EECS 388

Introduction to Computer Security

Lecture 8:

Web Attacks and Defenses

September 19, 2024

Prof. Chen

Web Security

This week:

- The Web Platform
- **Web Attacks and Defenses**

Next week:

- HTTPS and the Web PKI
- HTTPS Pitfalls

Later in the course:

- User Authentication
- Privacy and Online Tracking

Three Classic Web Attacks

Cross-Site Request Forgery (CSRF)

SQL injection (SQLi)

Cross site scripting (XSS)

You’ll exploit all three in Project 2

Review: Same-Origin Policy

Essential security question:

When can one site access data contained in another site?

Example:

If you visit **attacker.com**, what stops it from reading your Gmail messages?

What if **attacker.com** loads Gmail in a frame or runs JavaScript files from **gmail.com**?

Browsers enforce isolation between sites by applying **Same-Origin Policy** (SOP).

The SOP separates content into different trust domains ("**origins**") and restricts data flows between them.

What defines an origin?

scheme://domain:port

example: **https://eecs388.org:443**

What's isolated?

Each origin has local client-side resources that are protected:

- Cookies (local state)
- DOM storage
- DOM tree
- JavaScript namespace
- Permission to use local hardware (e.g., camera or GPS)

Review: Cookie Sending

Your browser contains these cookies:

1) domain: **bank.com** AuthToken=012...
2) domain: **login.bank.com** TrackingID=248e...
3) domain: **attacker.com** VictimID=456...

Which cookies does your browser send when...?

a) You visit **bank.com** . . . . Cookie 1

b) A page on **bank.com** contains
`<img src="https://bank.com/img/logo.png">` . . . Cookie 1

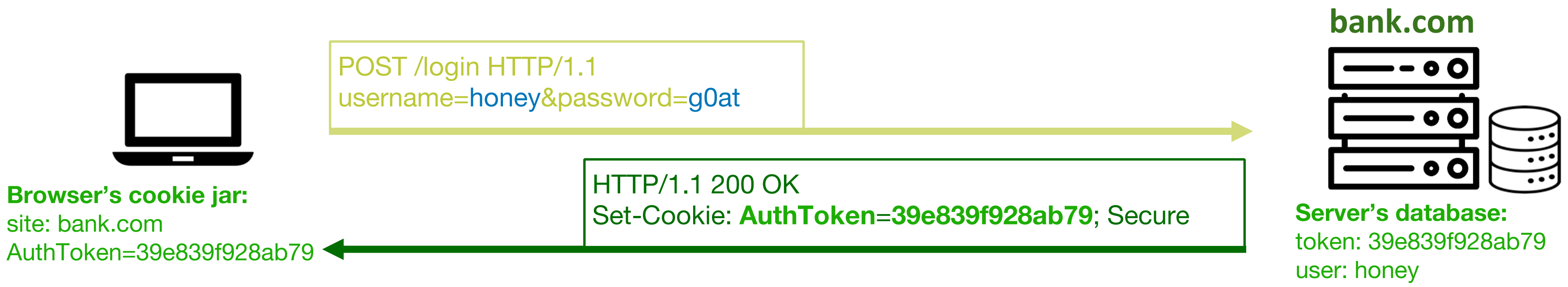
c) You visit **attacker.com** . . . . Cookie 3

d) A page on **attacker.com** contains
`<img src="https://bank.com/img/logo.png">` . . . Cookie 1

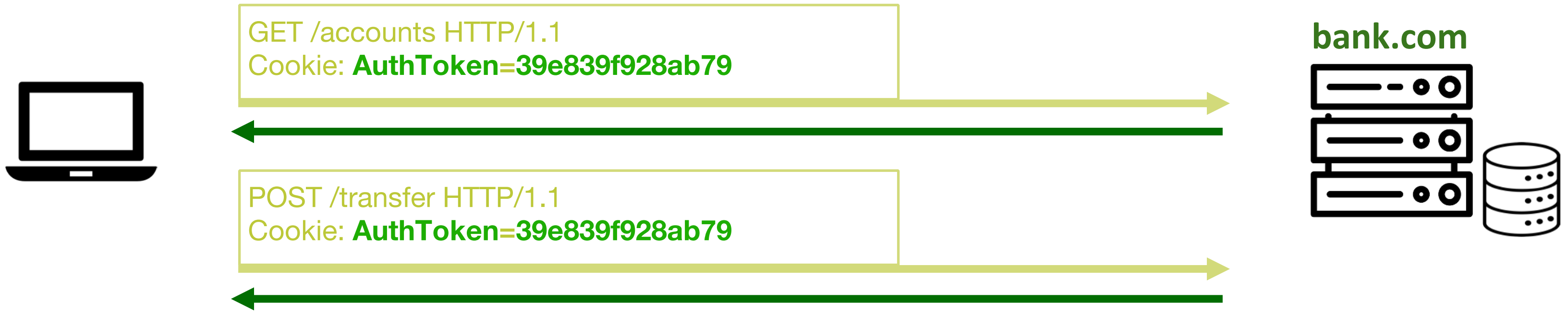
Cookies sent by browser are determined by the domain of the **resource being requested**

