CSE127, Computer Security

Web Security II

UC San Diego

Housekeeping

General course things to know

- PA2 was due
  - Grades coming this week
- PA3 is out! Due on **2/10**, note the bump up w/ midterm
- Midterm is **2/12,** during class hours, location **Center Hall 109**
  - Class topics will go through web security (2/5) and include PA3 material
  - One sheet of paper front and back is allowed as a “cheatsheet”
  - You must bring photo ID to the exam!

Previously on CSE127…

Webs of webs

- We started talking about the **web**
  - The basic interaction model, HTTP, cookies, etc.
- We learned the web lives on *top* of the Internet and is primarily a mechanism to connect documents together
  - But today, websites can do lots of things… run code, interface with OS, etc.
  - **Big attack surface**

Today's lecture — Web Security

Learning Objectives

- Understand the basic browser execution model, the DOM, and the idea that websites are programs
- Learn about the Same-Origin Policy — the fundamental security policy that runs all of the web
- Understand the concepts of three common web attacks:
  - Injection attacks
  - CSRF attacks
  - XSS attacks

Web Execution

Recall from last time…

Client / Server Model

Client

Web Server

Recall from last time…

Client / Server Model

Where in these components is there code running?

Client

Web Server

Recall from last time…

Client / Server Model

Where in these components is there code running?

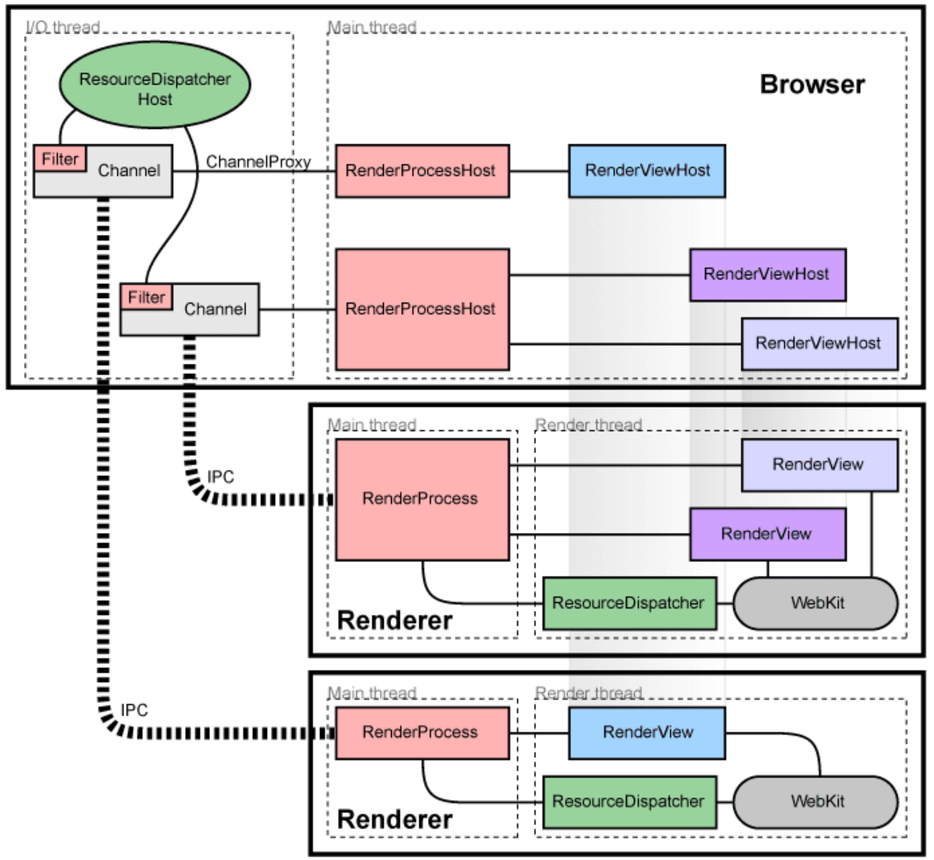
Client

Code is running on the **server** (e.g., Python, Go, C), on the **client** (e.g., Chrome, Firefox), and *in the client* (e.g., JavaScript)

Web Server

Browser execution model

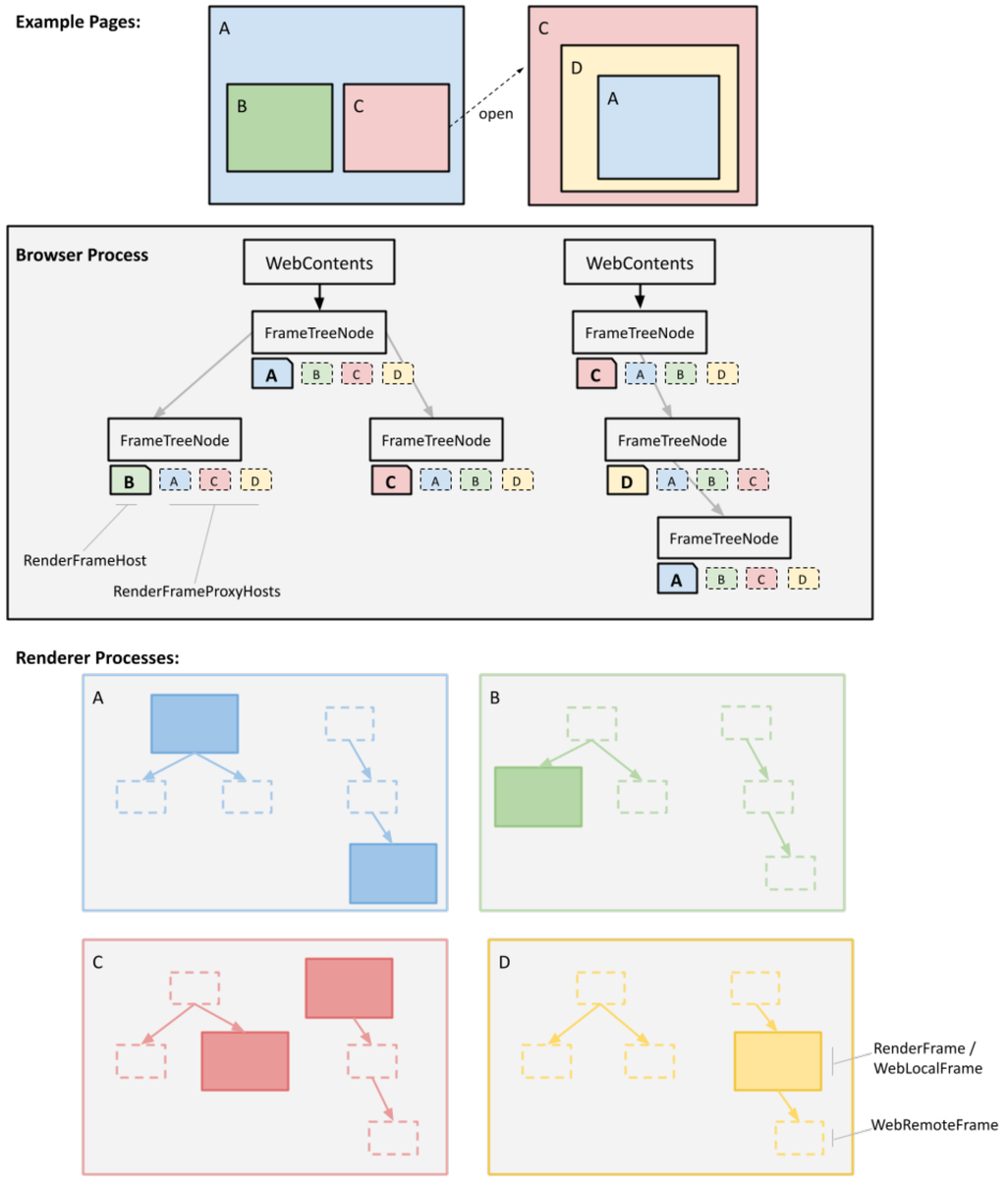
- Browsers use an inordinate amount of processes to handle modern websites
- Each browser window / tab…
  - Loads and renders content
  - Parses HTML and runs JavaScript
  - Fetches subresources (e.g., images, CSS, JavaScript)
  - Responds to events like onClick, onMouseover, onLoad, onTimeout



https://www.chromium.org/developers/design-documents/multi-process-architecture/

Nested execution model

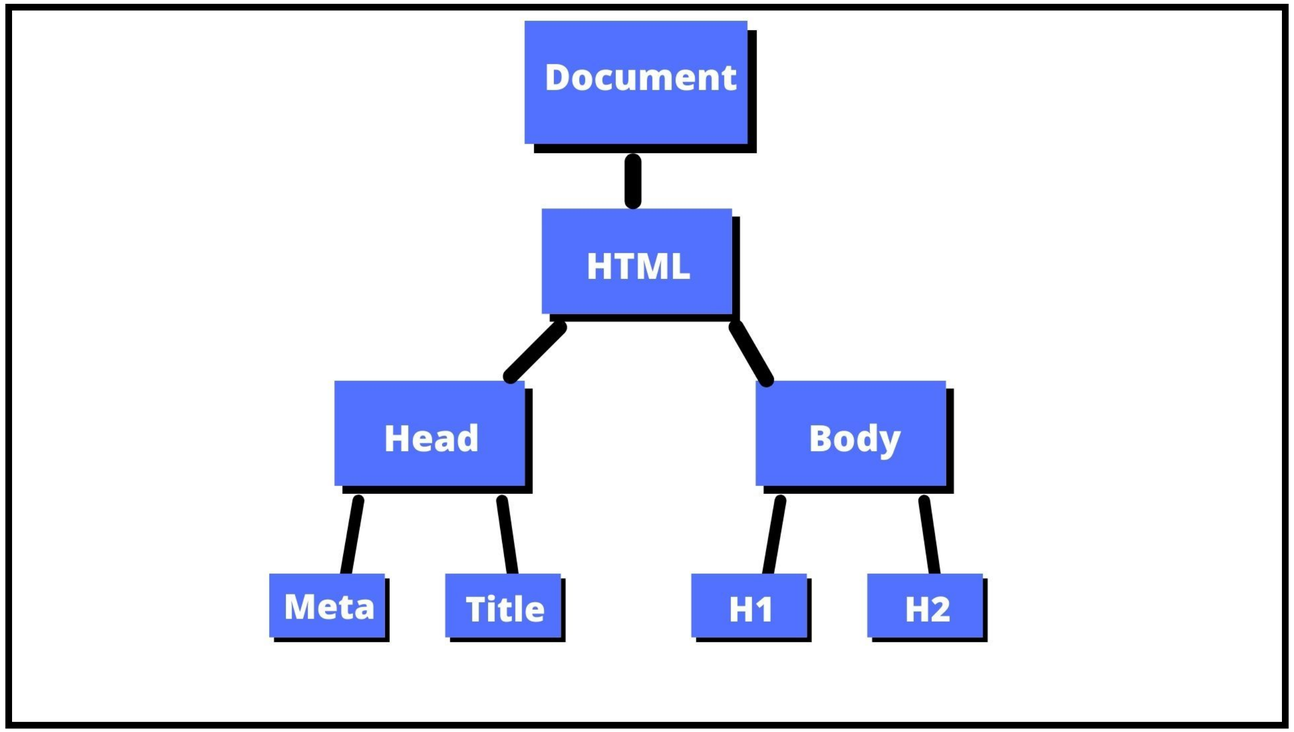
- Websites may contain frames from other sources
  - Frame: rigid, visible division
  - iFrame: floating, inline frame
- Why use frames?
  - Delegate screen area to content from another source (e.g., ads)
  - Browser provides **isolation** based on frame (remember site isolation?); each frame gets its own rendering process



https://www.chromium.org/developers/design-documents/oop-iframes/

What is the layout of webpages?

