Error Based SQL Injection

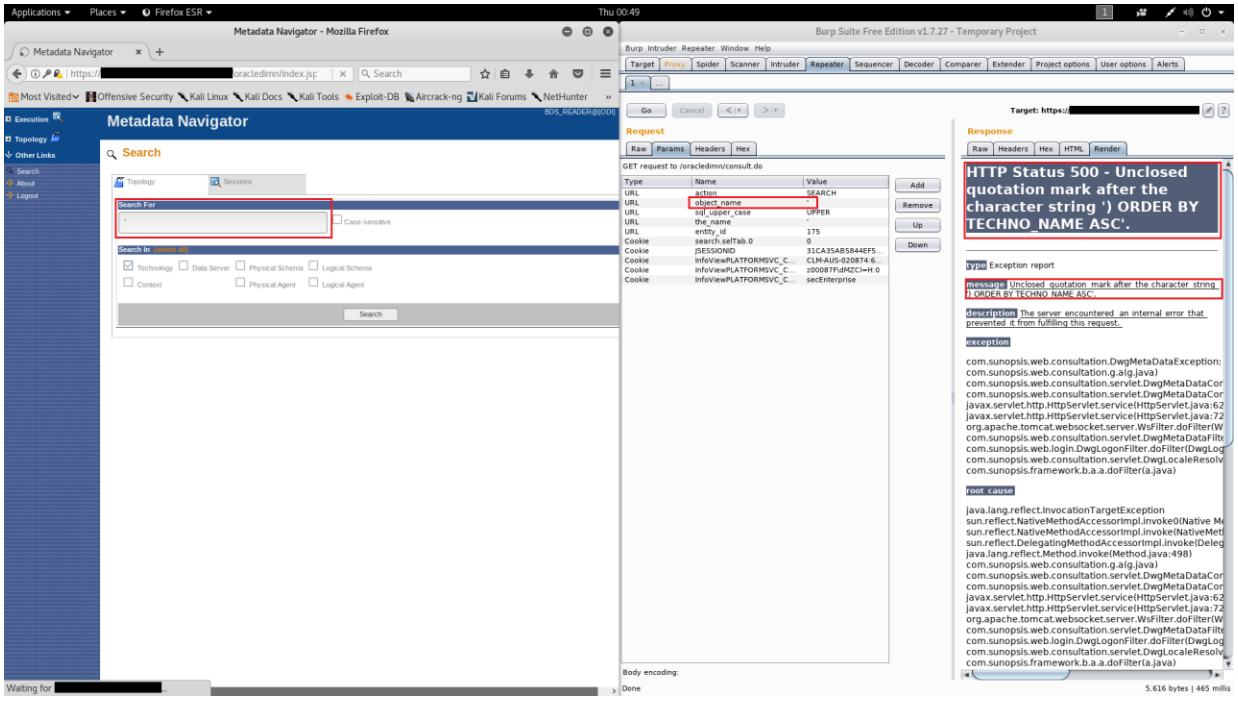
Vulnerability Description:

SQL injection vulnerabilities arise when user-controllable data is incorporated into database SQL queries in an unsafe manner. An attacker can supply crafted input to break out of the data context in which their input appears and interfere with the structure of the surrounding query. In error-based injection, we'll be looking for errors generated by the database when we pass unsanitized input in the URL.

We found the Vulnerability in a Third-Party Application i.e ORACLE ODI MetaData Navigator. Application is Vulnerable to Error Based SQL Injection in Search tab Logical Parameter Search For and Actual Parameter object_name.