Cookie-Based Authentication

Upon successful login, server sets a cookie with an unguessable random value, the **authentication token**. Server DB stores the token, username, and expiry time



In later requests, browser present the authentication cookie. Server validates via DB

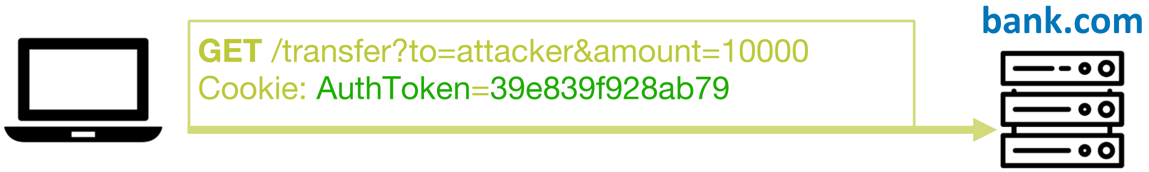


CSRF Attack

Cross-Site Request Forgery (CSRF) attacks cause the user's browser to perform unwanted actions on a different site on the user's behalf

Example: User visits **attacker.com**

```
<html>
 <img src="//bank.com/transfer?to=attacker&amount=10000">
</html>
```



If user is logged into the bank site, browser sends user's valid AuthToken cookie to **bank.com** along with the request

Good News!
attacker.com can't read the **bank.com** AuthToken cookie (due to SOP)

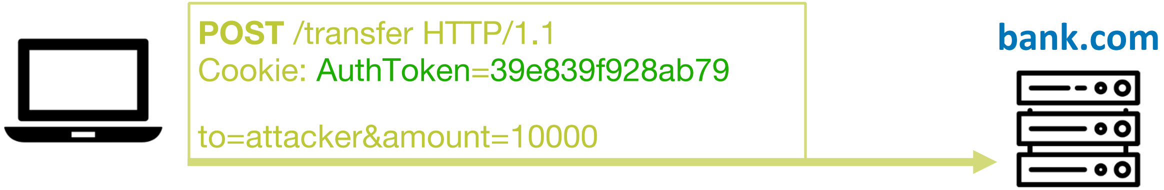
Bad News!
Your money is gone

CSRF via POST Request

What if **bank.com/transfer** endpoint only allowed HTTP **POST requests**?

Example: User visits **attacker.com**

```
<form name=f method=post action="//bank.com/transfer">
  <input type="hidden" name="to" value="attacker">
  <input type="hidden" name="amount" value="10000">
</form>
<script>document.f.submit();</script>
```



Attacker can trigger a POST request using HTML and JS. Like in other requests, the browser sends cookies that match the domain of the target resource (i.e., **bank.com**)

Good News!
attacker.com still can't read AuthToken cookie or POST response from **bank.com**