- Webpages follow a Document Object Model (DOM) — this is the structure of the page itself (encoded via HTML, sometimes XML)
  - The page itself is a *tree* of *nodes* designated by *tags* (e.g.,)



Can I modify the layout on the fly?

Can I modify the layout on the fly?

- Yes!
- The DOM can be manipulated via **code** or **scripts** included on the page
  - AKA: everything you see can be rendered dynamically
- Even the *browser* can be manipulated via code
  - Code can change your window, move you back and forward through browsing history, read cookies… *anything*

Where is JavaScript running?

- Chrome has developed its own high-performance JavaScript (and wasm) engine in C++
- If you've ever used `node.js`, it's the same underlying engine
- ~ 2.3 million lines of C++ code that is under constant development…
  - Seems totally secure?

https://www.chromium.org/developers/design-documents/multi-process-architecture/

Where is JavaScript running?

BUGS, NEWS

Chrome zero-day under active attack: visiting the wrong site could hijack your browser

by Pieter Arntz | November 18, 2025

CVE-2025-10585 Vulnerability: A New Zero-Day Exploit in Chrome's V8 JavaScript and WebAssembly Engine Weaponized in Real-World Attacks

https://socprime.com/blog/cve-2025-10585-zero-day-vulnerability/

Always remember: websites are programs!

- Partially executed on the client side
  - HTML rendering, JavaScript, extensions, etc.
- Partially executed on the server side
  - Python, CGI, PHP, ASP, server-side JavaScript (ew), etc.
- And programs, as we know, can have lots of problems…

So many problems!

- Code is running in a lot of places, it can be hard to keep track of it all
    - And much code is constantly crossing trust boundaries (e.g., the browser has to safely run user generated JavaScript)
- Many opportunities to confuse the server and client about where data is coming from, what data it can trust, and what it should do about that data
- **How do we handle security on the web?**

Same Origin Policy

Web Threat Model

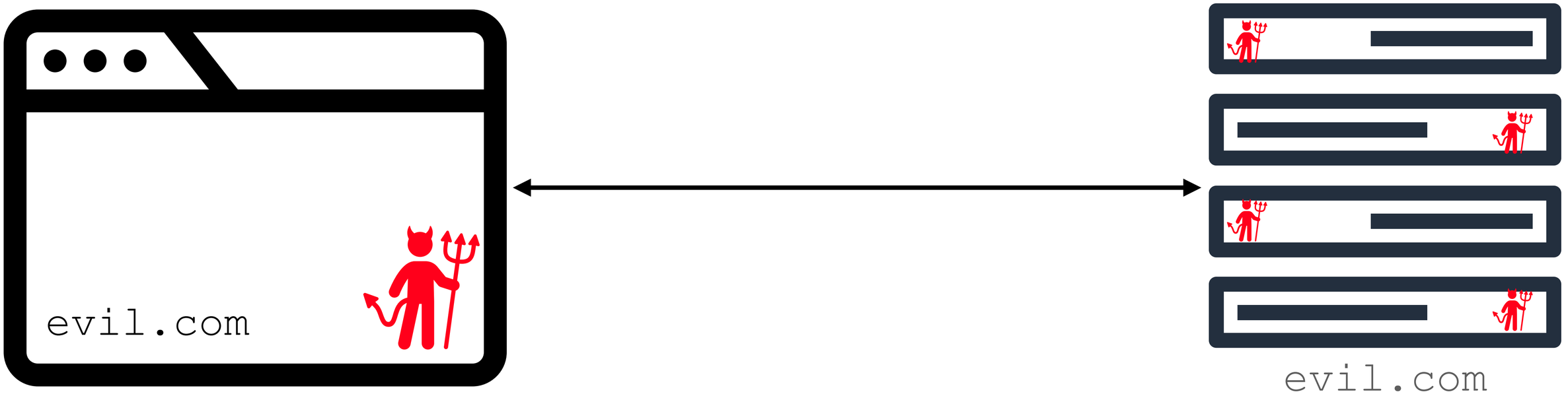
- Let's say you're a browser developer, and you're trying to run two different websites at the same time. **What outcomes are you trying to prevent?**

Web Threat Model

- Let's say you're a browser developer, and you're trying to run two different websites at the same time. **What outcomes are you trying to prevent?**
  - Evil websites *reading* private information from other websites (e.g., cookies)
  - Evil websites *modifying* content on other non-evil pages
  - Evil websites *making requests* on behalf of non-evil websites

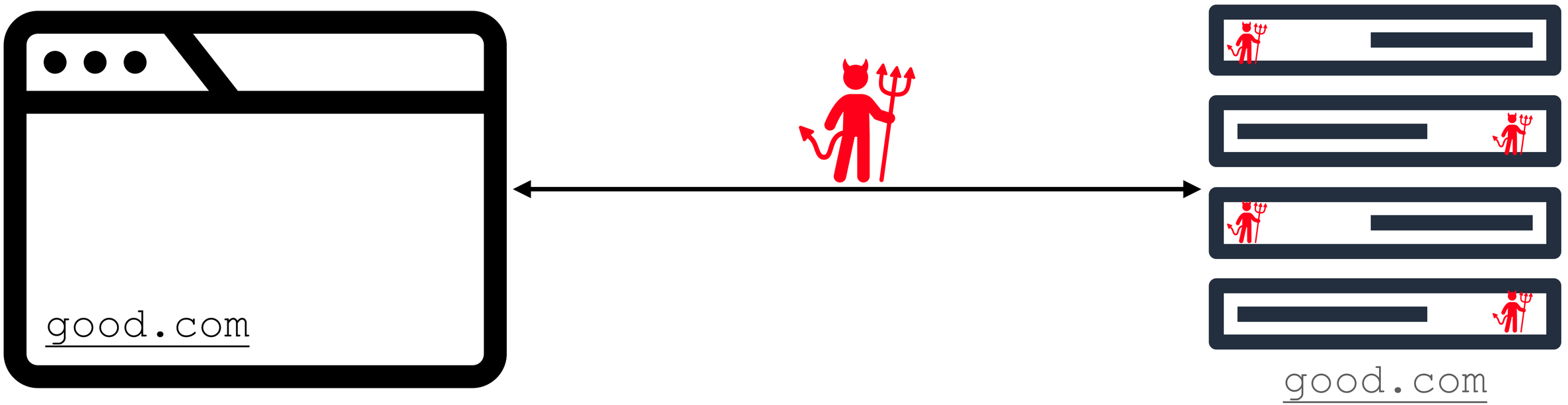
Web Attacker Models

- Several types of attackers on the web… our main focus is the **web attacker**



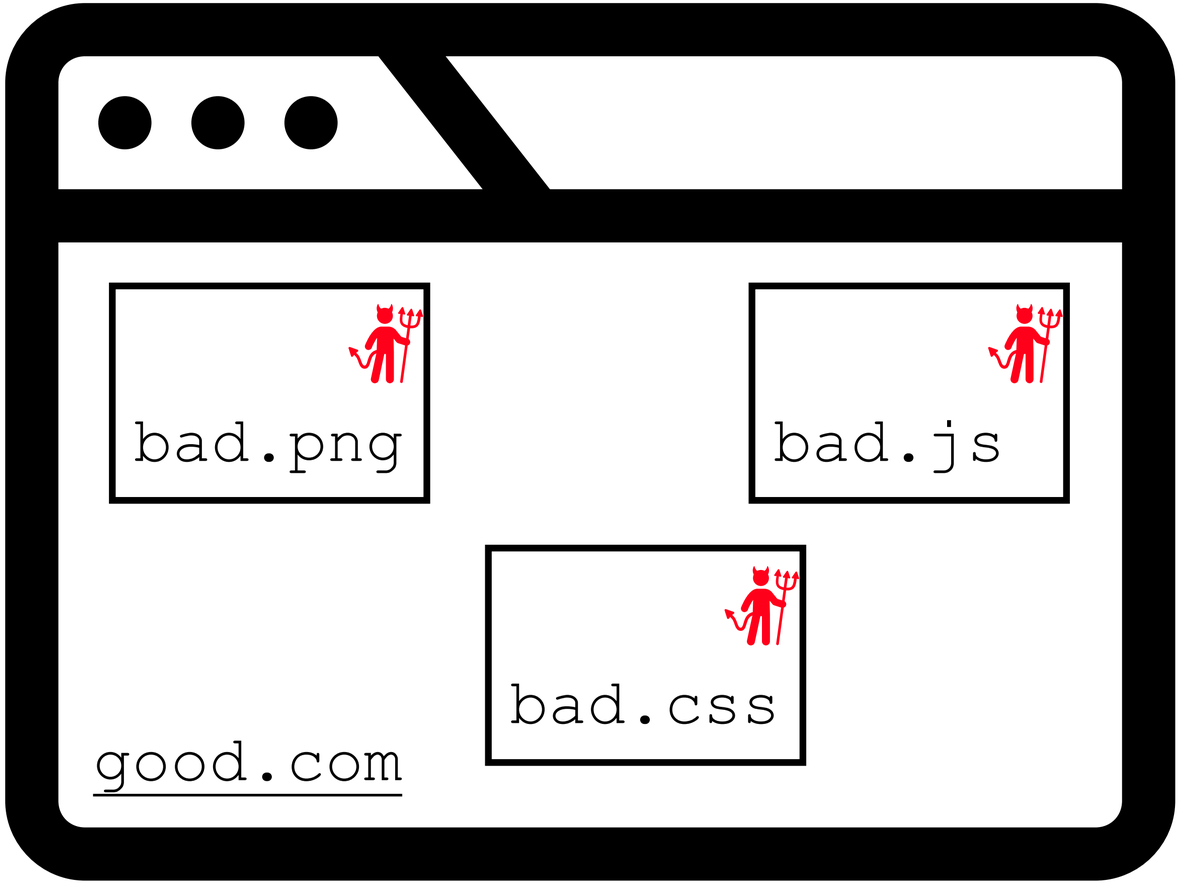
Other models (so you're aware)

- Network attacker



Other models (so you're aware)

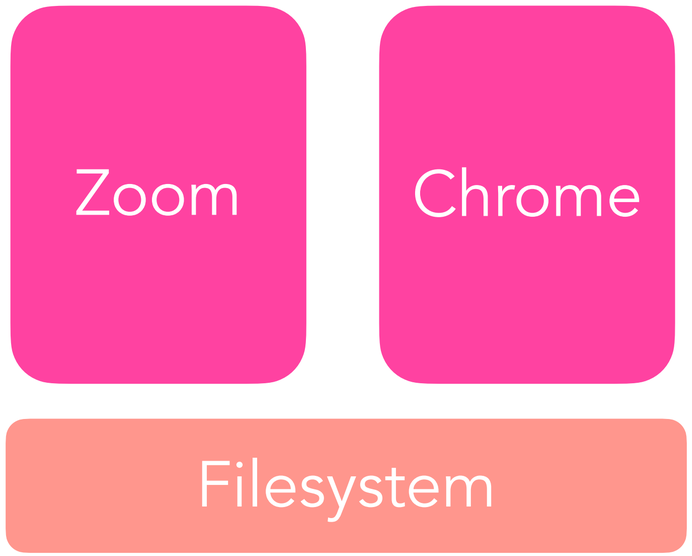
- Gadget attacker (more on this next time)



Lots of variants of web attacker!