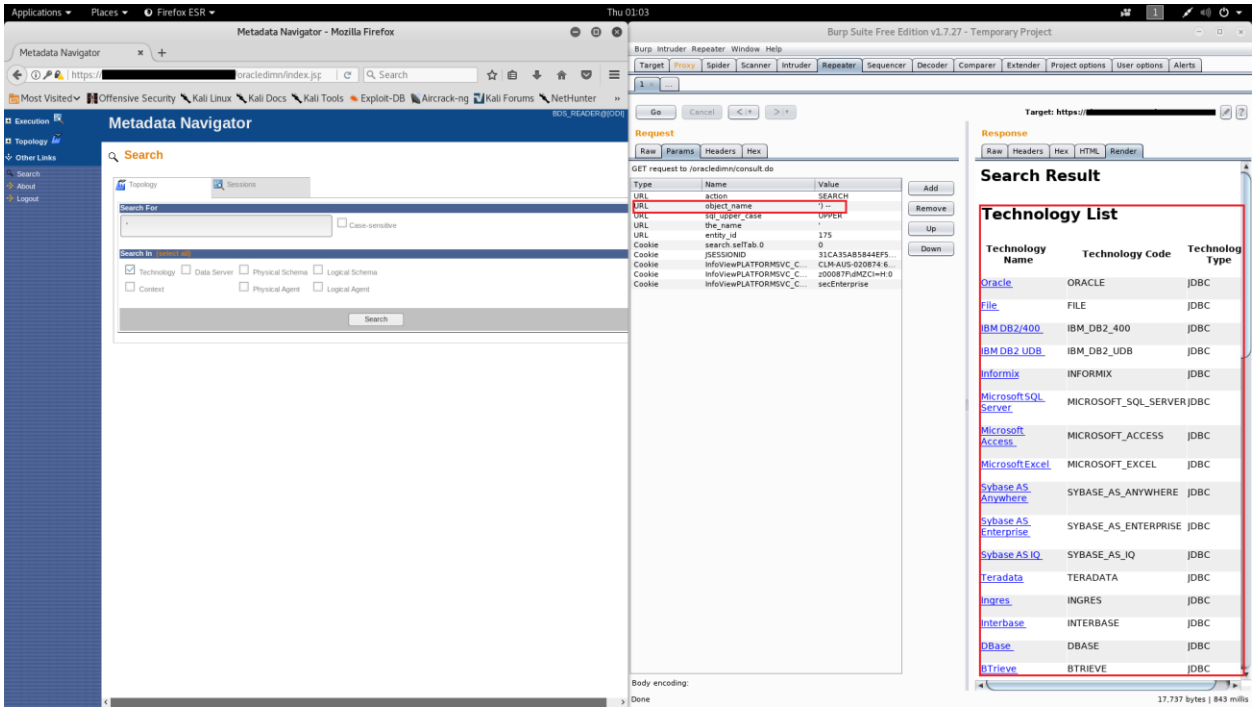
1. In this Search For, we put a Single Quote (') in the text parameter and intercept the request with burp and then check object_name as single quote Parameter then we send a request to the server after we got an error.

   If it shows an error and the error is a SQL syntax error, then this page is connected with the database and website is also vulnerable. What the single quote does is actually single quote breaks the query and the syntax of the query has been changed, or we could say that it was converted into the wrong syntax. So, we get the error SQL syntax error.

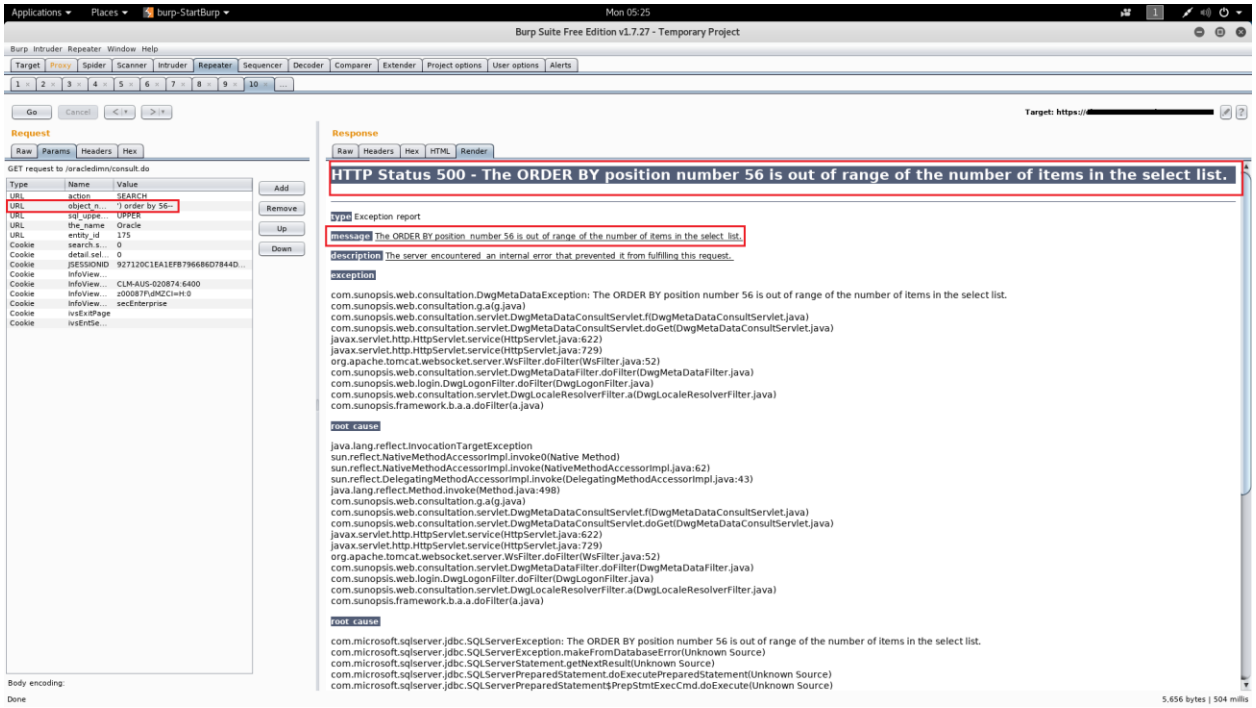


Deepak Sharma

2. Now we know the website is vulnerable to SQL Injection just need to patch the error by using the appropriate comments so that error has fixed. We are using ") --" comments and fixed the query.

