

SQL Injections

1. Whitebox testing: I have the app implementation

App: Second Hand Cars Business website (*my warm up example*)

This webapp was fully developed by me, using a MySQL database. On start it prompts a login form which can be bypassed with *idk' or 1=1 #*, because the query is a simple:

```
"select * from users where username='" + user + "'and password='" + password + ""
```

Then, as a hacker, the next phase of the attack is to maintain my access (<https://mstefanc.com/cybersecurity-info/> - here I list and explain the phases of a cyberattack). Therefore, I can create my own user in the 'users' table. This way it would be less noisy than always using SQL injections in the input fields, and, if the vulnerability is solved, I can still login.

1st way to add the user: use the car 'Color' input field of the add car form, as the query is:

```
"INSERT INTO cars(model, hp, fuel, price, color, age) VALUES ('" + car.Model + "', '" + car.Hp + "', '" + car.Fuel + "', '" + car.Price + "', '" + car.Color + "', '" + car.Age + "');"

with the injection: g', '1'); insert into users(username, password) values('xd', 'xd'); #
```

2nd way to add the user: use the 'Color' input field from the update car form, as the query is:

```
"UPDATE cars SET model='" + car.Model + "', hp='" + car.Hp + "', fuel='" + car.Fuel + "', price='" + car.Price + "', color='" + car.Color + "', age='" + car.Age + "' WHERE id='" + car.Id + "';"

with the injection: g' where id='6'; insert into users(username, password) values('xd', 'xd') #
```

2. Blackbox testing: I don't have any knowledge about the app implementation

App: BadStore.net website (*hosted on my local Linux server – a realistic example*)

Let's obtain **full control** over the website by only using SQL Injections 😊. The following is proof that you don't need very complex / complicated SQL Injections to "pwn" an old unpatched web service.

First, test if the app is vulnerable to SQL Injections: input in "Quick Item Search" field a single comma, and it shows an error with some important information: type of DB – **MySQL**, table(itemdb) and columns(itemnum, sdesc, ldesc, price) names, and the **path** to the .cgi file: /usr/local/apache/cgi-bin/badstore.cgi. Because of MySQL, we can use the '#' comment instead of the '--' one.

Now we can get all the items in the store with an injection like: **'OR 1=1 #**

We can also try to force login as a random user with the same **'OR 1=1 #** because we don't know any exact usernames. I usually don't like spending time guessing (manual brute forcing) when there are tools which can do that, but this is what a **blind SQL Injection** means.

You may have problems with the max input size for a field if you want to write longer injections, so you can change it in the browser page inspection (the maxLength input attribute).

Now, because we know that MySQL is being used, we can take advantage of its features. One of its amazing features is **LOAD_FILE**, which can load a file from the server into a table.

Because I know that the app is hosted on an old Linux server, I can try to get the contents of '/etc/passwd' in a table from the site. We can use the previous "Item search" table as we already got some crucial info on it.

So, we know that the app uses there the query :


```
SELECT itemnum, sdesc, ldesc, price FROM itemdb WHERE '<my_input>' IN (itemnum,sdesc,ldesc)
```

Therefore, I can use an injection where I make the WHERE clause false (to avoid showing any item from the itemdb table), followed by an UNION SELECT (that needs to have the same number of columns as itemdb - 4 columns) **which will display the data in the same columns** where the itemdb table would normally display its data (but this time it won't because I set its 'where' to be always false). In the new select, I choose the "Description" column to show the content of the file, and I set the 3 remaining columns to 0.

- ⇒ **<my_input>** is `1'='0' UNION SELECT 0, 0, LOAD_FILE('/etc/passwd'), 0 #`
- ⇒ Full query:

```
SELECT itemnum, sdesc, ldesc, price FROM itemdb WHERE '1'='0'
UNION
SELECT 0, 0, LOAD_FILE('/etc/passwd'), 0 # the rest is ignored
```

- ⇒ We find out about the users of the Linux server that hosts the app: *root* and *nobody*:

ItemNum	Item	Description	Price	Image	Add to Cart
0	0	root::0:0:Trinux Root::/bin/sh nobody:x:65534:65534:nobody:/tmp:/bin/sh	\$0.00		<input type="checkbox"/>

Now, because this was possible, it means that I can get the content of other important files from the server. But what should I be looking for? Well, I know that **.cgi** scripts are executable files that come with some security vulnerabilities, so I'll just get the content of the .cgi file that the MySQL error disclosed at the beginning (/usr/local/apache/cgi-bin/badstore.cgi).

To do this, I first set the input *maxlen* attribute to 420 to fit the longer query, and then I use the injection in the same input :

```
1'='0' UNION SELECT 0, 0, LOAD_FILE('/usr/local/apache/cgi-bin/badstore.cgi'), 0 #
```

As a result I get a lot of messed up code from that script, but no problem : CTRL-F was invented. Most probably, because there's so much code, this script does almost everything (which, btw, is an atrocious practice), including connecting to the database 😊.

Therefore I search for 'connect' and in a matter of seconds I find:

```
connect("DBI:mysql:database=badstoredb;host=localhost", "root", "secret")
```

Voilà, now we have the **user** (*root*) and the **password** (*secret*) to connect directly to the app's database. So I boot up my Kali Linux and I use the following commands

Connect to MySQL database:

```
mysql -u <user> -p <user_password> -h <host_IP_address>
```

MySQL commands:

```
SHOW DATABASES; -- shows me the DB badstoredb
```

```
USE badstoredb;
```

```
SHOW TABLES; -- I find out about the table userdb
```

```
DESCRIBE table_name; -- show table column details
```

```
SELECT * FROM table_name; -- I query the table userdb
```

Now I have full control over the database, but to have full control over the ENTIRE app, I need to login as admin. So I query the usersdb table with

```
SELECT * FROM userdb WHERE email LIKE '%admin%'
```

and, surprise surprise, the admin was right there (with email 'admin'), but with the password **encrypted** (not bad). Well, by simply looking at that encryption, I realized that it is not such an extremely strong algorithm that did that (also no salts and peppers 😊).

Therefore, I just went to crackstation.net and pasted the encrypted password there. It cracked it immediately (hash type was md5), and the result was : **secret**. Funny simple passwords, *c'est la vie*.

Silly admin. I now 'own' the entire BadStore web service.

Mitigation

One of the methods that I prefer in order to prevent these attacks is **sanitization by prepared statements**, which decouples the code and the data. Example:

```
$statement = $db->prepare("select * from Users where(name=? and password=?);");  
$statement->bind_param("ss", $user, $pass);  
$statement->execute();
```

In the end, I'll just leave some SQL Injection humor to fill the blank of this page.

