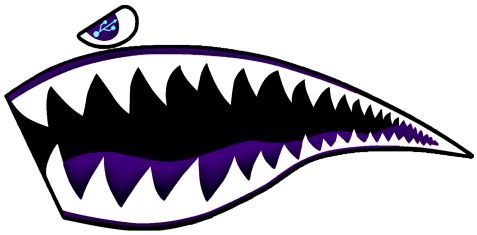
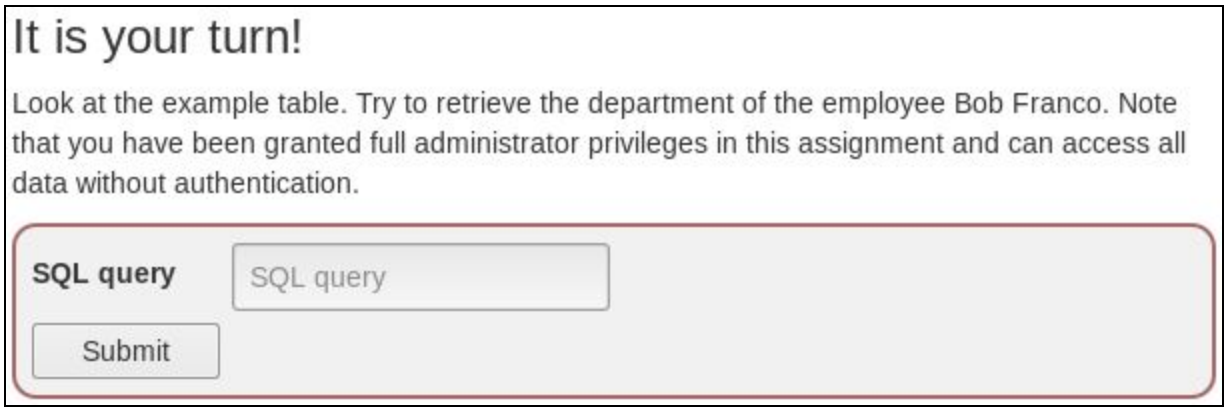


WebGoat SQL Injection (Introduction)

Twitter: @BlackSheepSpicy
Twitch: https://twitch.tv/BlackSheepSpicy

Settle in ladies and gentlemen, we're in for a long one. Also keep in mind I am in no way a SME regarding SQL so the methods used here might not be (read: are probably not) best practices.

2. Onto the first challenge then:



Before we go any further, let's all take a look at the example table that OWASP has provided further up the page:

Example SQL table with employees, the name of the table is 'employees':

Employees Table

userid	first_name	last_name	department	salary	auth_tan
32147	Paulina	Travers	Accounting	$46.000	P45JSI
89762	Tobi	Barnett	Development	$77.000	TA9LL1
96134	Bob	Franco	Marketing	$83.700	LO9S2V
34477	Abraham	Holman	Development	$50.000	UU2ALK
37648	John	Smith	Marketing	$64.350	3SL99A

Lowkey feel a bit spoiled with the amount of information we have to complete this challenge.

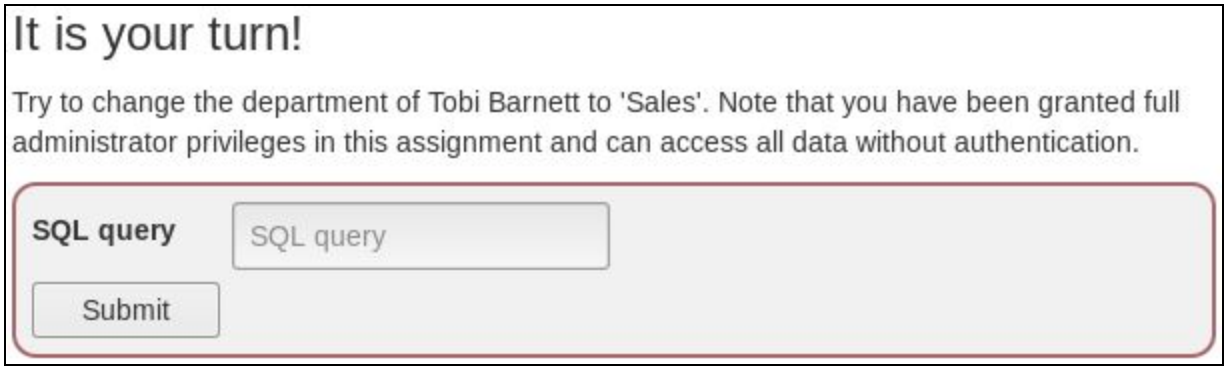
Armed with all the database info we could ever want, we can now construct our query:

```
select * from employees where last_name = 'Franco';
```