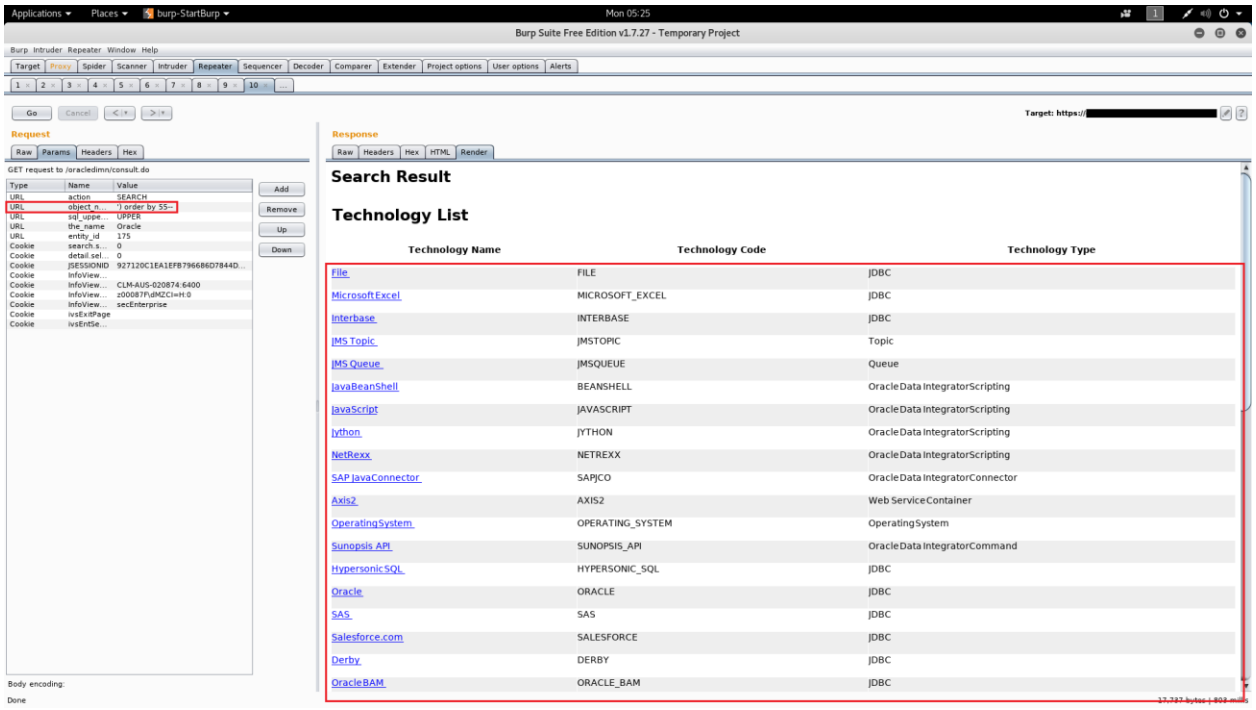


3. Again we are going to inject a query for getting the error. Now, what we do is getting the column number by using 'order by clause' like order by 1, order by 2,.... We will do this until it shows up in the unknown columns. If it shows the unknown column error on N, that means it has the total number of Columns N-1 because it shows the content in order by N-1.
4. In below snapshot, we got an Error on **56 column number** when we used **order by 56**.