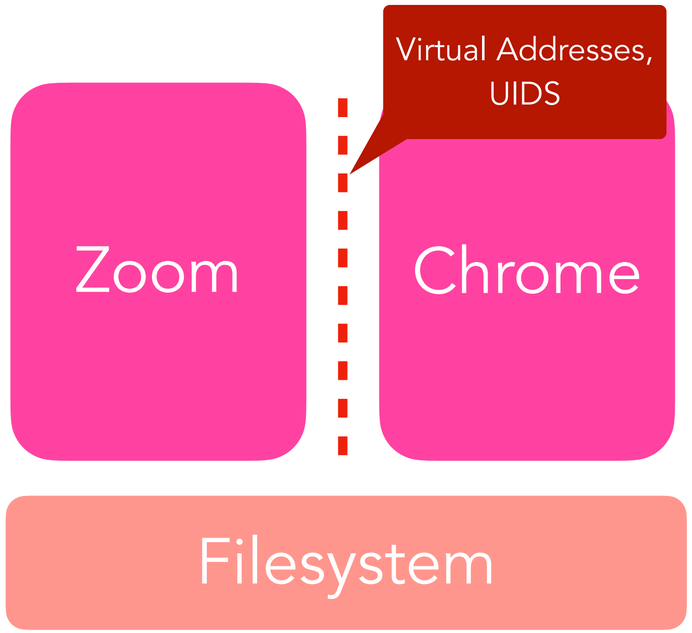
Web security model

- Much like the OS model, we want to safely browse the web in the presence of web attackers
  - Browsers these days are just like operating systems, need to provide **isolation**



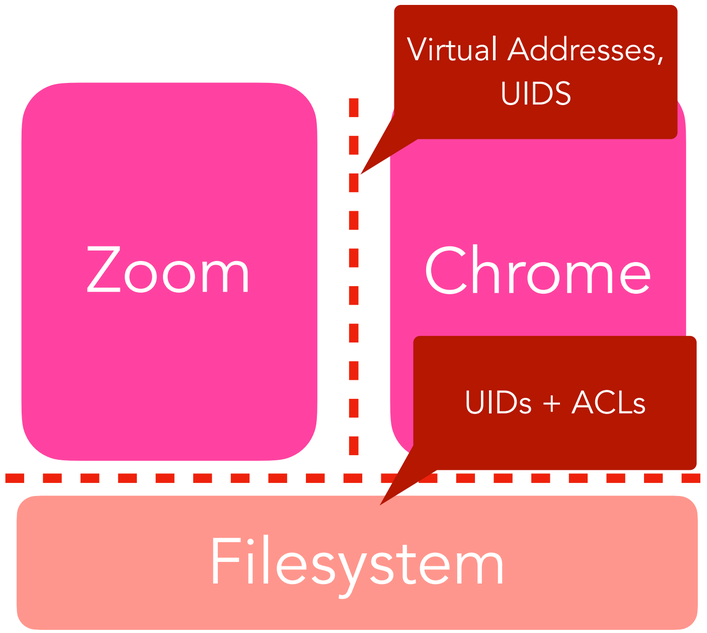
Web security model

- Much like the OS model, we want to safely browse the web in the presence of web attackers
  - Browsers these days are just like operating systems, need to provide **isolation**



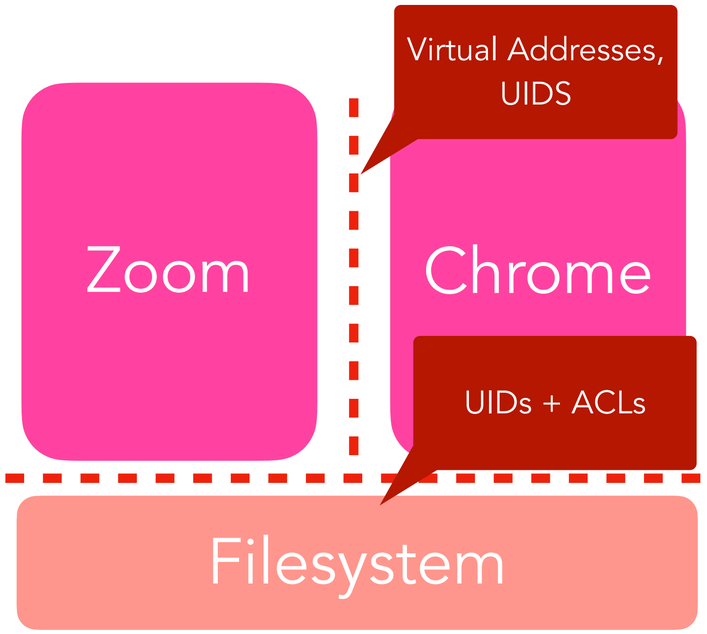
Web security model

- Much like the OS model, we want to safely browse the web in the presence of web attackers
  - Browsers these days are just like operating systems, need to provide **isolation**



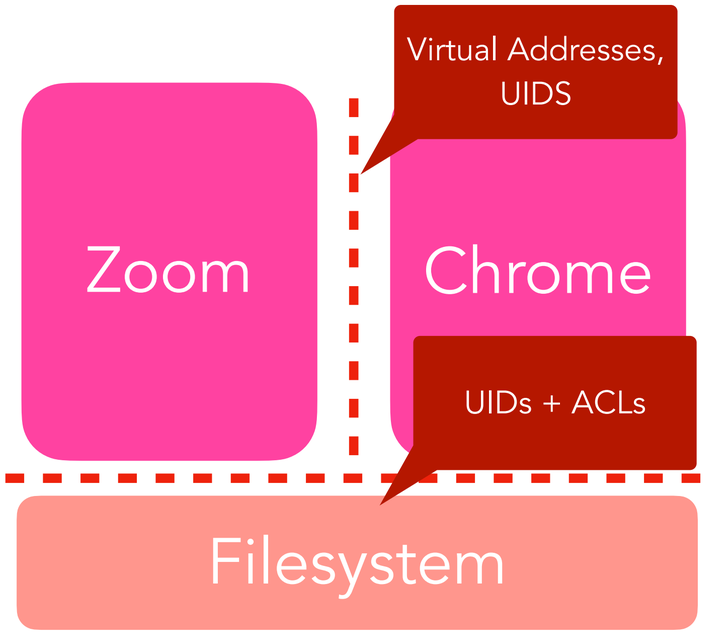
Web security model

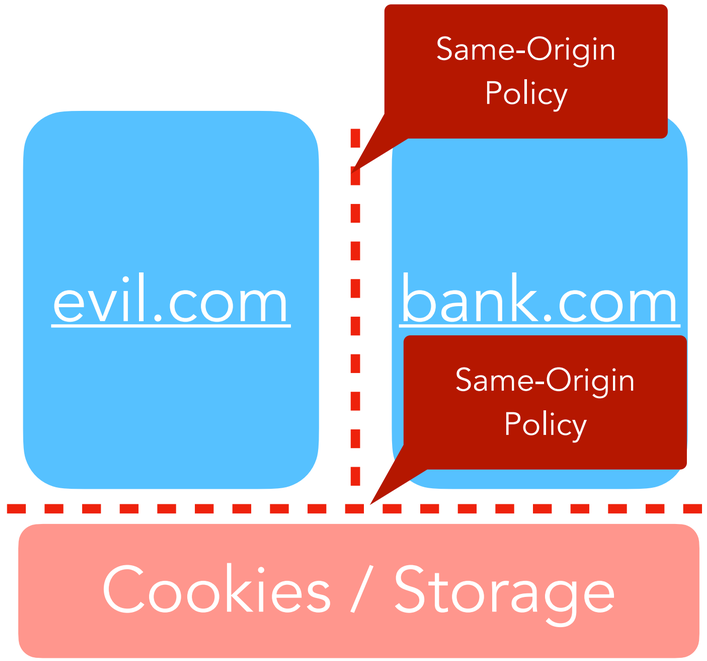
- Much like the OS model, we want to safely browse the web in the presence of web attackers
  - Browsers these days are just like operating systems, need to provide **isolation**



Web security model

- Much like the OS model, we want to safely browse the web in the presence of web attackers
  - Browsers these days are just like operating systems, need to provide **isolation**





Same origin policy (SOP)

- Origin: isolation unit/trust boundary on the web
  - Origin is defined as (**scheme, domain, port**) triple derived from the URL
  - Fate sharing: If you come from the same place, you must be authorized
- SOP goal: isolate content of different origins
  - **Confidentiality:** script contained in **evil.com** should not be allowed to read data in **good.com**'s page
  - **Integrity:** script from **evil.com** should not be able to modify the content of **good.com**'s page

Understanding Origins

- Are these the same origin?
  - https://www.google.com, http://www.google.com

Understanding Origins

- Are these the same origin?
  - https://www.google.com, http://www.google.com
  - https://www.google.com:443, https://www.google.com

Understanding Origins

- Are these the same origin?
  - https://www.google.com, http://www.google.com
  - https://www.google.com:443, https://www.google.com
  - https://www.google.com:443, https://google.com:443

Understanding Origins

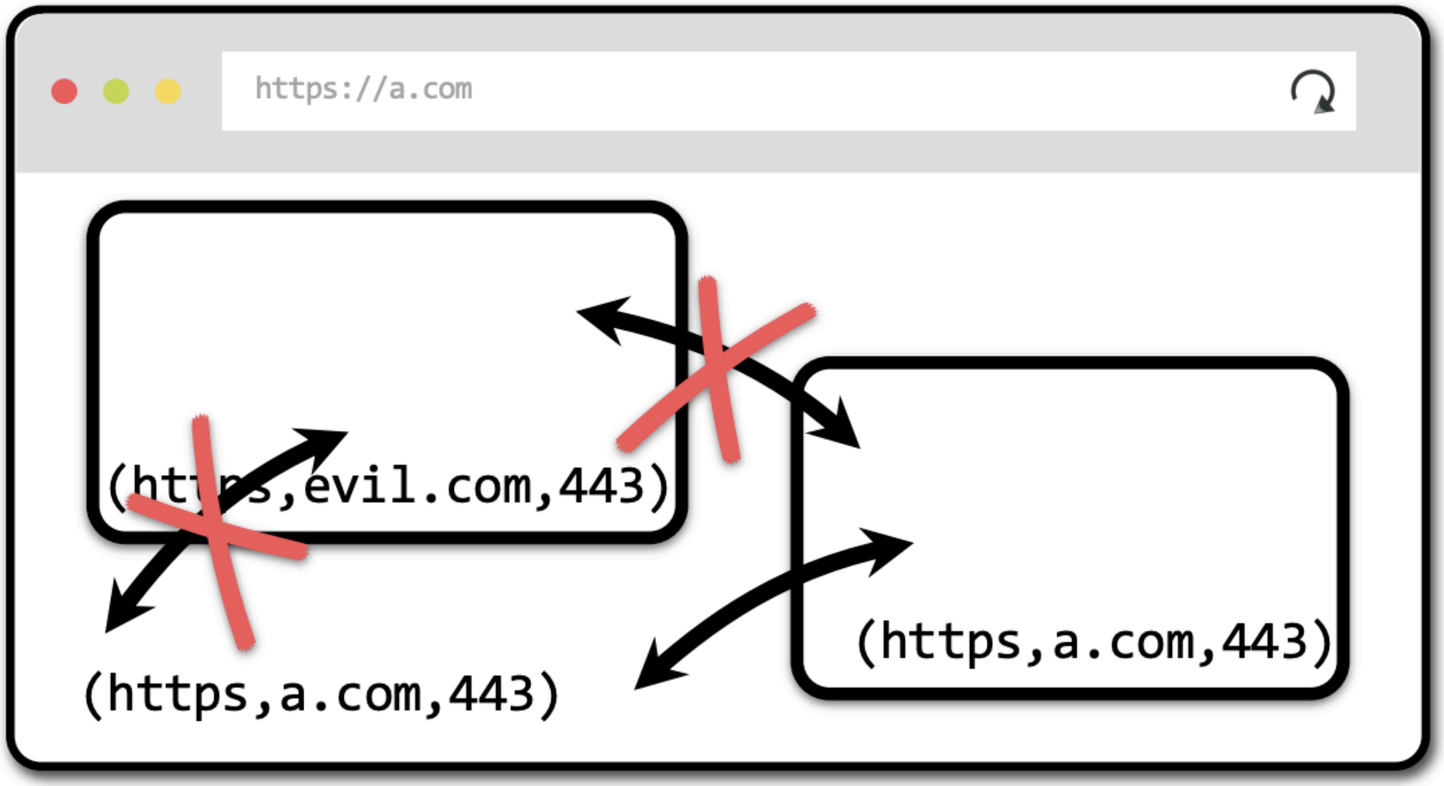
- Are these the same origin?
  - https://www.google.com, http://www.google.com
  - https://www.google.com:443, https://www.google.com
  - https://www.google.com:443, https://google.com:443
    - **These are different origins even though they end up on the same page!**