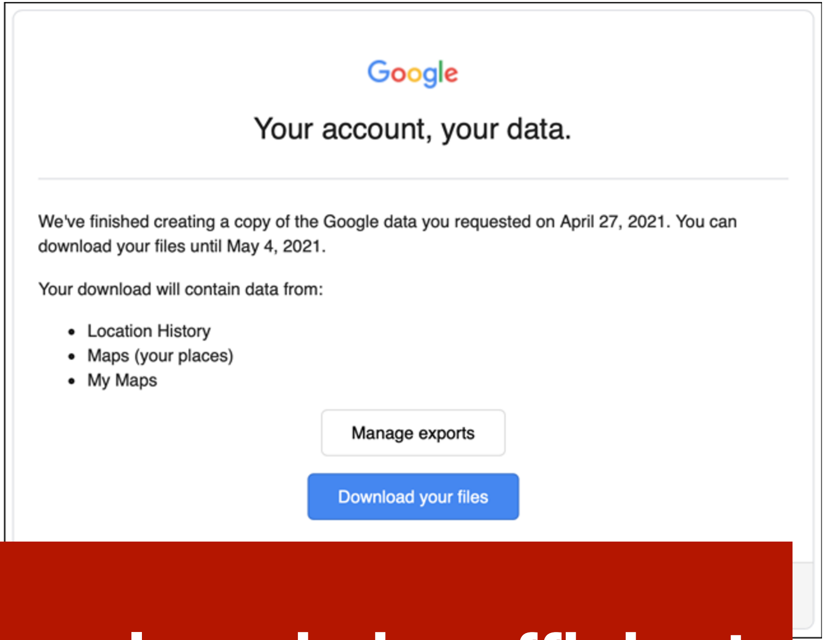
Bad News!
Your money is gone

Login CSRF Attack

What if a site's login form
isn't protected against CSRF?

Login CSRF attack can log in victim's browser to an honest site with an **account controlled by the attacker**

[Examples of harm this can do?]


Google

Your account, your data.

We've finished creating a copy of the Google data you requested on April 27, 2021. You can download your files until May 4, 2021.

Your download will contain data from:

- Location History
- Maps (your places)
- My Maps

Manage exports

Download your files

Cookie-based authentication alone is insufficient for requests that have any side effects

CSRF Defenses

CSRF attacks rely on the fact that cookies are attached to any request to a given domain, **no matter which origin initiates the request**.

Need some mechanism to ensure requests are authentic
(i.e., initiated by a trusted page).

Options:

- Referer validation
- Secret token validation
- SameSite cookies

CSRF Defense: Referer Validation

The `Referer` [*sic*] **HTTP request header** contains URL* of page making the request (or page from which link to current page was followed). Allows sites to identify where visitors are coming from

* For privacy, modern browsers send only the domain on cross-origin requests

Our goal: Authenticate that each user action originates from our site

`Referer: https://`**`bank`**`.com` → ✓

`Referer: https://`**`attacker`**`.com` → X

`Referer: [omitted]` → ?

bank.com

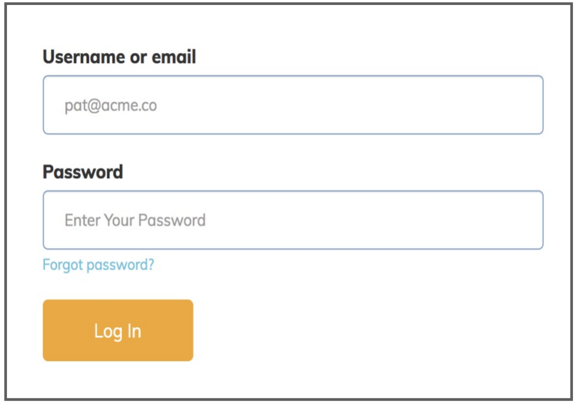
Complication:
`Referer` not always sent.
Users can turn it off for privacy.
Attacker.com can disable it.
Not sent from bookmark or URL bar.

What to do when it’s not?

CSRF Defense: Secret Token Validation

Pages served by the site **embed a secret value in each request**, server validates it.

bank.com



Every form contains a secret value that the server validates:

```
<form action="https://bank.com/login" method="post">
  <input name="csrf_token" type="hidden"
value="434ec7e838e">
  <input name="login"  type="text">
  <input name="password" type="password">
  <button type="submit">Log In</button>
</form>
```

Caution: *Static* tokens provide no protection! (Attacker can simply look them up)

Must use a **session-dependent** token, typically tied to a session cookie.
(Attacker cannot *retrieve* the cookie due to SOP)

CSRF Defense: SameSite Cookies

`SameSite` attribute prevents browser from sending cookie in cross-site requests:

```
Set-Cookie: AuthToken=X; Secure; SameSite=Lax
```

`SameSite=Strict` Cookie isn't sent in any cross-site context, even when following a regular link.
E.g., if a logged-in user follows a link to a private GitHub project from Gmail, GitHub will not receive the session cookie and the user will not be able to access the project without further clicks.

`SameSite=Lax` Cookie *is* sent when navigating to cross-site links, but not cross-site POSTS/subrequests
E.g., Following links to GitHub works as expected, but the cookie will not be sent if a third-party site POSTs to GitHub, loads GitHub script or images, or embeds GitHub pages in an iframe, etc.

Good news!

Popular browsers have switched to setting **SameSite=Lax** by default.