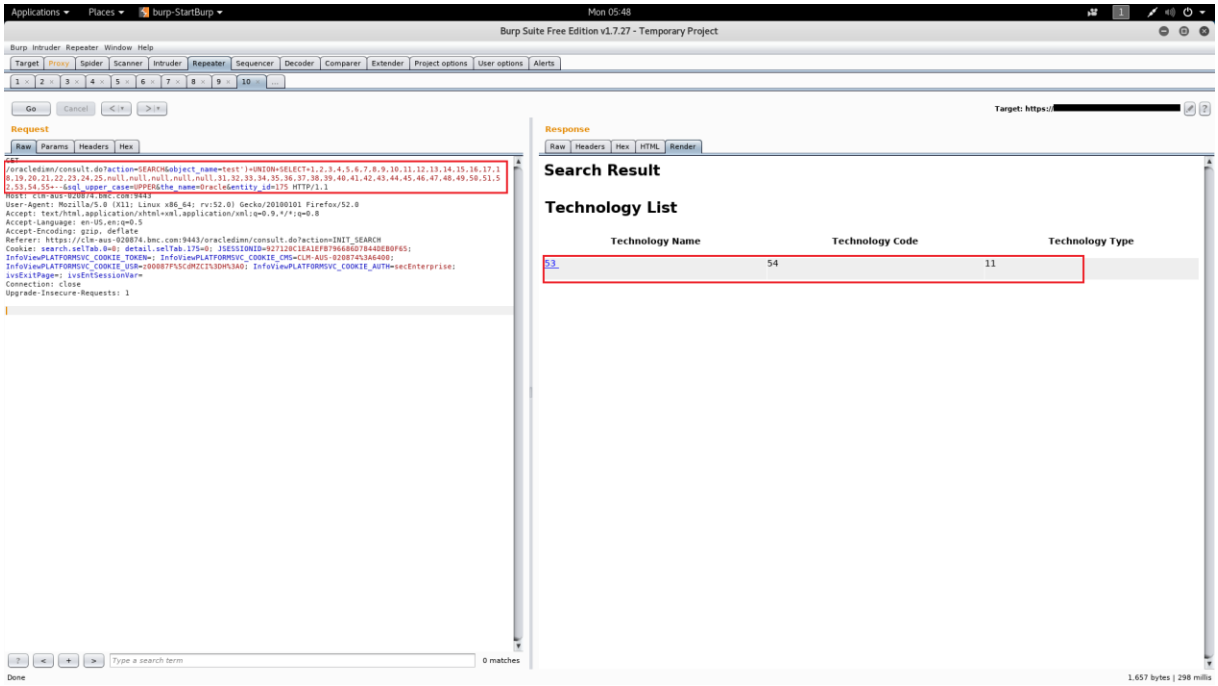


Deepak Sharma

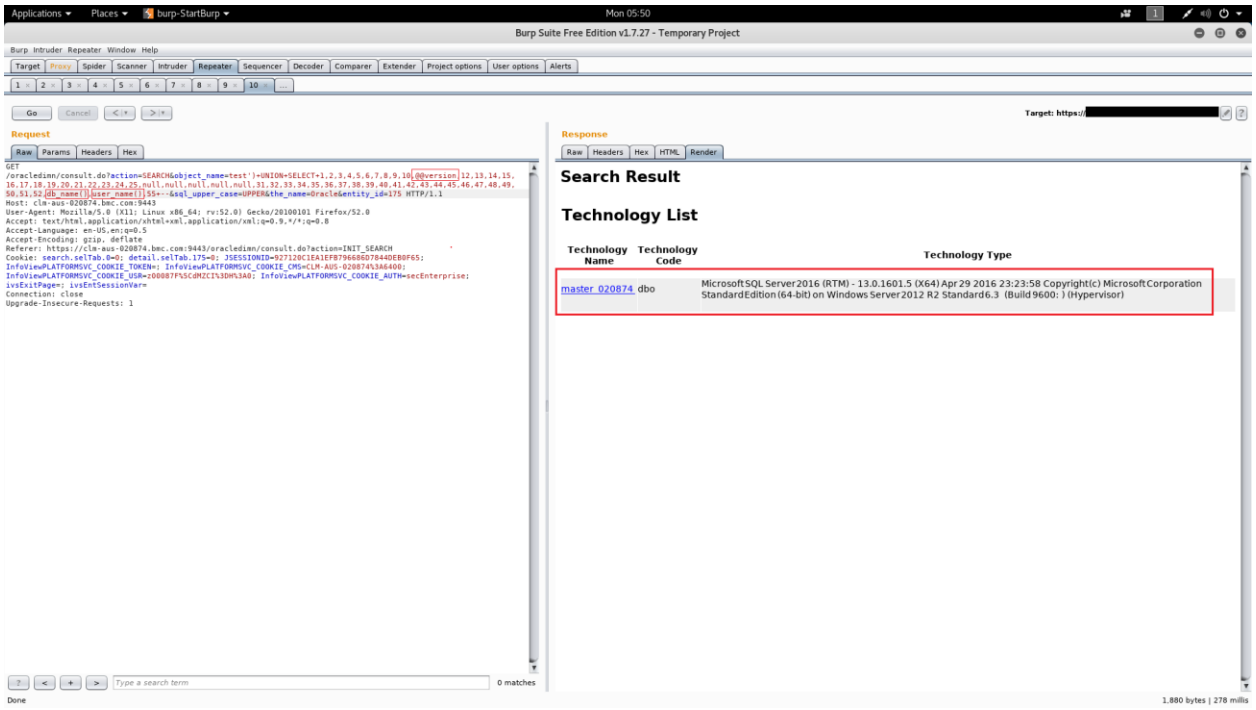
5. But when we checked for **55 column number** we didn't get any error it means there is 55 columns.