Understanding Origins

- Are these the same origin?
  - https://www.google.com, http://www.google.com
  - https://www.google.com:443, https://www.google.com
  - https://www.google.com:443, https://google.com:443
    - **These are different origins even though they end up on the same page!**
  - https://www.kumarde.com/cse127, https://www.kumarde.com/cse227

SOP for the DOM

- Each frame in a window has its own *origin*
- Frame can only access data with the same origin

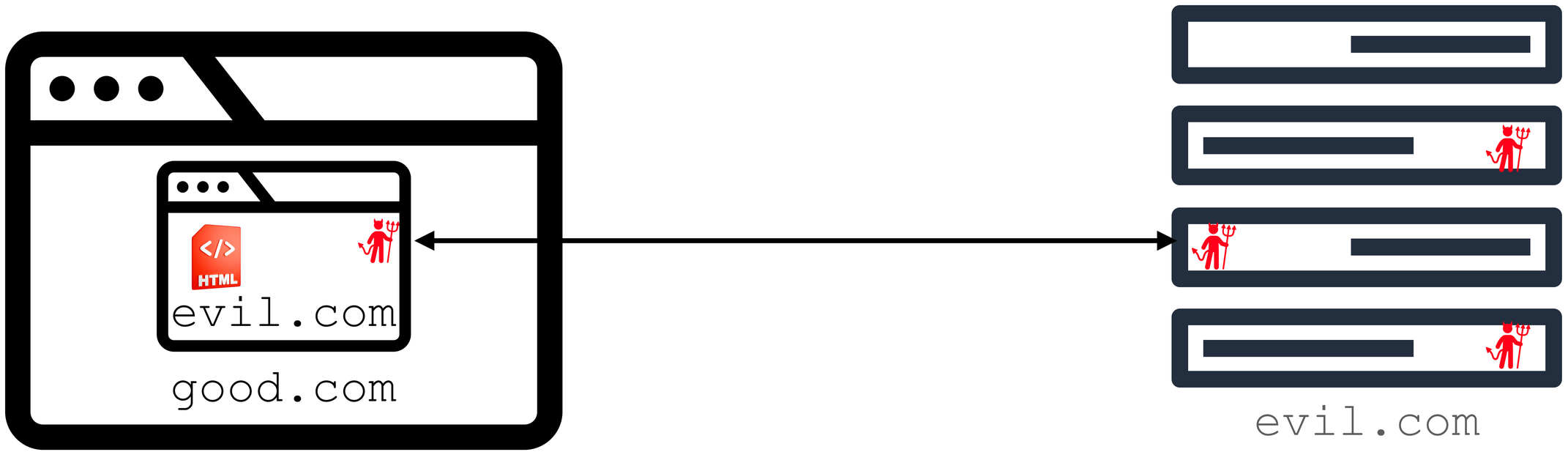


SOP for HTTP

- Pages *can* perform requests across origins
  - SOP does **not** prevent a page from leaking data to another origin by encoding it in a URL, request body, etc.
  - Advertisers with “backroom deals” will often do this on the backend
- SOP *does* prevent code from *directly inspecting* and *modifying* HTTP responses

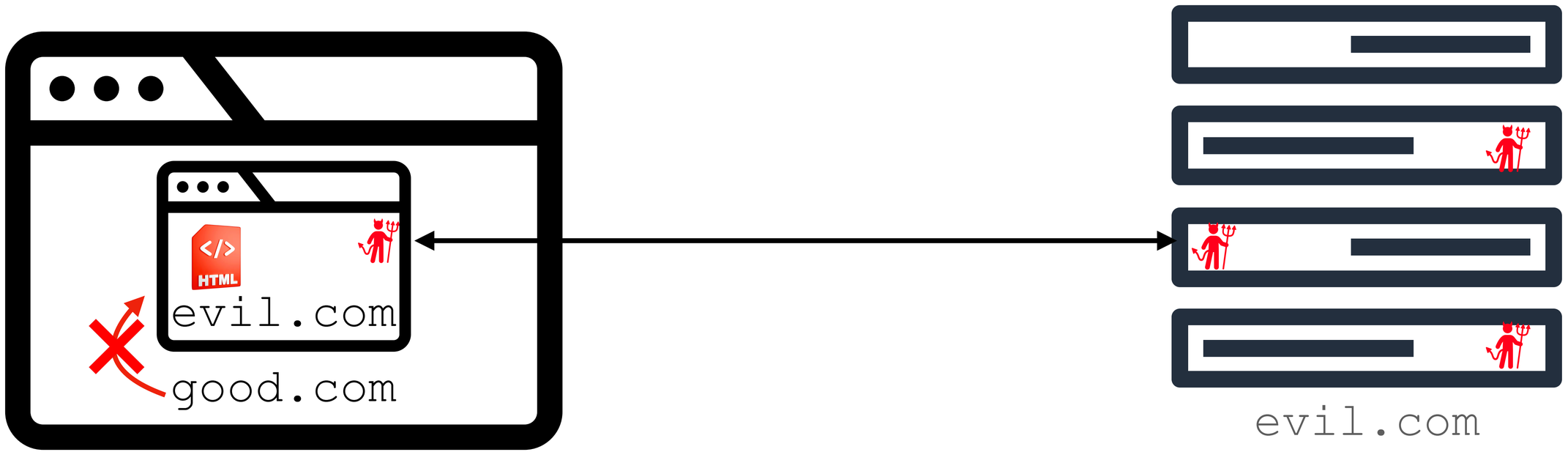
SOP for Documents

- Can load cross-origin HTML in *frames*, but cannot inspect or modify the frame content



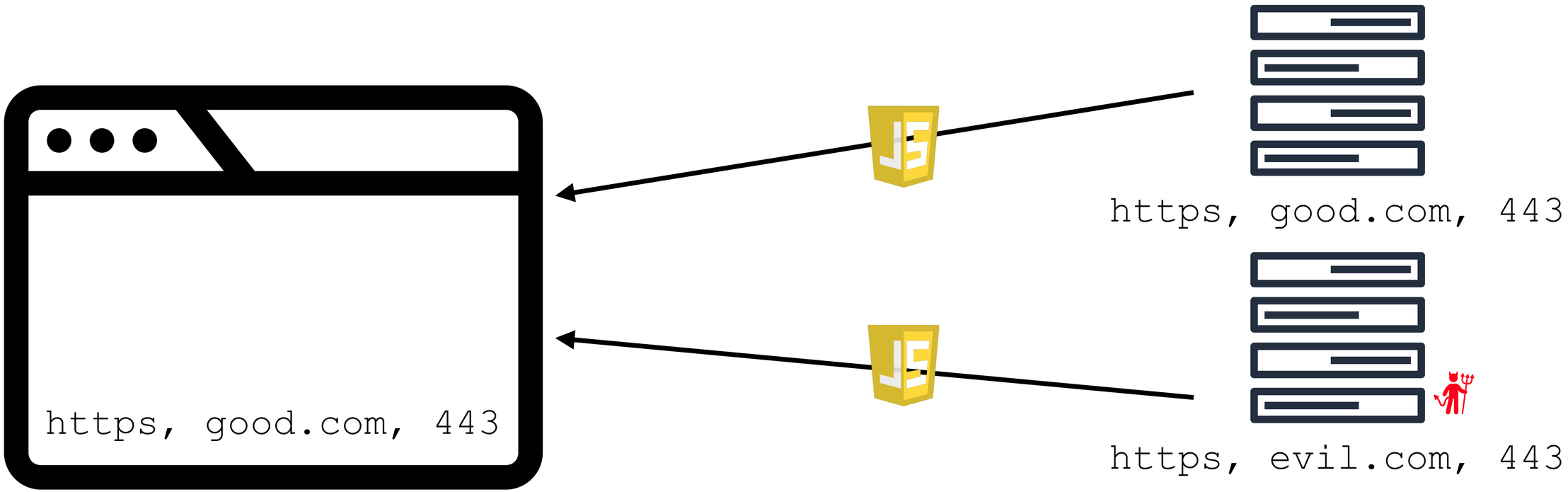
SOP for Documents

- Can load cross-origin HTML in *frames*, but cannot inspect or modify the frame content



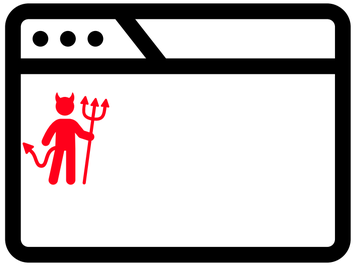
SOP for Scripts

- **Can** load scripts across origins! (e.g., jQuery!)
- Scripts execute with the privileges of the *page*

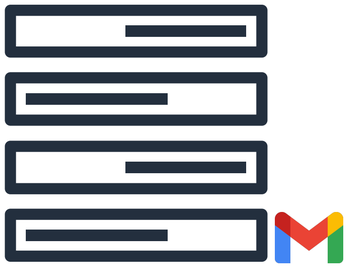


SOP: Cross-Origin Data with JS

evil.com



https, evil.com, 443

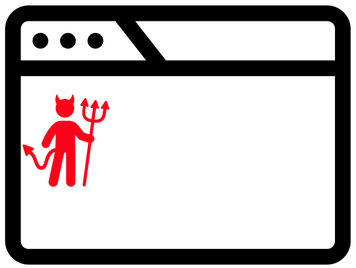


gmail.com

SOP: Cross-Origin Data with JS

GET /
Host: evil.com

evil.com



https, evil.com, 443

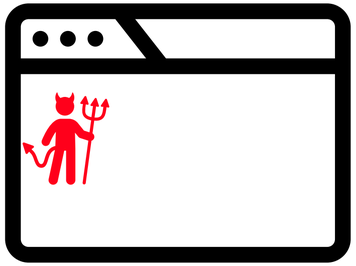
gmail.com

SOP: Cross-Origin Data with JS

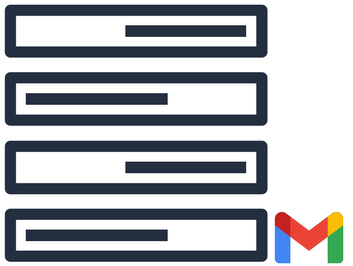
GET /
Host: evil.com

evil.com

```
<script>
$.get('http://gmail.com/msgs.json', function(data) {
    alert(data);
}
</script>
```

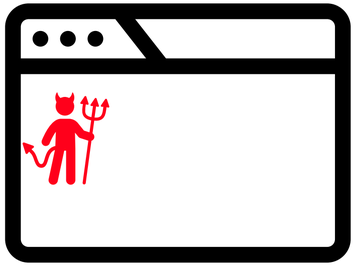


https, evil.com, 443



gmail.com

SOP: Cross-Origin Data with JS



https, evil.com, 443

GET /
Host: evil.com

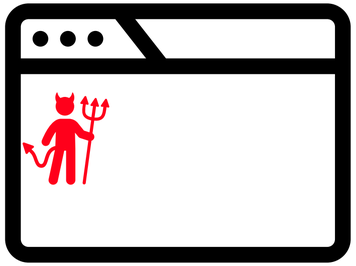
evil.com

```
<script>
$.get('http://gmail.com/msgs.json', function(data) {
    alert(data);
}
</script>
```

GET msgs.json
Host: gmail.com

gmail.com

SOP: Cross-Origin Data with JS



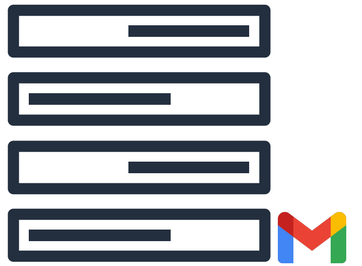
https, evil.com, 443

GET /
Host: evil.com

evil.com

```
<script>
$.get('http://gmail.com/msgs.json', function(data) {
    alert(data);
}
</script>
```

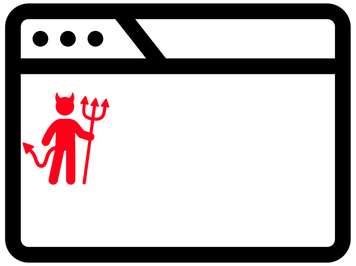
GET msgs.json
Host: gmail.com



gmail.com

Will gmail.com send back msgs.json?