But...**Lax** only prevents some cases of CSRF

CSRF Summary

Cross-Site Request Forgery (CSRF) attacks cause the user's browser to perform unwanted actions on a different site on the user's behalf (typically, but not always, where the user is already logged in)

CSRF exploits the trust that a site has in a user's browser

CSRF attacks specifically target state-changing requests, not data theft, since the attacker *cannot see the response* to the forged request due to the SOP

Defend against CSRF using a combination of:

- **Secret Validation Tokens**
- **SameSite Cookies**

Three Classic Web Attacks

Cross-Site Request Forgery (CSRF)

SQL injection (SQLi)

Cross site scripting (XSS)

Injection Attacks

Injection attacks exploit vulnerabilities that **mistake untrusted data for code**, allowing specially crafted inputs to cause execution of malicious instructions

[*What's the difference between code and data?*]

Types of injection attacks:

- **SQL Injection** Data changes meaning of SQL statements
- **Cross-site Scripting** Data changes HTML and JS on web page
- **Shell Injection** (later) Data executes shell script commands
- **Control Hijacking** (later) Data injects new machine code

Structured Query Language (SQL)

Structured Query Language (SQL) is a domain-specific language used for managing databases.

Powerful and ubiquitous, SQL can be used directly (e.g., **`sqlite3`** command) or from inside other code

```
$ sudo apt install sqlite3
$ sqlite3
Enter ".help" for usage hints.
Connected to a transient in-memory database.
sqlite> .headers on
sqlite> .mode column

sqlite> CREATE TABLE users (id INT, username VARCHAR, password VARCHAR);

sqlite> INSERT INTO users VALUES (1, 'honey', 'g0at');
sqlite> INSERT INTO users VALUES (2, 'paulgrub', 'sw0rdf!sh');
sqlite> INSERT INTO users VALUES (3, 'ensafi', 'j0shu@');

sqlite> SELECT * FROM users;
id          username    password
----------  ----------  ----------
1           honey       g0at
2           paulgrub    sw0rdf!sh
3           ensafi      j0shu@

sqlite> SELECT id FROM users WHERE username=honey AND password='g0at';
1
```

SQL Injection

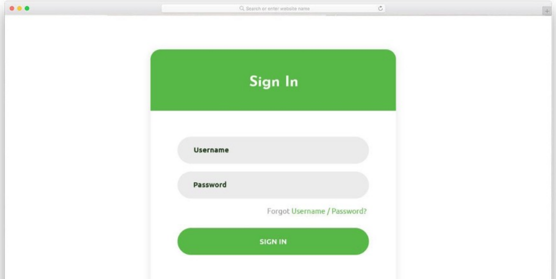
SQL injection (SQLi) vulnerabilities occur when a program **passes unsanitized inputs into SQL database statements**

Despite being easy to avoid, they are a common and dangerous mistake.

Example: Vulnerable Login Form

```
$user = $_POST['username'];
$pass = $_POST['password'];
$sql = "SELECT * FROM users WHERE
   username = '$user' AND password = '$pass'";
$rs = $db->executeQuery($sql);
if $rs.count > 0 {
     // login success
}
```

users		
username	password	id
honey	g0at	1
paulgrub	sw0rdf!sh	2
ensafi	j0shu@	3

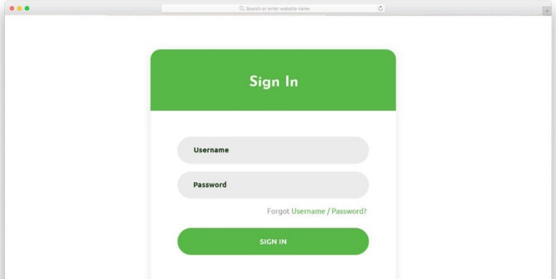


SQLi Example: Vulnerable Login Form

When a normal user logs in:

```
$user = $_POST['username']; $pass = $_POST['password'];
honey                            g0at
$sql = "SELECT * FROM users WHERE
  username = '$user' AND password = '$pass'";
SELECT * FROM users WHERE
  username = 'honey' AND password='g0at'
$rs = $db->executeQuery($sql);
[{username: "honey", password: "g0at", id: 1}]
if $rs.count > 0 {
     // success
     Yep!
}
```

users		
username	**password**	**id**
honey	g0at	1
paulgrub	sw0rdf!sh	2
ensafi	j0shu@	3