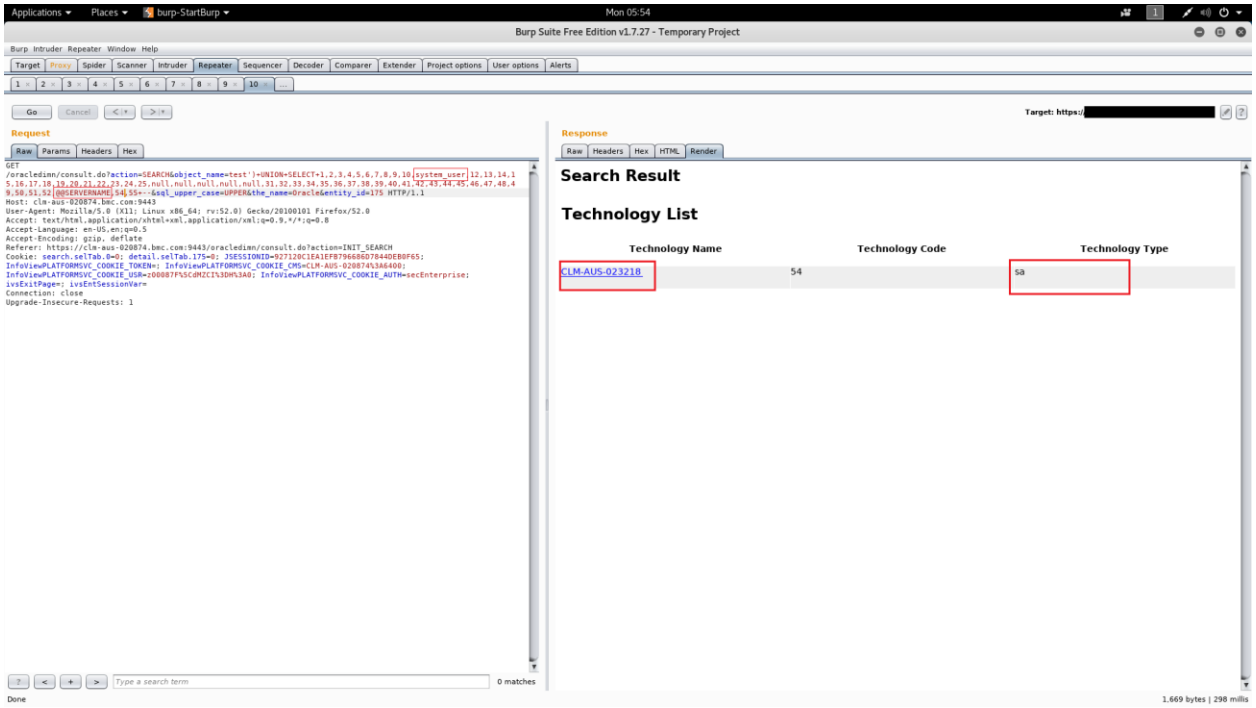
6. Now we are going to find out which columns are connected to our current URL page so that we can say which columns are vulnerable. Technically it's not vulnerable. It is just connected to our current URL page.
7. So with the help of Union and Select statement and Null query we have found which column number are vulnerable and shows on webpages. It shows us the current page connected columns because we can only access the database by these columns. It shows the columns' number. If it shows **11, 53 and 54** then this is connected with our current URL page. The next step is to find out the version and database. We will do this by using any one of the **11, 53 or 54**. We are using **11, 53 or 54** just because they are only connected to the current page.



8. We can find other information Like the **database name** and it's **version**, Current user by using a few things. As we are using a function **@@version** in column number 11 for getting the **Database version, db_name()** function in column number 53 for getting the Database name save in Database and **user_name()** function in column number 54 for getting current user name that access to the database.



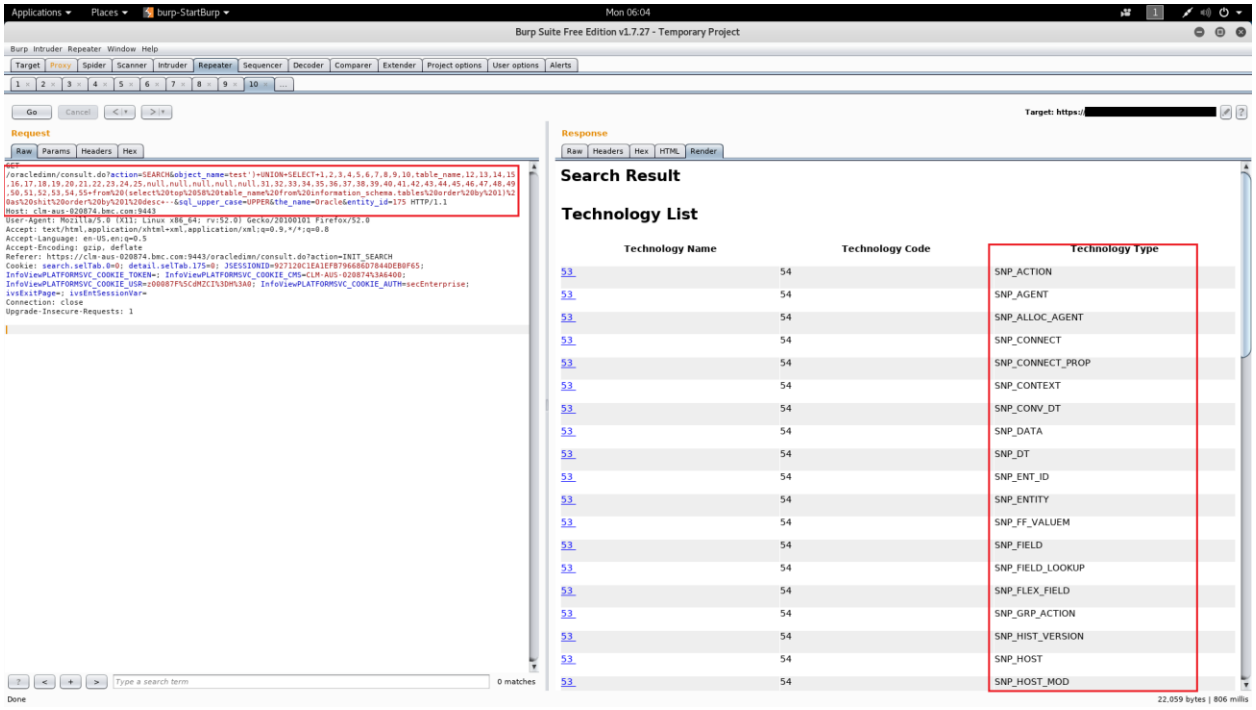
9. We should have one thing to keep in mind is : The more information you have, the more powerful you will be. So we are also getting **system username** and the **Server name or hostname**.

