

Cross-Site Scripting: Lab

Student worksheet · OWASP Juice Shop

You will make the page run *your* JavaScript, from a search box, from a link and from the database.

Open **Juice Shop**: `http://localhost:3000`, and the **Score Board**: `http://localhost:3000/#/score-board`. Each solved challenge turns green there.

Rule: do this only on our own Juice Shop. Injecting a script into a site you do not own is an offence under the IT Act.

One warning before you start: every payload uses back-ticks around `xss`: `alert(`xss`)`. That is the grave accent ` (top-left of most keyboards), *not* a straight quote. **Copy the payloads, do not retype them**, or the attack will not fire.

1. DOM XSS (1 star)

1. Click the magnifier (top right) to open the search box.
2. Paste exactly, and press Enter:

   ```
   <iframe src="javascript:alert(`xss`)">
   ```

3. A small box saying `xss` pops up. That is JavaScript running. The Score Board shows **DOM XSS** solved.

Why: the search page writes your text into the page as *HTML*, so your `<iframe>` becomes a real element and its `javascript:` code runs. Text that should be data was treated as code.

If stuck: No box: you typed straight quotes instead of back-ticks, or you retyped it. Paste the payload. Make sure you pressed Enter in the search box, not the address bar.

2. Bonus Payload (1 star)

1. On the Score Board, open the **Bonus Payload** challenge and copy the long `<iframe ...>` shown in its description (it is a SoundCloud player).
2. Paste it into the search box and press Enter. A music player appears and plays. (Turn your volume down first.) The Score Board shows **Bonus Payload** solved.

Why: same hole as DOM XSS, a bigger payload: if an attacker can load one `<iframe>`, they can load anything, here a whole media player.

3. Reflected XSS (2 stars)

The order-tracking page writes the order id from the URL straight onto the page, so a script put in the id will run.

1. First see a normal result: open `http://localhost:3000/#/track-result?id=test123`.
2. Now replace the id with the payload. Copy this **whole line** into the address bar and press Enter:

   ```
   http://localhost:3000/#/track-result?id=<iframe src="javascript:alert(`xss`)">
   ```

3. The `xss` box pops up. The Score Board shows **Reflected XSS** solved.

Why: the order id comes straight from the URL and is written onto the page. Unlike DOM XSS, the text rides in on a link: an attacker emails that link, and whoever clicks it runs the script in their browser.

If stuck: Type or paste the payload as **one unbroken line**, from `<iframe` to `>`; if it wraps when you copy it, the JavaScript breaks. The back-ticks around `xss` are the grave accent ‘ (top-left key), not straight quotes.

4. API-only XSS (3 stars)

This time the script is *stored*: you write it into a product description in the database, and every visitor who looks at that product runs it. The challenge says: do it *without using the shop's web pages*, so you talk to its REST API directly.

1. See the product as the API returns it: open `http://localhost:3000/api/Products/1`. Note the `"description": The all-time classic.`
2. On the Juice Shop tab, press **F12** to open the developer tools, and click the **Console** tab. If Chrome warns you about pasting, type `allow pasting` and press Enter. Now paste this and press Enter:

   ```
   fetch('/api/Products/1',{method:'PUT',headers:{'Content-Type':'application/json'},
     body:JSON.stringify({description:'<iframe src="javascript:alert(`xss`)">'})})
   ```

   We do this in the browser so `JSON.stringify` builds the request body for us: no quotes to escape and nothing to break when you paste.
3. The Score Board shows **API-only XSS** solved. Reload the page from step 1: your iframe is now the description.
4. Now be the victim: open `http://localhost:3000`, and click the **Apple Juice** product. The `xss` box pops up.

The console is only the easy way in: any tool that sends an HTTP request works, since the challenge just needs a `PUT` to `/api/Products/1`. With `curl`, keep the `-d` line whole (the JSON body must not wrap, or you get a `Bad control character` error):

```
curl -X PUT http://localhost:3000/api/Products/1 \
  -H "Content-Type: application/json" \
  -d '{"description":"<iframe src=\"javascript:alert(`xss`)\">"}'
```

Why: two mistakes together. The API let you change a product without logging in (broken access control), and the page shows descriptions as trusted HTML. Unlike the first three, nobody has to type or click anything special: the code waits in the database and runs for *every* visitor.

If stuck: Nothing happens, or you see a JSON error: the back-ticks around `xss` are the grave accent ` (top-left key), not straight quotes, and you must paste both lines, not retype them. Run it on the Juice Shop tab itself (not a blank tab), so the `/api/...` address resolves.

What you did

Four times you made the browser run text as code: from the search box (DOM XSS), with a bigger payload that embeds a media player (Bonus Payload), through a link (Reflected XSS), and from the database, for every visitor (API-only XSS). The cause is always user text treated as HTML. The fix, next, is to escape the text so it is shown as text, never run as code.