SQLi Example: Vulnerable Login Form

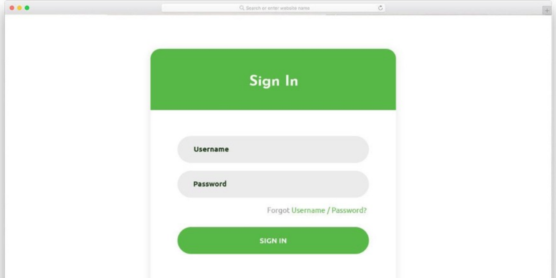
What if the untrusted input contains **special characters**?

```
$user = $_POST['username']; $pass = $_POST['password'];
honey                       g0at'
$sql = "SELECT * FROM users WHERE
  username = '$user' AND password = '$pass'";
SELECT * FROM users WHERE
  username = 'honey' AND password='g0at''
$rs = $db->executeQuery($sql);
```

SQL syntax error!

users		
username	password	id
honey	g0at	1
paulgrub	sw0rdf!sh	2
ensafi	j0shu@	3

Often the easiest test for SQLi is to enter a **single quote (')** as part of the data, to check whether program constructs the SQL statement without properly **sanitizing** the input



SQLi Example: Vulnerable Login Form

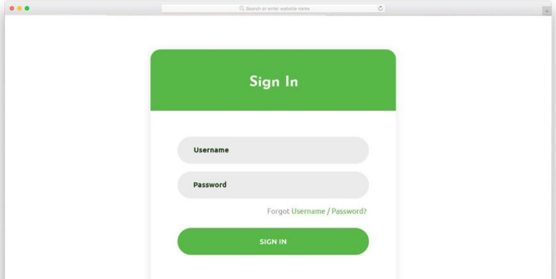
What if the untrusted input contains **special characters**?

```
$user = $_POST['username']; $pass = $_POST['password'];
honey                          g0at'--
$sql = "SELECT * FROM users WHERE
  username = '$user' AND password = '$pass'";
SELECT * FROM users WHERE
  username = 'honey' AND password='g0at'--'
$rs = $db->executeQuery($sql);
[{username: "honey", password: "g0at", id: 1}]
if $rs.count > 0 {
     // success
     Now there's
     no error
}
```

-- begins SQL comment

Using **--** starts a comment and "consumes" the final quote provided by the application (and any later part of the original SQL)

users		
username	**password**	**id**
honey	g0at	1
paulgrub	sw0rdf!sh	2
ensafi	j0shu@	3



SQLi Example: Vulnerable Login Form

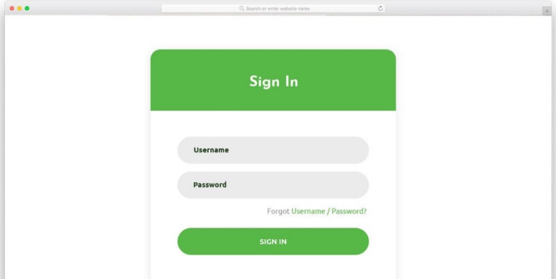
Crafting a **malicious input** to log in **without knowing the password**:

```
$user = $_POST['username']; $pass = $_POST['password'];
honey                                '--
$sql = "SELECT * FROM users WHERE
  username = '$user' AND password = '$pass'";
SELECT * FROM users WHERE
  username = 'honey' AND password=''--'
$rs = $db->executeQuery($sql);
[]
if $rs.count > 0 {
     // success
}
```

No matches, since provided password is treated as empty

Fails to reach here, since no records matched

users		
username	password	id
honey	g0at	1
paulgrub	sw0rdf!sh	2
ensafi	j0shu@	3



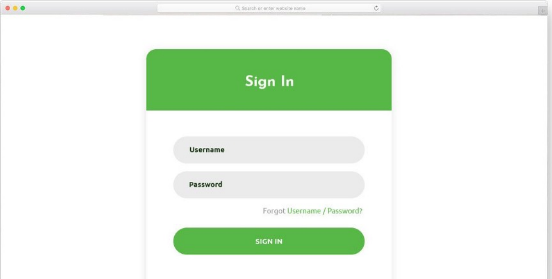
SQLi Example: Vulnerable Login Form

Crafting a **malicious input** to log in **without knowing the password**:

```
$user = $_POST['username']; $pass = $_POST['password'];
honey                          ' OR 1=1 --
$sql = "SELECT * FROM users WHERE
  username = '$user' AND password = '$pass'";
SELECT * FROM users WHERE
  username = 'honey' AND password='' OR 1=1 --'
$rs = $db->executeQuery($sql);
[{username: "honey", password: "g0at", id: 1},
 {username: "paulgrub", password: "sw0rdf!sh", id: 2},
 {username: "ensafi", password: "j0shu@", id: 3}]
if $rs.count > 0 {
   Yay! Pwned!
}
```