SOP: Cross-Origin Data with JS



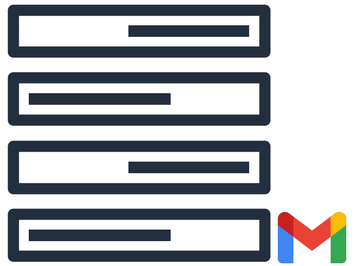
https, evil.com, 443

GET /
Host: evil.com

evil.com

```
<script>
$.get('http://gmail.com/msgs.json', function(data) {
    alert(data);
}
</script>
```

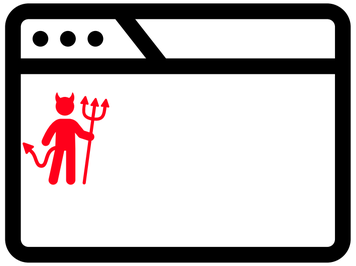
GET msgs.json
Host: gmail.com



gmail.com

HTTP/1.1 200 OK
{key: value}

SOP: Cross-Origin Data with JS



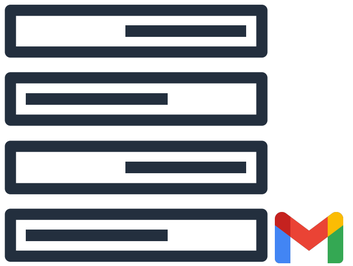
`https, evil.com, 443`

GET /
Host: evil.com

`evil.com`

```
<script>
$.get('http://gmail.com/msgs.json', function(data) {
    alert(data);
}
</script>
```

GET msgs.json
Host: gmail.com

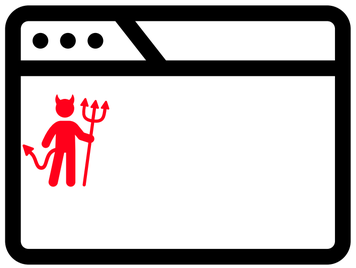


`gmail.com`

HTTP/1.1 200 OK
{key: value}

Can we read msgs.json?

SOP: Cross-Origin Data with JS



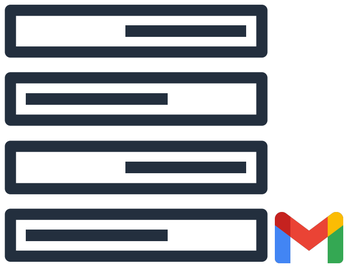
https, evil.com, 443

GET /
Host: evil.com

evil.com

```
<script>
$.get('http://gmail.com/msgs.json', function(data) {
    alert(data);
}
</script>
```

GET msgs.json
Host: gmail.com



gmail.com

HTTP/1.1 200 OK
{key: value}

Blocked by SOP

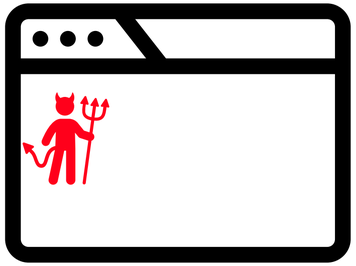
evil.com cannot read contents

SOP: Cross-Origin Data with Embedded JS

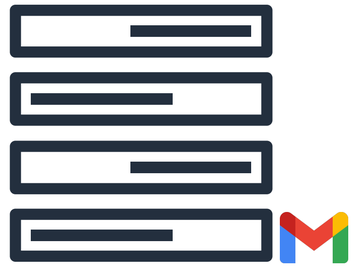
GET /
Host: evil.com

<script src = “http://gmail.com/chat.js"></script>

evil.com

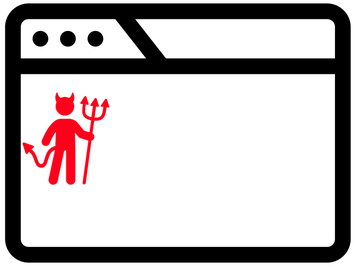


https, evil.com, 443



gmail.com

SOP: Cross-Origin Data with Embedded JS



https, evil.com, 443

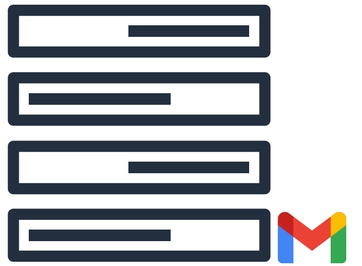
GET /
Host: evil.com

evil.com

<script src = “http://gmail.com/chat.js"></script>

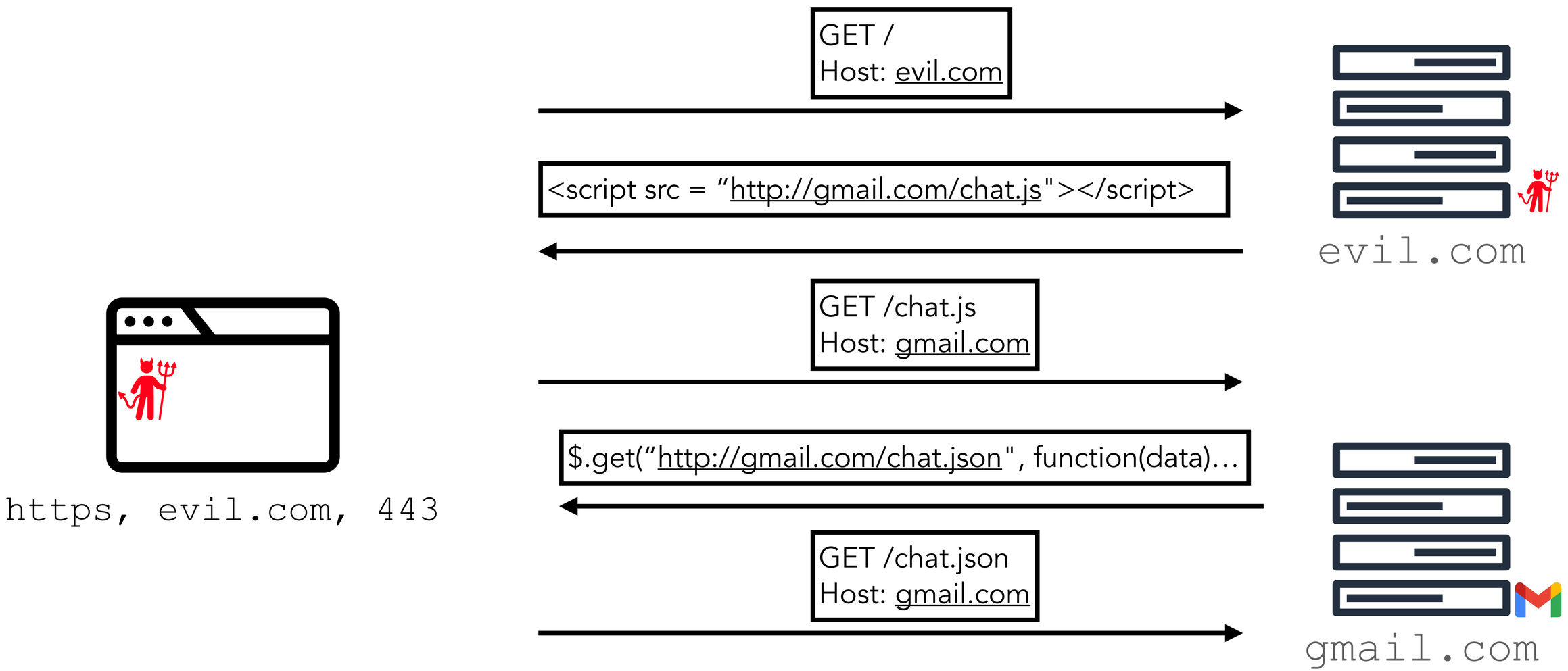
GET /chat.js
Host: gmail.com

$.get(“http://gmail.com/chat.json", function(data)…



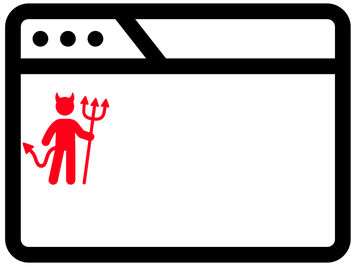
gmail.com

SOP: Cross-Origin Data with Embedded JS



Will gmail.com send back chat.json?

SOP: Cross-Origin Data with Embedded JS



`https, evil.com, 443`

GET /
Host: evil.com

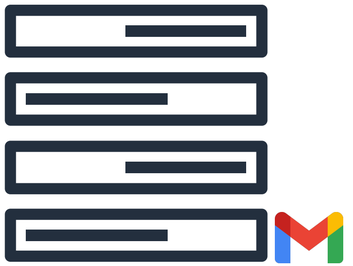
`evil.com`

<script src = “http://gmail.com/chat.js"></script>

GET /chat.js
Host: gmail.com

$.get(“http://gmail.com/chat.json", function(data)…

GET /chat.json
Host: gmail.com

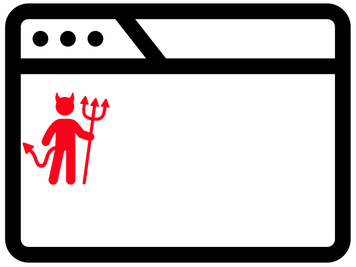


`gmail.com`

HTTP/1.1 200 OK
{key: value}

Can we read chat.json?