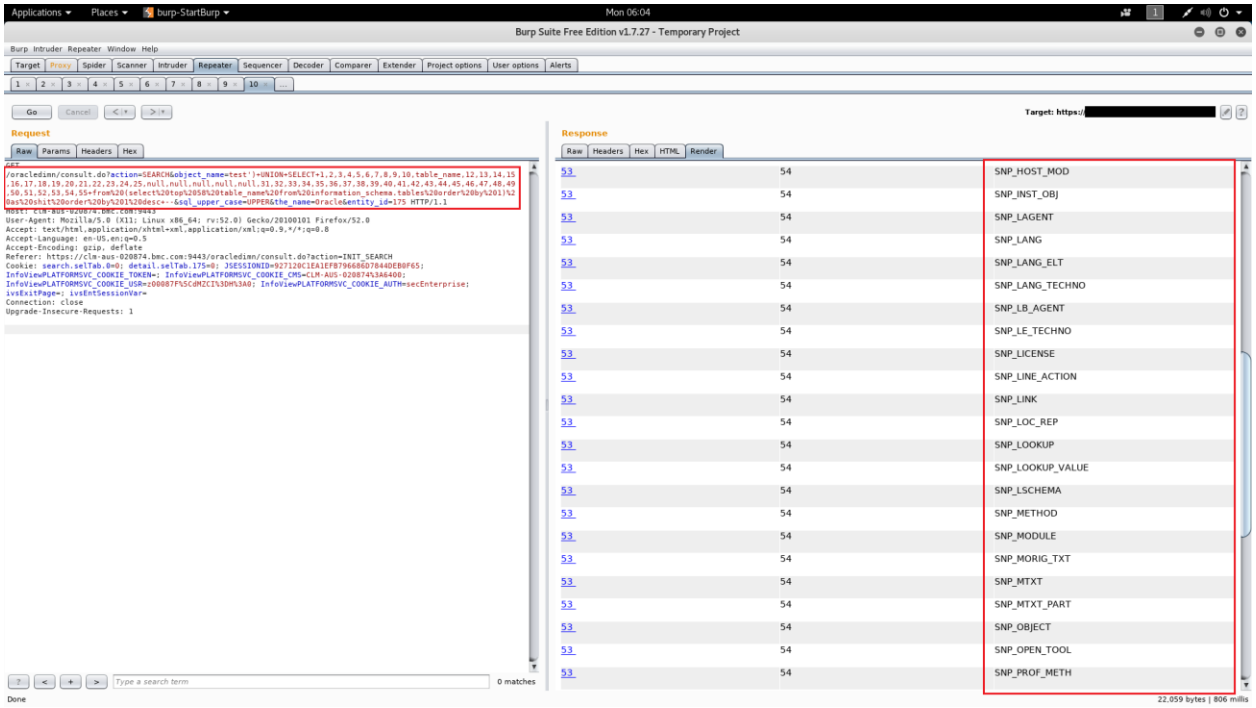


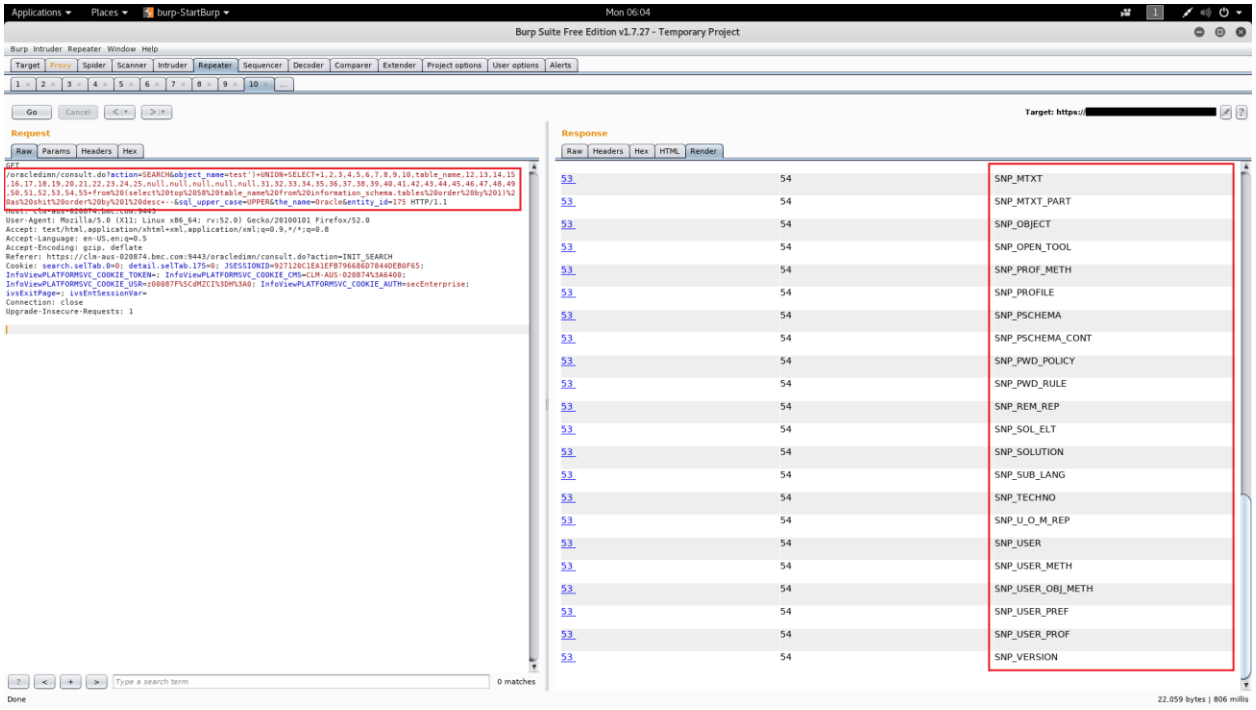
Deepak Sharma

10. Now our next step is to find out all the table names. To do this, we will again inject a query. If the version of