The **OR 1=1** clause causes the SELECT statement to match every record

users		
username	**password**	**id**
honey	g0at	1
paulgrub	sw0rdf!sh	2
ensafi	j0shu@	3

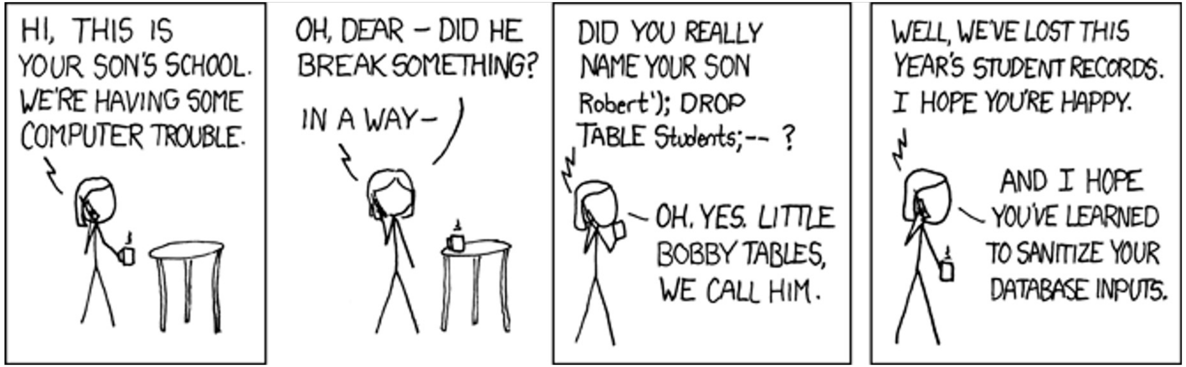


Causing Damage with SQLi

```
$user = $_POST['username'];
   '; DROP TABLE users--
$sql = "SELECT * FROM users WHERE username = '$user'";
   SELECT * FROM users WHERE username = ''; DROP TABLE users--'
$rs = $db->executeQuery($sql);
```

Note: Not all libraries allow compound statements as shown here, but there can be other ways to nest statements…

The entire users table is gone!



users

username	password	id
honey	g0at	1
paulgrub	sw0rdf!sh	2
ensafi	j0shu@	3

Preventing SQL Injection

Is it sufficient to escape or filter out single quotes?

No! Consider integer values... `SELECT password FROM users WHERE id = $n`

Correct approach:

Avoid building SQL commands yourself at runtime.

Options:

- Parameterized (a.k.a. prepared) SQL statements
- ORM (Object Relational Mapper):
  language-specific methods for accessing data using native code

SQLi Defense: Parameterized SQL

Parameterized SQL provides query and arguments separately, avoiding code/data confusion.

```
sql = "SELECT * FROM users WHERE username = ?"   // Statement parsed with argument stubs
cursor.execute(sql, [honey])                     // Literal values are provided later
```

```
sql = "INSERT INTO users(username, password) VALUES(?,?)"
cursor.execute(sql, [ensafi, 'j0shu@'])
```

Benefit: Data *cannot* change semantic meaning of the statement. No need to sanitize input

Extra Benefit: Parameterized queries can be *faster,* because server can cache query plan

Three Classic Web Attacks

Cross-Site Request Forgery (CSRF)

SQL injection (SQLi)

Cross site scripting (XSS)

Cross Site Scripting (XSS)

Cross Site Scripting (XSS) attacks exploit sites that send untrusted inputs to browsers without proper validation or sanitization

SQL Injection

attacker's malicious **SQL code** is executed on victim's **server**

Cross Site Scripting

attacker's malicious **JS code** is executed on victim's **browser**

Types of XSS

An XSS vulnerability is present when an attacker can inject script into pages generated by a web application

Two Types:

Reflected XSS echoes script back to the same user in the context of the site

Stored XSS stores malicious code in a resource managed by the server, such as a database, where it can target other users

Reflected XSS

Vulnerability: Site echoes inputs back to user without properly escaping them

Exploitation example: User follows malicious link to the site, link causes attacker-provided script to execute (in the user's authentication context)