SOP: Cross-Origin Data with Embedded JS



https, evil.com, 443

GET /
Host: evil.com

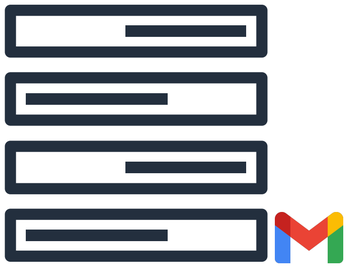
evil.com

<script src = "http://gmail.com/chat.js"></script>

GET /chat.js
Host: gmail.com

$.get("http://gmail.com/chat.json", function(data)…

GET /chat.json
Host: gmail.com



gmail.com

HTTP/1.1 200 OK
{key: value}

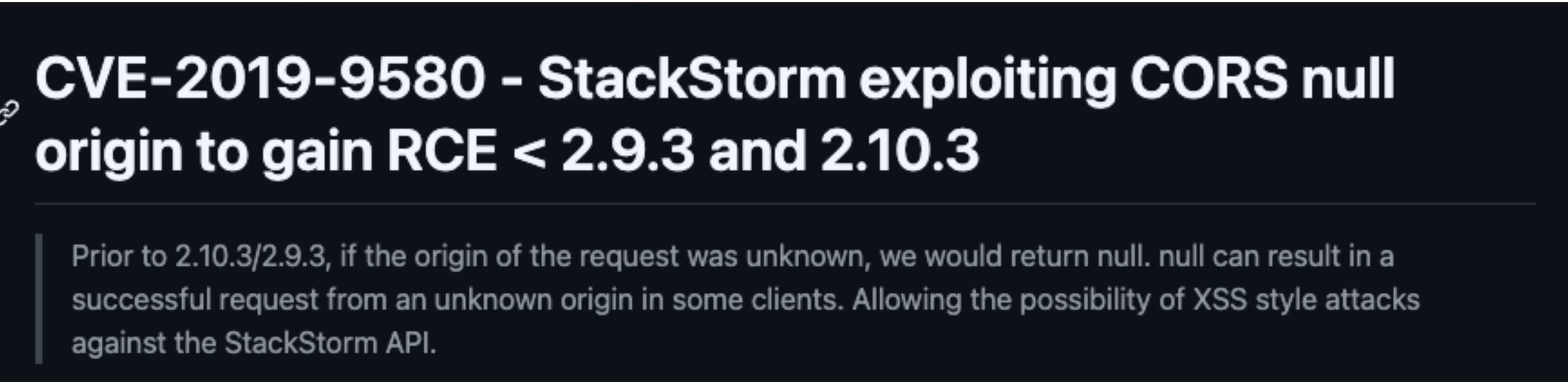
Blocked by SOP
Even though script came from gmail.com, ran with evil.com's privileges

Aside: Cross-Origin Resource Sharing

- Cross-Origin Resource Sharing (CORS) is a mechanism in browsers that allow servers to “opt-out” of SOP for specific resources
  - Set in HTTP headers
    - `Access-Control-Allow-Origin`
    - `Access-Control-Allow-Headers`
    - `Access-Control-Allow-Credentials`
    - `Access-Control-Expose-Headers`
- Notoriously tricky to check appropriately, a deep well of pain…

Aside: Cross-Origin Resource Sharing

- Cross-Origin Resource Sharing (CORS) is a mechanism in browsers that allow

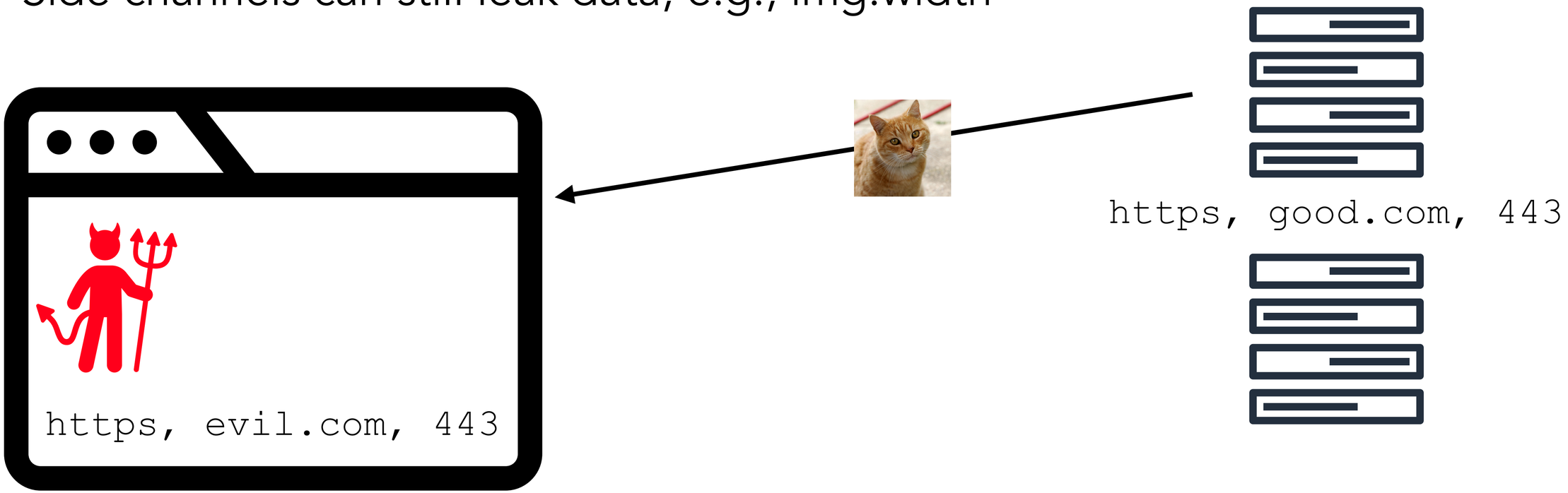


  - `Access-Control-Expose-Headers`

- Notoriously tricky to check appropriately, a deep well of pain…

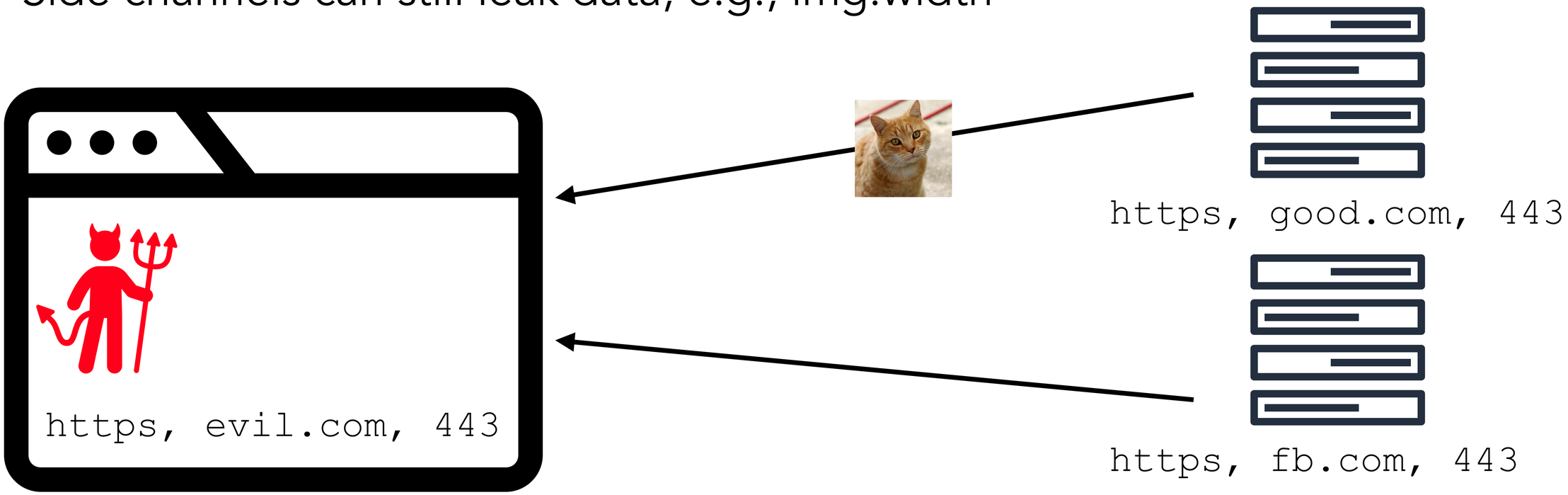
SOP for Images

- Browser renders cross-origin images, but SOP prevents page from inspecting individual pixels
- Side channels can still leak data, e.g., img.width



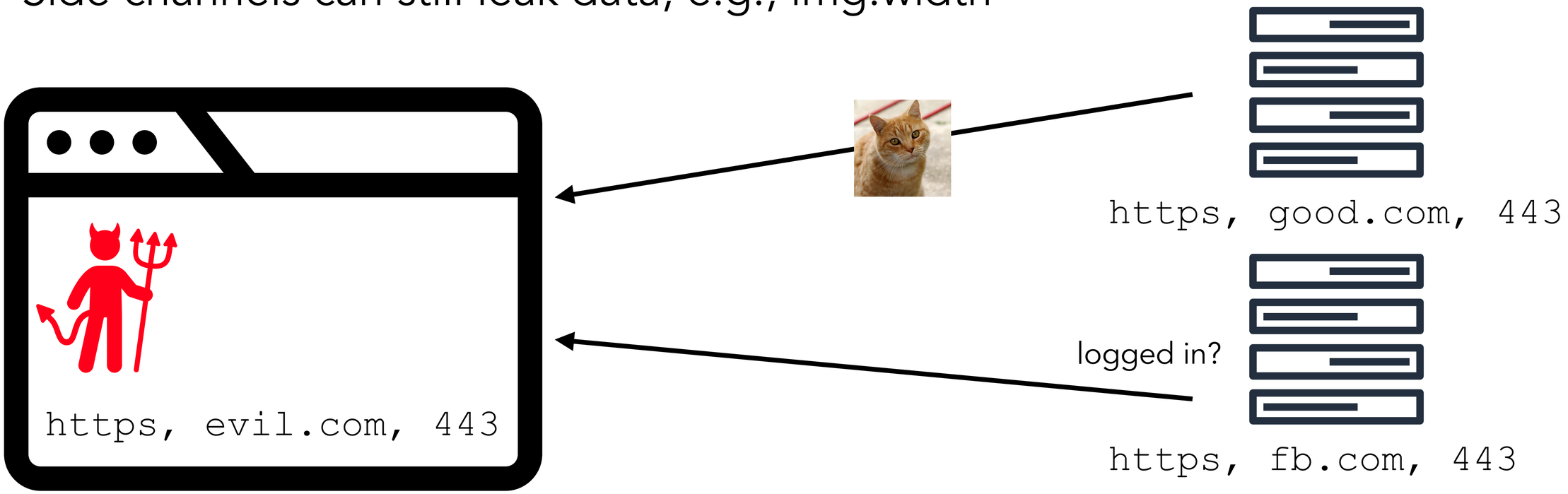
SOP for Images

- Browser renders cross-origin images, but SOP prevents page from inspecting individual pixels
- Side channels can still leak data, e.g., img.width



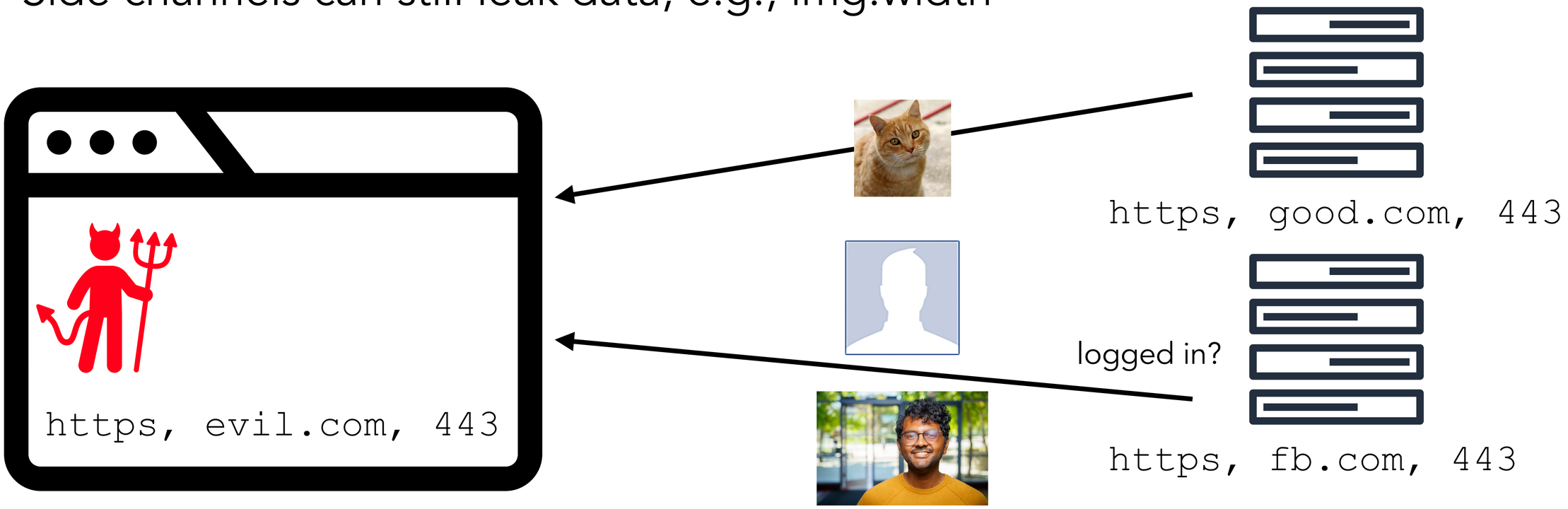
SOP for Images

- Browser renders cross-origin images, but SOP prevents page from inspecting individual pixels
- Side channels can still leak data, e.g., img.width