Lets go ahead and break down this query anyway to inflate my ego a bit, if you're already familiar with SQL feel free to skip this section:

- **select:** pretty straightforward statement, just tells the database to return the following information.
- ***:** this symbol is a bit vague, but in SQL (and in many other contexts) this is known as a wildcard character, basically it's just short for the word "any" or "all"
- **from employees:** just telling the database which table to fetch the info from (in this case the **employees** table)
- **Where last_name = 'Franco':** conditional statement where we specify that we only want rows where the column **last_name** has only the word "**Franco**" in it.

3. Moving on, we are now tasked with modifying data with a query:

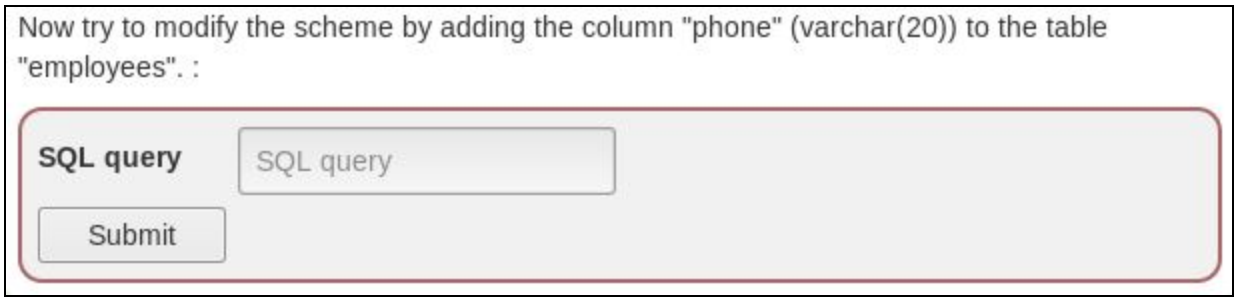


There's a couple new statements to be used here from the previous challenge, but once again not the worst thing ever, especially when we can reference the example in the previous challenge:

```
update employees set department = 'Sales' where last_name = 'Barnett';
```

All this query does is tells the database to **update** the table **employees** to **set** the column **department** to **sales** for any entry **where** the column **last_name** has only the letters "**Barnett**".

4. So now we know how to modify existing data, what if we want to add or remove variables (or columns, depending on how pedantic you want to be)? Let's take a look at the challenge on page 4:

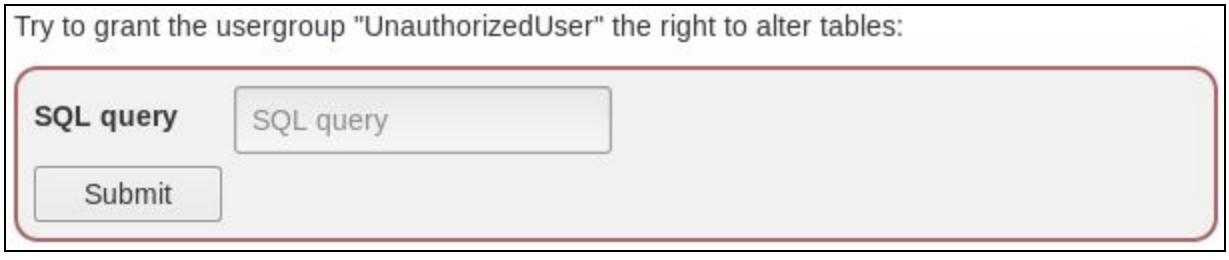


Now your first reaction might be to use the update statement again here (hopefully not, please be smarter than me and read the content webgoat offers in the pages) however instead we must instead use a new statement: **alter.** The resulting query looks like this:

```
alter table employees add phone varchar(20);
```

This query tells the database to **alter** the **table employees** to **add** the variable (column) **phone** which will be a **variable character field** that can contain a maximum of **20** characters.