SQL is more than **5.0.0**, it means it has the schema. Schema is just like an index of all databases.

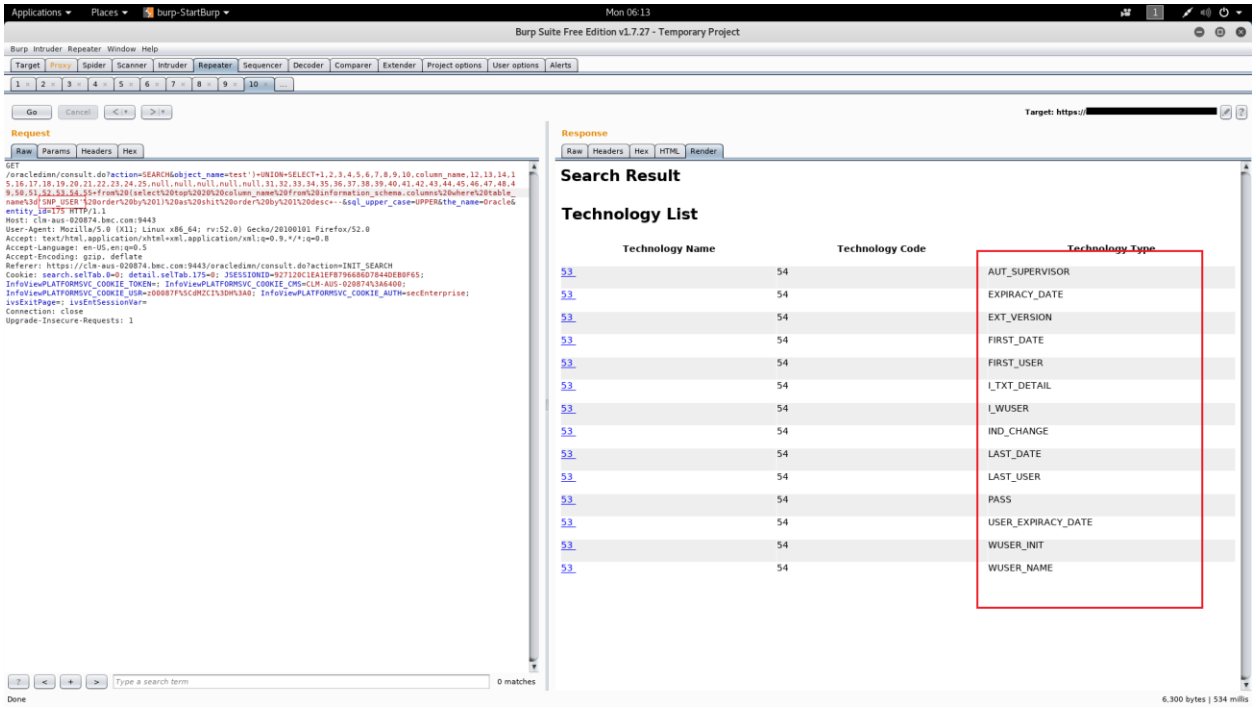
11. It will show all the table names from the **table_schema**.