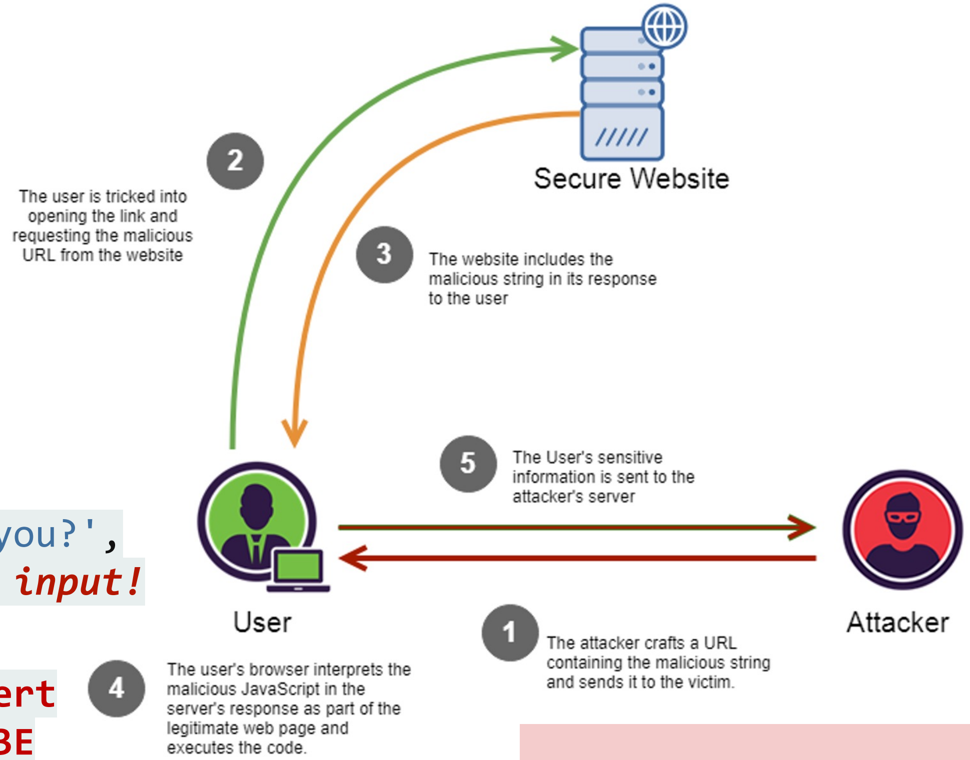
```
@route('/hello/<name>')
def greet(name):
    return f'Hello {{name}}, how are you?',
      name=name) # Echoes unsanitized input!
```

If user follows link to:

```
https://site.com/hello/%3Cscript%3Ealert
%28document.cookie%29%3B%3C%2Fscript%3E
```

Page on **site.com** will contain code from the link:

```
Hello <script>alert(document.cookie);</script>, how are you?
```



Attacker's code executes in site's origin!

Reflected XSS Example

Danger: If ***any page***, anywhere on the site, has a reflected XSS vulnerability, can exploit it to compromise ***any data*** accessible to code within the site's origin

Example: In February 2021, attackers contacted PayPal users via email and fooled them into accessing an obscure URL hosted on the legitimate PayPal website.

A reflected XSS vulnerability in that page allowed the link to inject code that generated a warning that the user's account had been compromised.
It appeared to come from the real Paypal site.

Victims were then redirected to a phishing site and prompted to enter sensitive financial data.

Stored XSS

Vulnerability: Site **stores and displays** user content without properly escaping it

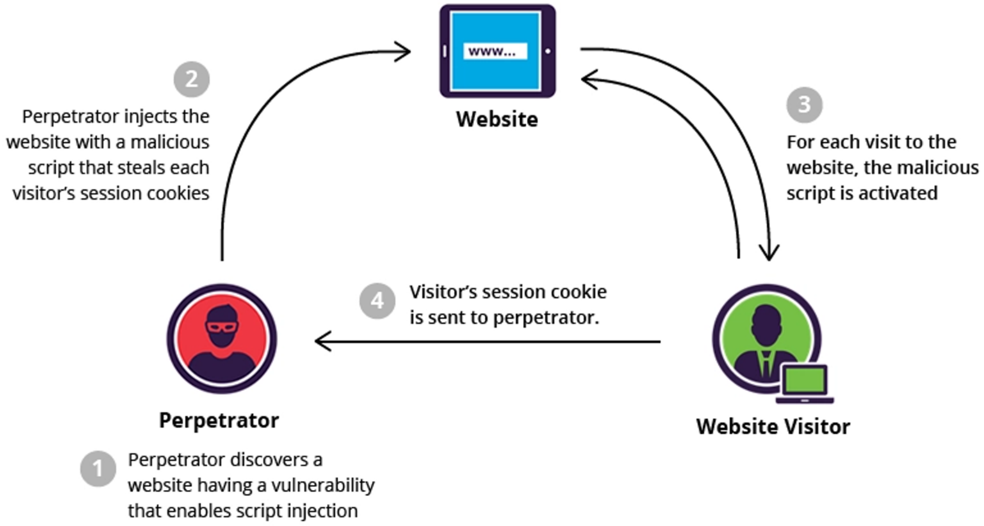
Exploitation example: Attacker uploads content that site shows to ***other users***. Their browsers load and execute the code (in their authentication contexts)

```
@route('/profile/<user>')
def show_bio(user):
    profile = get_user_profile(user)
    return f'<p>{{bio}}</p>',
      bio=profile.bio) # Unsanitized!
```