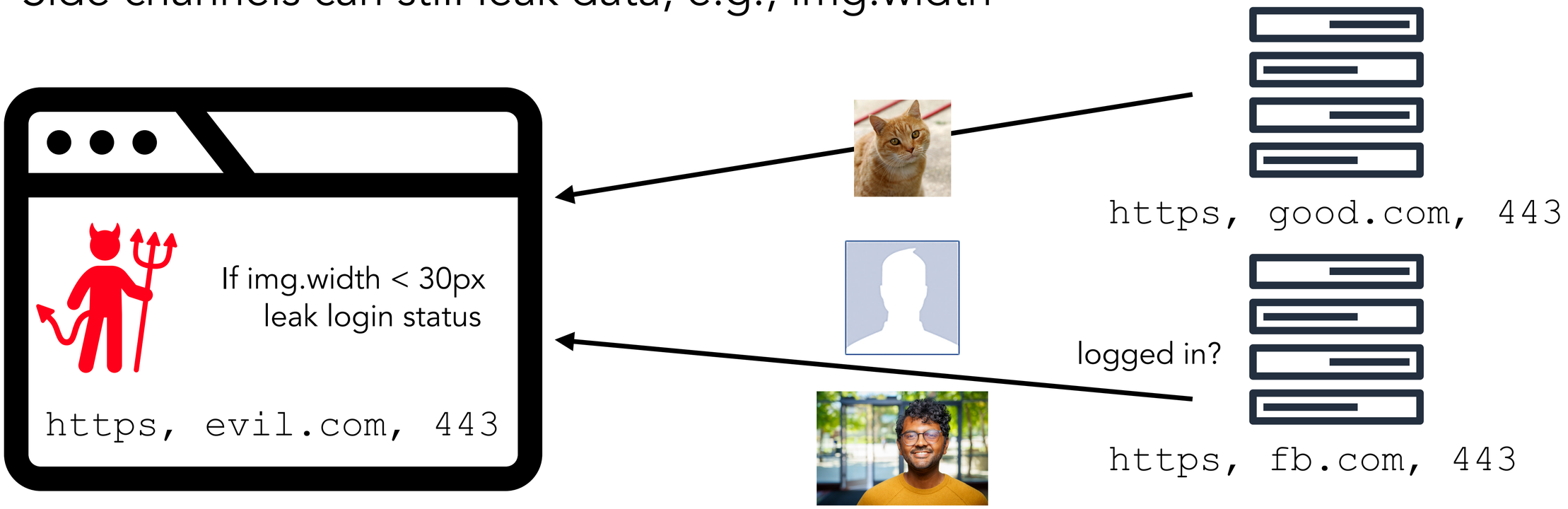
SOP for Images

- Browser renders cross-origin images, but SOP prevents page from inspecting individual pixels
- Side channels can still leak data, e.g., img.width



SOP for Images

- Browser renders cross-origin images, but SOP prevents page from inspecting individual pixels
- Side channels can still leak data, e.g., img.width

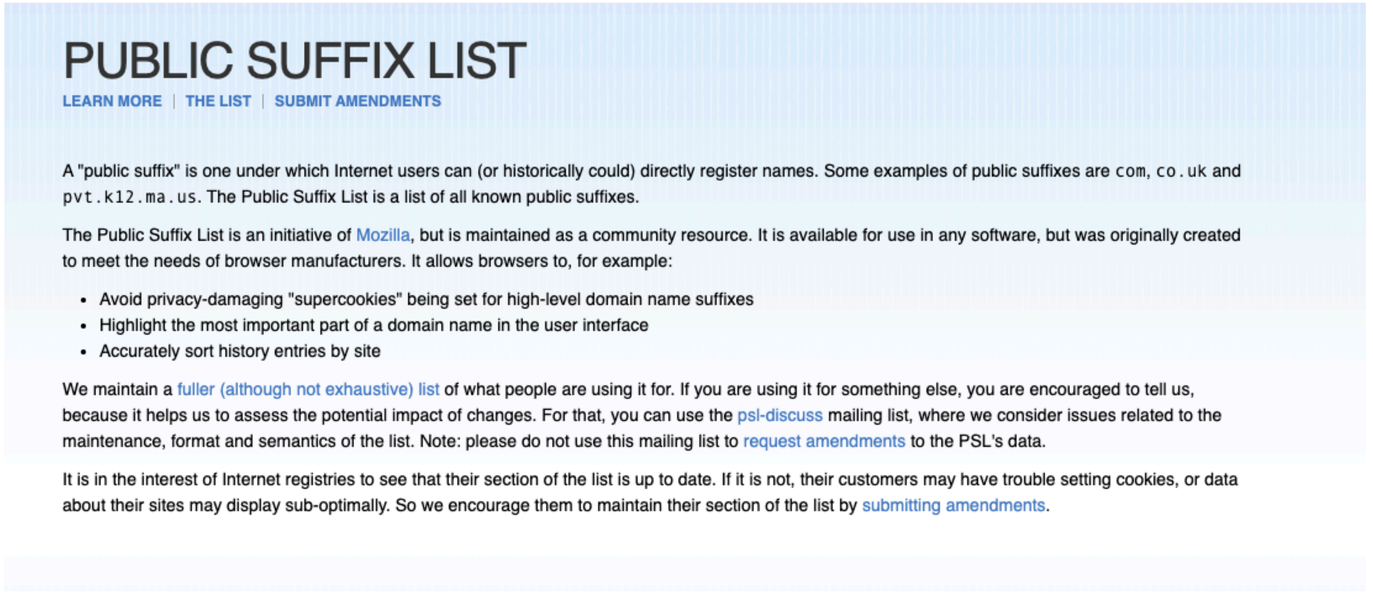


SOP for Cookies

- Cookies are special, so they get their own slightly different definition of origin
  - Cookie SOP: (scheme, domain, *path*)
  - (https, cseweb.used.edu, /classes/wi26/cse127-a)
- Server can declare **domain** property for any cookie
  - Set-Cookie: <cookie-name>=<cookie-value>; Domain=<domain-value>

SOP: Cookie Scope Setting

- What domains can a web server set cookies for?
- domain: any domain-suffix of URL-hostname, except “public suffixes”
  - host = “login.site.com," can set login.site.com, site.com, but **not** .com
- path: can always be set to anything



PUBLIC SUFFIX LIST

LEARN MORE | THE LIST | SUBMIT AMENDMENTS

A "public suffix" is one under which Internet users can (or historically could) directly register names. Some examples of public suffixes are `com`, `co.uk` and `pvt.k12.ma.us`. The Public Suffix List is a list of all known public suffixes.

The Public Suffix List is an initiative of Mozilla, but is maintained as a community resource. It is available for use in any software, but was originally created to meet the needs of browser manufacturers. It allows browsers to, for example:

- Avoid privacy-damaging "supercookies" being set for high-level domain name suffixes
- Highlight the most important part of a domain name in the user interface
- Accurately sort history entries by site

We maintain a fuller (although not exhaustive) list of what people are using it for. If you are using it for something else, you are encouraged to tell us, because it helps us to assess the potential impact of changes. For that, you can use the psl-discuss mailing list, where we consider issues related to the maintenance, format and semantics of the list. Note: please do not use this mailing list to request amendments to the PSL's data.

It is in the interest of Internet registries to see that their section of the list is up to date. If it is not, their customers may have trouble setting cookies, or data about their sites may display sub-optimally. So we encourage them to maintain their section of the list by submitting amendments.

How do we decide to send cookies?

- Browser sends all cookies in a URL's scope:
  - Cookie's domain is a domain suffix of URL's domain
  - Cookie's path is a **prefix** of the URL path

How do we decide to send cookies?

Cookie 1:
name = mycookie
value = value
domain = login.site.com
path = /

Cookie 2:
name = mycookie2
value = value
domain = site.com
path = /

Cookie 3:
name = mycookie3
value = value
domain = site.com
path = /my/home

How do we decide to send cookies?

Cookie 1:
name = mycookie
value = value
domain = login.site.com
path = /

Cookie 2:
name = mycookie2
value = value
domain = site.com
path = /

Cookie 3:
name = mycookie3
value = value
domain = site.com
path = /my/home

Request to URL:	Cookie 1	Cookie 2	Cookie 3
checkout.site.com			
login.site.com			
login.site.com/my/home			
site.com/my			

How do we decide to send cookies?

Cookie 1:
name = mycookie
value = value
domain = login.site.com
path = /

Cookie 2:
name = mycookie2
value = value
domain = site.com
path = /

Cookie 3:
name = mycookie3
value = value
domain = site.com
path = /my/home

Request to URL:	Cookie 1	Cookie 2	Cookie 3
checkout.site.com	No	Yes	No
login.site.com			
login.site.com/my/home			
site.com/my			

How do we decide to send cookies?

Cookie 1:
name = mycookie
value = value
domain = login.site.com
path = /

Cookie 2:
name = mycookie2
value = value
domain = site.com
path = /

Cookie 3:
name = mycookie3
value = value
domain = site.com
path = /my/home

Request to URL:	Cookie 1	Cookie 2	Cookie 3
checkout.site.com	No	Yes	No
login.site.com	Yes	Yes	No
login.site.com/my/home			
site.com/my			

How do we decide to send cookies?

Cookie 1:
name = mycookie
value = value
domain = login.site.com
path = /

Cookie 2:
name = mycookie2
value = value
domain = site.com
path = /

Cookie 3:
name = mycookie3
value = value
domain = site.com
path = /my/home

Request to URL:	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/my			

How do we decide to send cookies?

Cookie 1:
name = mycookie
value = value
domain = login.site.com
path = /

Cookie 2:
name = mycookie2
value = value
domain = site.com
path = /

Cookie 3:
name = mycookie3
value = value
domain = site.com
path = /my/home

Request to URL:	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/my	No	Yes	No

Web Attacks

Three classical web attacks

- Cross-Site Request Forgery (CSRF)
- SQL Injection
- Cross-Site Scripting (XSS)
- **You will implement all three in PA3!**

Cross-Site Request Forgery

- Recall: Browsers send cookies with requests all the time. How?
  - If a user clicked a link (on a website, or even in email)
  - If another page embedded the target page in an iframe
  - If a client-side script issued the request

Cross-Site Request Forgery

- Recall: Browsers send cookies with requests all the time. How?
  - If a user clicked a link (on a website, or even in email)
  - If another page embedded the target page in an iframe
  - If a client-side script issued the request
- Doesn't matter where the request comes from, only thing that matters is the ***target*** of the request
  - Where might there be a problem?