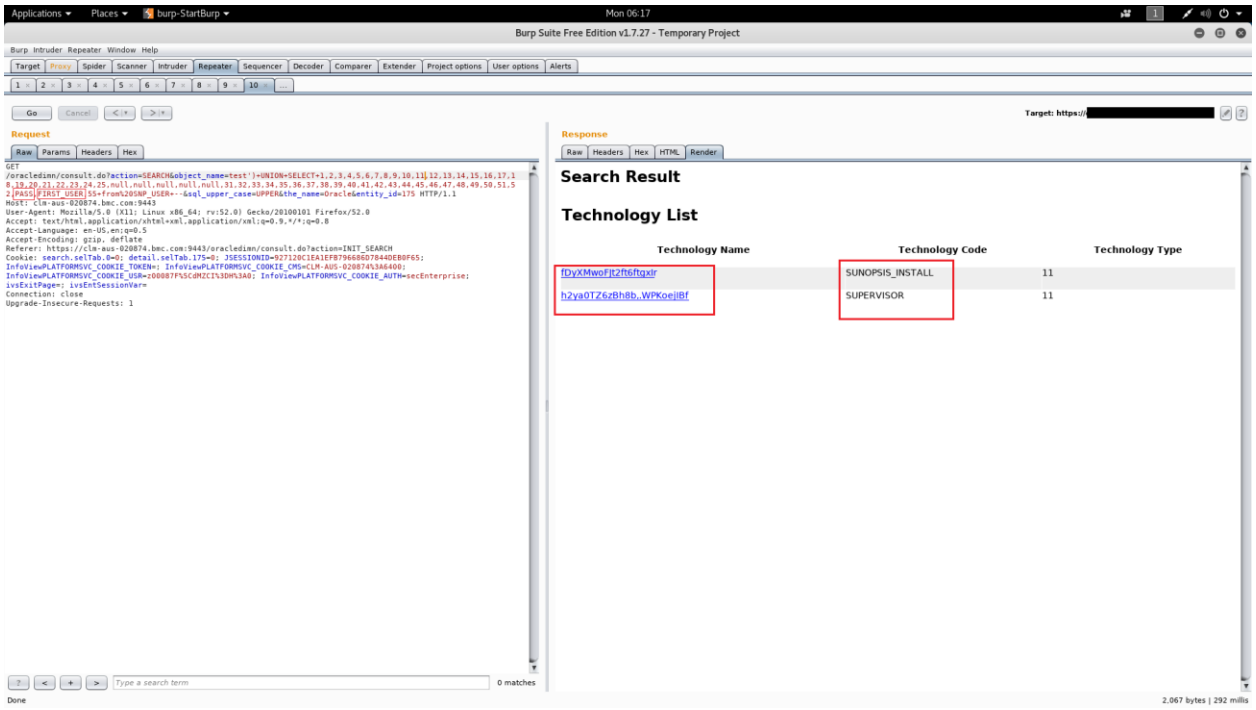
12. Our next step is to find out all the columns. Take any table name usually we are going to choose a table name in which we think we will get critical information like Usename and Password or Credit card information, because this information is too much critical. In our case I choose table name 'SNP_USER' because in below snapshot i got username and password field.



Deepak Sharma

13. Now we extract the Username and Password field from the column name 'SNP_USER'. With the help of these Username and password we will try to login and check for Administration privileges in database.

14. You can also extract the more field and column by using the same steps follow onwards from Step 7.



15. A wide range of damaging attacks can often be delivered via SQL injection, including reading or modifying critical application data, interfering with application logic, escalating privileges within the database and taking control of the database server.

Remediation:

The most effective way to prevent SQL injection attacks is to use parameterized queries (also known as prepared statements) for all database access. This method uses two steps to incorporate potentially tainted data into SQL queries: first, the application specifies the structure of the query, leaving placeholders for each item of user input; second, the application specifies the contents of each placeholder. Because the structure of the query has already been defined in the first step, it is not possible for malformed data in the second step to interfere with the query structure. You should review the documentation for your database and application platform to determine the appropriate APIs which you can use to perform parameterized queries. It is strongly recommended that you parameterize *every* variable data item that is incorporated into database queries, even if it is not obviously tainted, to prevent oversights occurring and avoid vulnerabilities being introduced by changes elsewhere within the code base of the application.

You should be aware that some commonly employed and recommended mitigations for SQL injection vulnerabilities are not always effective:

- One common defense is to double up any single quotation marks appearing within user input before incorporating that input into a SQL query. This defense is designed to prevent malformed data from terminating the string into which it is inserted. However, if the data being incorporated into queries is numeric, then the defense may fail, because numeric data may not be encapsulated within quotes, in which case only a space is required to break out of the data context and interfere with the query. Further, in second-order SQL injection attacks, data that has been safely escaped when initially inserted into the database is subsequently read from the database and then passed back to it again. Quotation marks that have been doubled up initially will return to their original form when the data is reused, allowing the defense to be bypassed.
- Another often cited defense is to use stored procedures for database access. While stored procedures can provide security benefits, they are not guaranteed to prevent SQL injection attacks. The same kinds of vulnerabilities that arise within standard dynamic SQL queries can arise if any SQL is dynamically constructed within stored procedures. Further, even if the procedure is sound, SQL injection can arise if the procedure is invoked in an unsafe manner using user-controllable data.

For more details please refer following link:

- SQL Injection Prevention Cheat Sheet