Attacker sets their bio to:

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

This code will be executed by every user who views the attacker's profile!



Code injected via XSS can contain arbitrary malicious **payloads**: Steal data and send it to the attacker, perform actions as the user, etc.

Stored XSS Example: Samy Worm

2005: ***XSS worm*** that spread on MySpace

When a user visited Samy Kamkar's profile, script he injected would:

- add the string "*but most of all, samy is my hero*" to the victim's MySpace profile page
- send Samy a friend request, and
- install the worm itself in the victim's profile, so anyone who viewed *their* profile also got infected

In 20 hours, it spread to >1 million users!

(Samy was quickly arrested and prosecuted.)

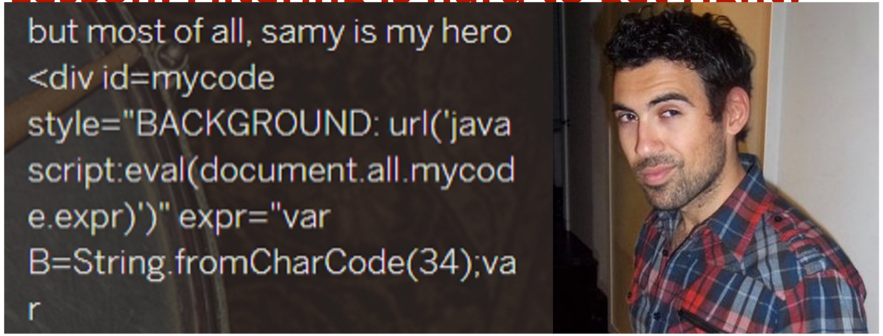
The problem: MySpace allowed users to post HTML to their profiles…

…it correctly filtered out <script>, <body>, onclick, <a href=javascript://>…

…but you can also run JS inside of CSS tags:

```
<div
style="background:url('javascript:alert(1)')">
```

Lesson: Filtering is hard to get right!



XSS Defense: Validation and Escaping

For a long time, the only way to prevent XSS attacks was to try to filter out malicious content. Two approaches used in tandem:

Input validation: Checks all headers, cookies, query strings, form fields, and hidden fields (i.e., all user-controlled parameters that might appear in output) against a rigorous specification of what should be allowed.

Output escaping: Encodes all special characters in output to prevent interpretation as code.

Adopt a “positive” security policy that specifies what is *allowed*.
“Negative” or attack signature-based policies (like MySpace used) are difficult to maintain and likely to be incomplete

XSS Defense: Content Security Policy

Content Security Policy (CSP) is a more modern approach that allows sites to eliminate XSS by tightly specifying what scripts are allowed to execute.

Site serves policy via an **HTTP header**. Browsers will only execute scripts loaded in source files received from specified domains. Inline scripts are prohibited.

Example Policy: Script files can only be loaded from the domain itself

```
Content-Security-Policy: default-src 'self'
```

Example Policy:

- include images from any origin, but
- restrict audio or video media to specific trusted providers, and
- only allow scripts from a one server that hosts trusted code

```
Content-Security-Policy: default-src 'self'; img-src *;
            media-src media1.com; script-src userscripts.example.com
```

Coming Up

Reminders:

Crypto Project, Part 2 due TODAY at 6 PM

Web Project available now, due in two weeks

Monday

HTTPS

HTTP over a secure (TLS) channel

Certificates and the CA ecosystem

Wednesday

Attacking HTTPS

Implementation flaws, social engineering, crypto failures