5. SQL can do a lot of different things, and when I say a lot I mean *a lot*. Some might say too much (provided we have admin rights but honestly when has that ever stopped us before?) and honestly I'd be inclined to believe them. Enter the challenge on page 5:



Yep… we can modify privileges straight from queries… that's… that's where we're at...

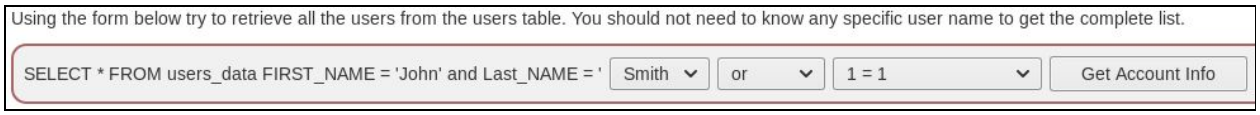
Alrighty so… we got our objective… let's… let's write our query then….

```
grant alter table to UnauthorizedUser;
```

If you've followed along with the previous challenges in this writeup this should be pretty clear. All it says is to **grant** the permission **alter table to UnauthorizedUser**

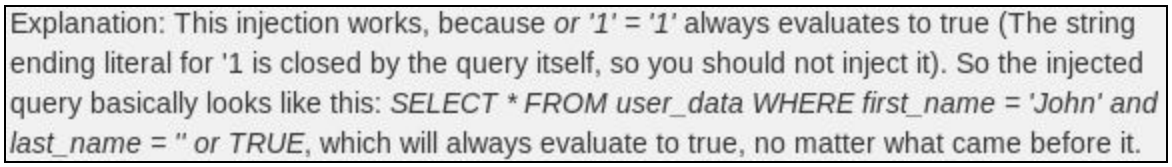
seriously who's fucking idea was this?

9. Cool! We know the basics of SQL, now lets see how we can break it! Forward unto the page 9 challenge (after taking a look at the couple pages prior to this, there's some bomb info there regarding SQLI):



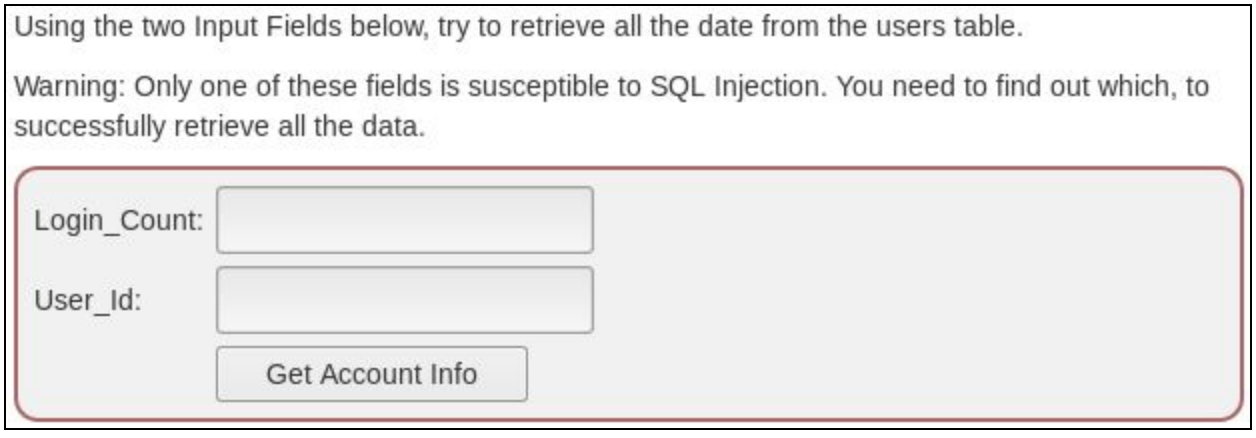
So… its a drop down menu, not as exciting as manually typing it out and feeling 1337 af but its still good practice, let's break down my solution here (after I correct my notes because I somehow managed to write down the wrong solution, I'm a professional btw):

Or… we could just let WebGoat steal my thunder, that's fine...



