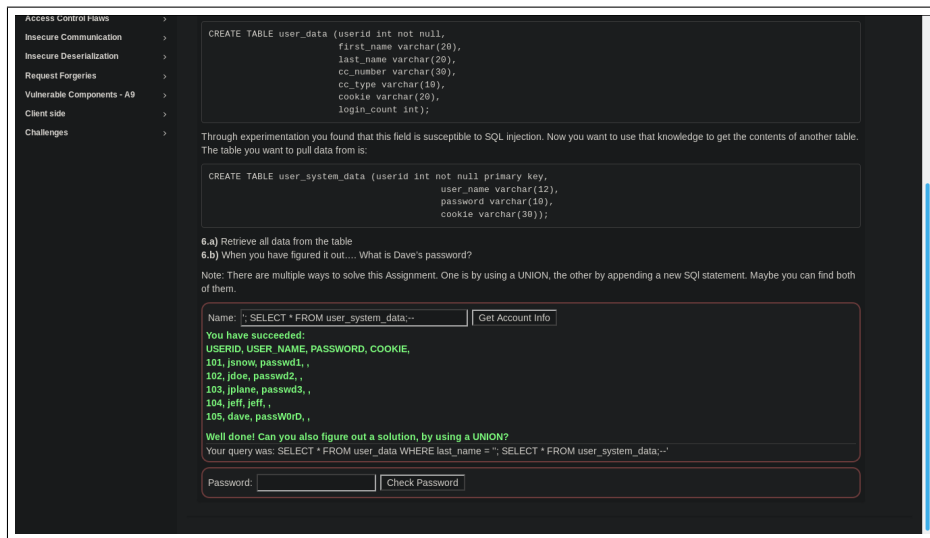


Figure 23: Again applying SELECT Query using Chaining on another Table.



Figure 24: We have successfully found out the required password.

Cross-Site Scripting (XSS) >
Access Control Flaws >
Insecure Communication >
Insecure Deserialization >
Request Forgeries >
Vulnerable Components - A9 >
Client side >
Challenges >

Blind SQL injection is a type of SQL injection attack that asks the database true or false questions and determines the answer based on the applications response. This attack is often used when the web application is configured to show generic error messages, but has not mitigated the code that is vulnerable to SQL injection.

Difference

Let us first start with the difference between a normal SQL injection and a blind SQL injection. In a normal SQL injection the error messages from the database are displayed and gives enough information to find out how the query is working. Or in the case of an UNION based SQL injection the application does not reflect the information directly on the web page. So in the case where nothing is displayed you will need to start asking the database questions based on a true or false statement. That is why a blind SQL injection is much more difficult to exploit.

There are several different types of blind SQL injections: content-based and time-based SQL injections.

Example

In this case we are trying to ask the database a boolean question based on for example an unique id, for example suppose we have the following url: `https://my-shop.com?article=4` On the server side this query will be translated as follows:

```
SELECT * from articles where article_id = 4
```

When we want to exploit this we change the url into: `https://my-shop.com?article=4 AND 1=1` This will be translated to:

```
SELECT * from articles where article_id = 4 AND 1 = 1
```

If the browser will return the same page as it used to when using `https://my-shop.com?article=4` you know the website is vulnerable for a blind SQL injection. If the browser responds with a page not found or something else you know a blind SQL injection might not work. You can now change the SQL query and test for example: `https://my-shop.com?article=4 AND 1=2` which will not return anything because the query returns false.

So but how do we actually take advantage of this? Above we only asked the database for trivial question but you can for example also use the following url: `https://my-shop.com?article=4 AND substring(database_version(),1,1) = 2`

Most of the time you start by finding which type of database is used, based on the type of database you can find the system tables of the database you can enumerate all the tables present in the database. With this information you can start getting information from all the tables and you are able to dump the database. Be aware that this approach might not work if the privileges of the database are setup correctly (meaning the system tables cannot be queried with the user used to connect from the web application to the database).

Another way is called a time-based SQL injection, in this case you will ask the database to wait before returning the result. You might need to use this if you are totally blind so there is no difference between the response you can use for example:

```
article = 4; sleep(10) --
```

Figure 25: Blind SQL Injection Introduction.

WEBGOAT

SQL Injection (mitigation)

Show hints
Reset lesson

1 2 3 4 5 6 7 8 9 10 11

Immutable Queries

These are the best defense against SQL injection. They either do not have data that could get interpreted or they treat the data as a single entity that is bound to a column without interpretation.

Static Queries

```
select * from products;
```

```
select * from users where user = "" + session.getAttribute("UserID") + "";
```

Parameterized Queries

```
String query = "SELECT * FROM users WHERE last_name = ?";
PreparedStatement statement = connection.prepareStatement(query);
statement.setString(1, accountName);
ResultSet results = statement.executeQuery();
```

Stored Procedures

Only if stored procedure does not generate dynamic SQL

Figure 26: Some SQL Injection Mitigation Practices.