SQL is like a basic gen x kid: as long as you keep saying true it'll spill its guts to you if left unchecked.

10. Welp we know how string SQL injections work now for the most part, turns out we can do it with other data types too:

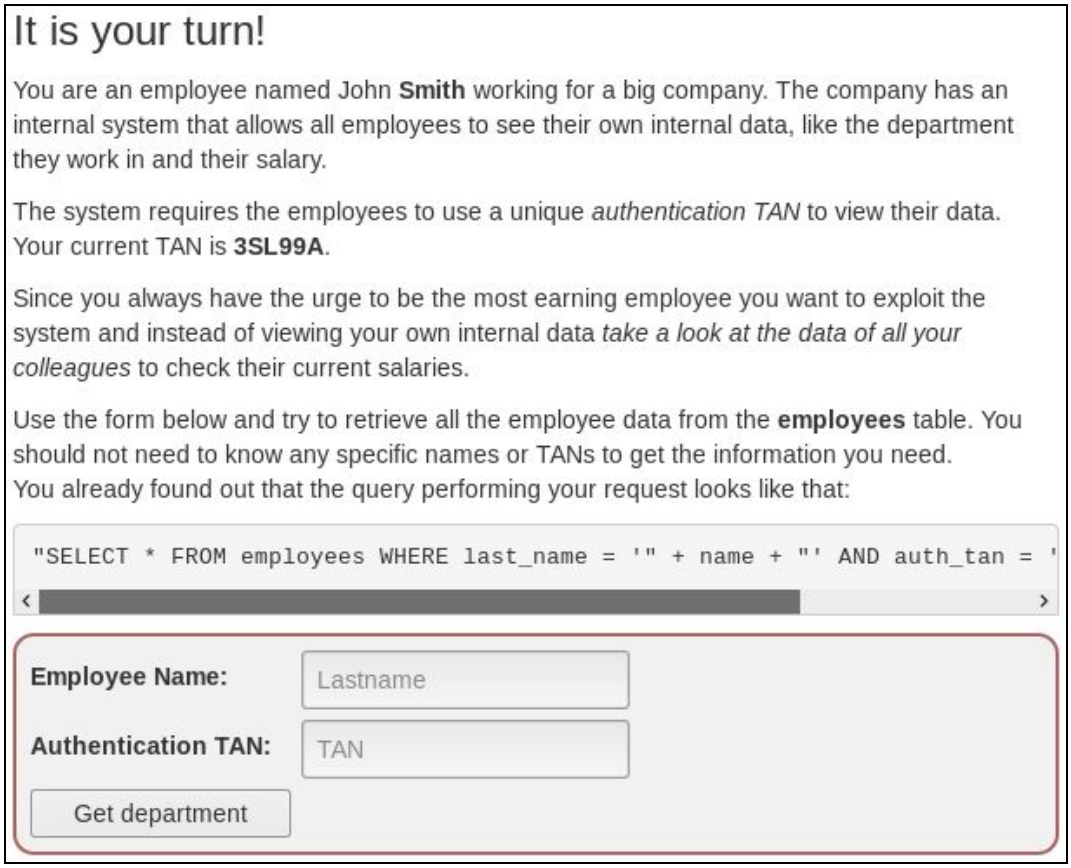


Well… we would do a numeric SQLI if we wanted to follow the rules… but like WebGoat said before… all we have to do is have a statement return true… so lets do that:

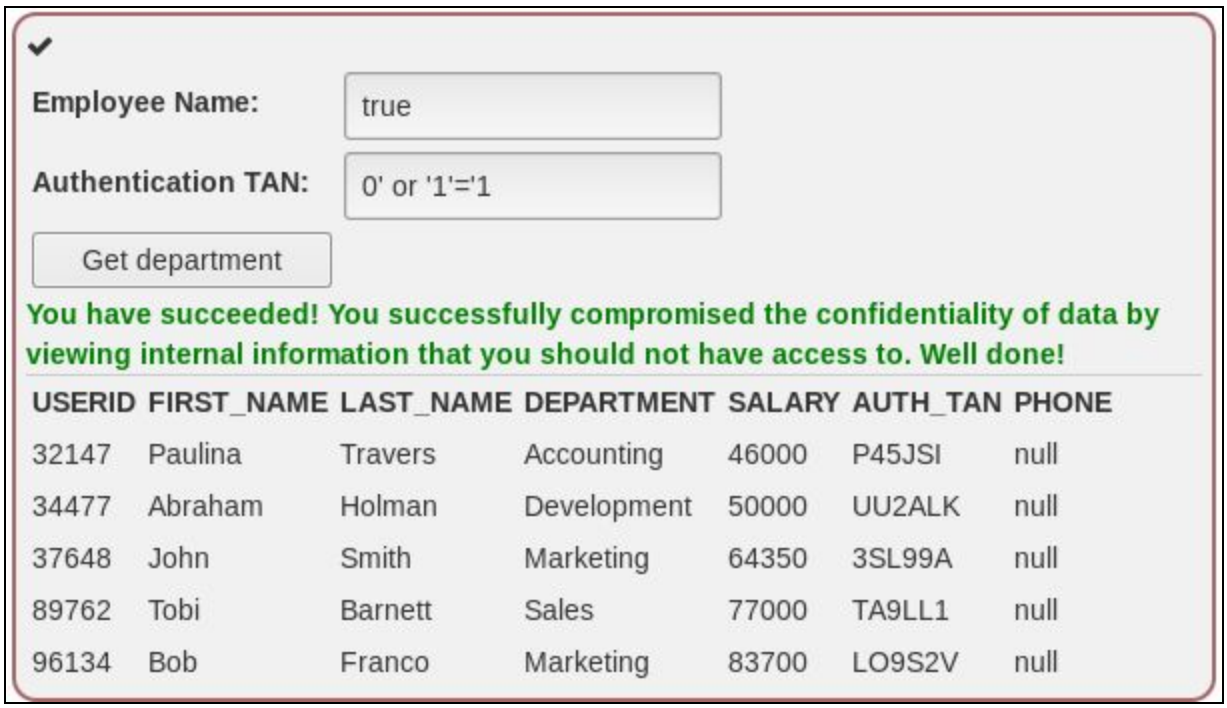
I mean we returned true ¯_(ツ)_/¯

It's also worth mentioning that rather than methodically looking for the injection flaw I just kinda fuzzed both inputs with "or true" until something good happened.

11. Y'all wanna get into some roleplay? Because OWASP is into it apparently:

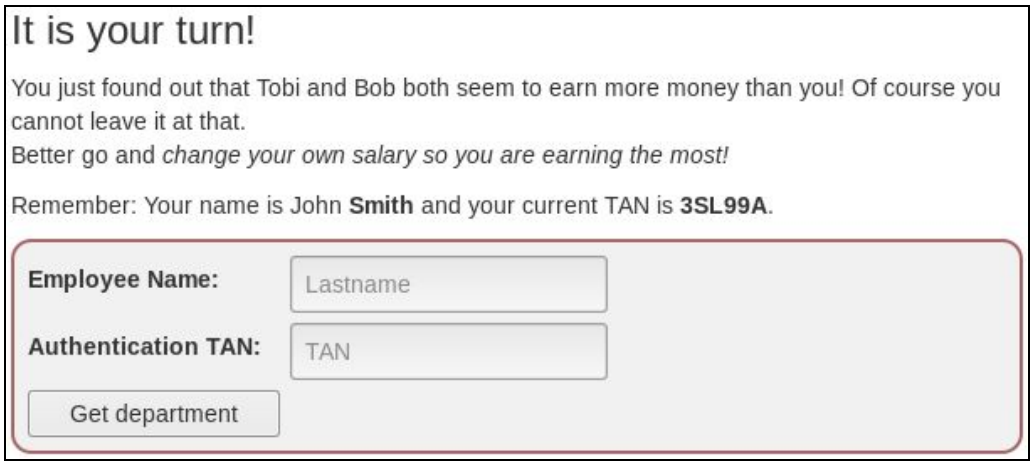


So once again rather than doing this with any sort of tact we can just use the shotgun method again using everything we have learned up to this point:



So in this instance we use **true** for the employee name because… well… they let us… but notice how we had to use a different injection string for the TAN input, this is because the database throws a fit if we give it an incorrectly formed string, so with that we can just chuck in anything we want as long as we manage to finesse a true or condition somewhere in there.

12. Continuing in our twisted roleplay, we are presented with a rather charged challenge:



So let's go commit a felony! Complete with new techniques to dive into!:

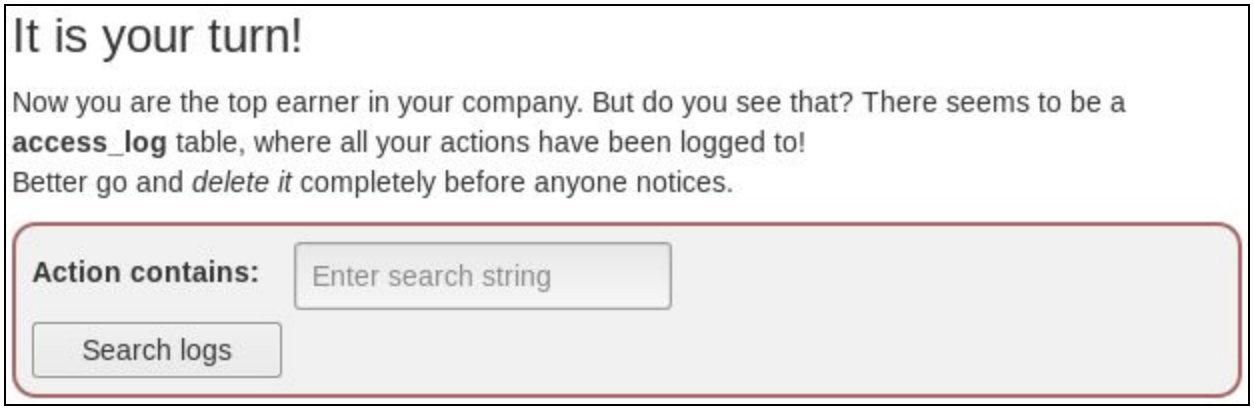
```
Employee Name: '; update employees set salary = 1000000 where last_name='Smith' --
Authentication TAN:
```

Query was too long to show in the input field, so here's what we got to work with

So as we can see, pretty standard fare here for the most part, but take a look at the beginning and end of the Employee Name input field we have the following characters:

- **';** : So we have the single quote to complete the string, and then we have a semicolon following it up. The keen eyed among you might have noticed them at the end of my previous queries. In SQL a semicolon is used to signify the end of a query, though in this context we can use this to kill the original query and then tack whatever query we want after the semicolon through something called **Query Chaining**.
- **--** : In SQL two dashes are used to denote a comment, meaning the database will ignore everything contained within it and another set of double dashes. Which in our case, means everything after this statement is ignored, meaning only our injected query will be processed, which is why we can leave TAN empty and the database won't throw a fit.

13. So now we're waist deep in shit due to OWASP's twisted fantasies, and I don't know about y'all but I ain't getting arrested over this, we have to cover our tracks:



It's only illegal if we get caught right?

We also get introduced to a new statement: the confusingly named **drop** statement (No idea why it isn't just delete but we use what we got)! The drop statement allows someone to completely remove a table from the database. Perfect for our needs!:

So in this challenge we can just simply use query chaining and our comment statement for added overkill then inject our drop statement to get away with it scott free (until they load the log backups of course, then we're a bit screwed...)

I really hope this write up will be helpful while getting into webapp pentesting! If you want to see me do these challenges live be sure to drop by my Twitch when I'm live and also follow my Twitter for some fresh memes!