Cross-Site Request Forgery

- Recall: Browsers send cookies with requests all the time. How?
  - If a user clicked a link (on a website, or even in email)
  - If another page embedded the target page in an iframe
  - If a client-side script issued the request
- Doesn't matter where the request comes from, only thing that matters is the ***target*** of the request
  - Where might there be a problem?
- Target doesn't know if the request was **intended** or **authorized** by the user

Typical Authentication Pattern

chase.com

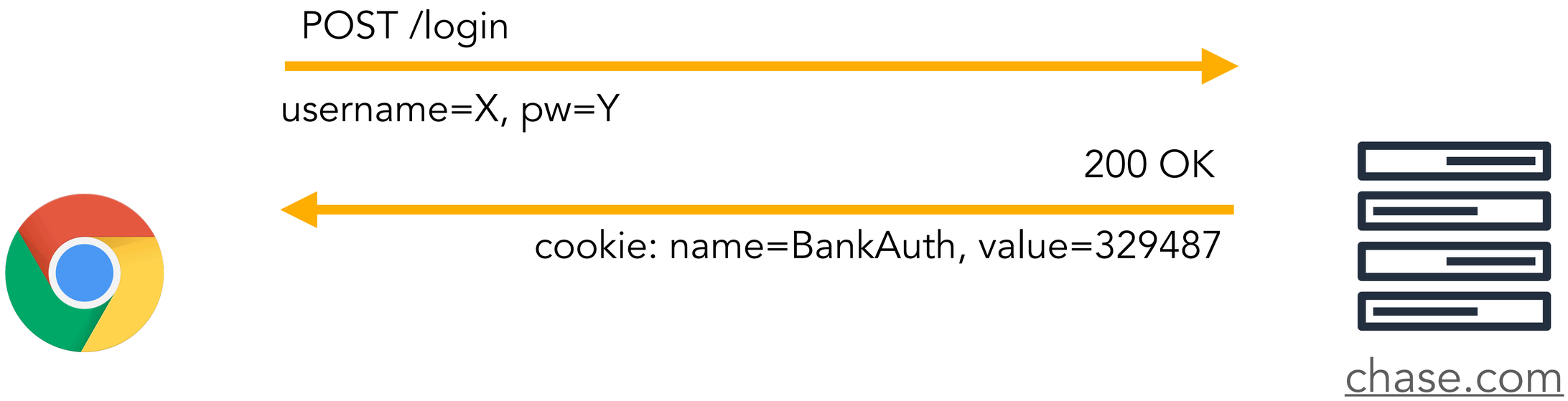
Typical Authentication Pattern

POST /login

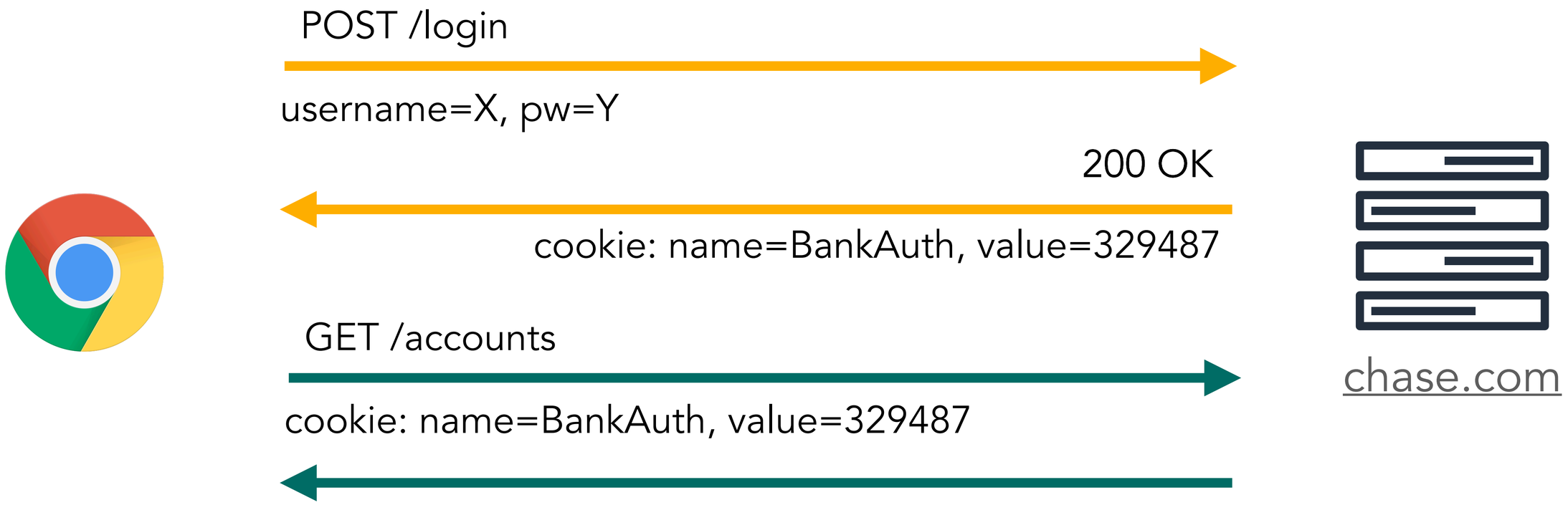
username=X, pw=Y

chase.com

Typical Authentication Pattern



Typical Authentication Pattern



Typical Authentication Pattern

POST /login

username=X, pw=Y

200 OK

cookie: name=BankAuth, value=329487

chase.com

GET /accounts

cookie: name=BankAuth, value=329487

200 OK

POST /transfer

cookie: name=BankAuth, value=329487

200 OK

CSRF Scenario

- User is signed into chase.com
  - Cookie **remains** in the browser's state
- User then visits a malicious website, containing the following:

```
<form name=BillPayForm action="https://chase.com/transfer">
<input name=recipient value=badguy>
<input amount=10000000>
<script> document.BillPayForm.submit(); </script>
```

What does the code above do?

CSRF Scenario

- User is signed into chase.com
  - Cookie **remains** in the browser's state
- User then visits a malicious website, containing the following:

```
<form name=BillPayForm action="https://chase.com/transfer">
<input name=recipient value=badguy>
<input amount=10000000>
<script> document.BillPayForm.submit(); </script>
```

- Code executes an HTTP Post to chase.com
  - Good news, attacker.com can't see the result of POST request thanks to SOP💪

CSRF Scenario

- User is signed into chase.com
  - Cookie **remains** in the browser's state
- User then visits a malicious website, containing the following:

```
<form name=BillPayForm action="https://chase.com/transfer">
<input name=recipient value=badguy>
<input amount=10000000>
<script> document.BillPayForm.submit(); </script>
```

- Code executes an HTTP Post to chase.com
  - Good news, attacker.com can't see the result of POST request thanks to SOP💪
  - Bad news, all your money is gone! 😭

CSRF Patterns

Currently logged into chase.com

evil.com

chase.com

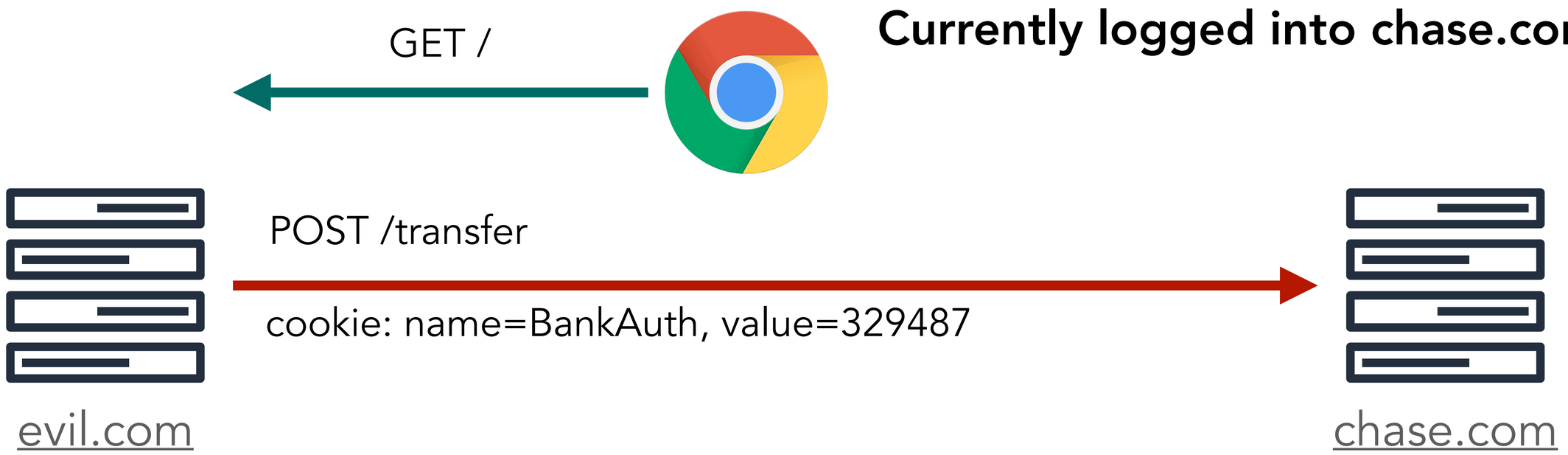
CSRF Patterns

Currently logged into chase.com

evil.com

chase.com

CSRF Patterns



Login CSRF (Special Case)

Currently not logged into google.com

evil.com

google.com

Login CSRF (Special Case)

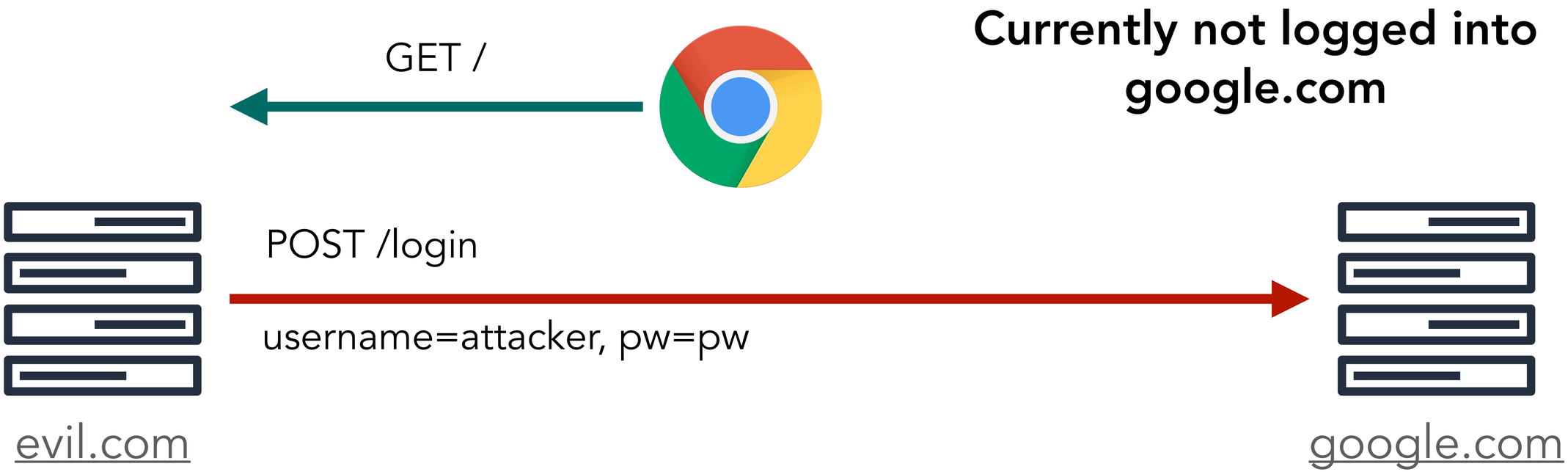
GET /

evil.com

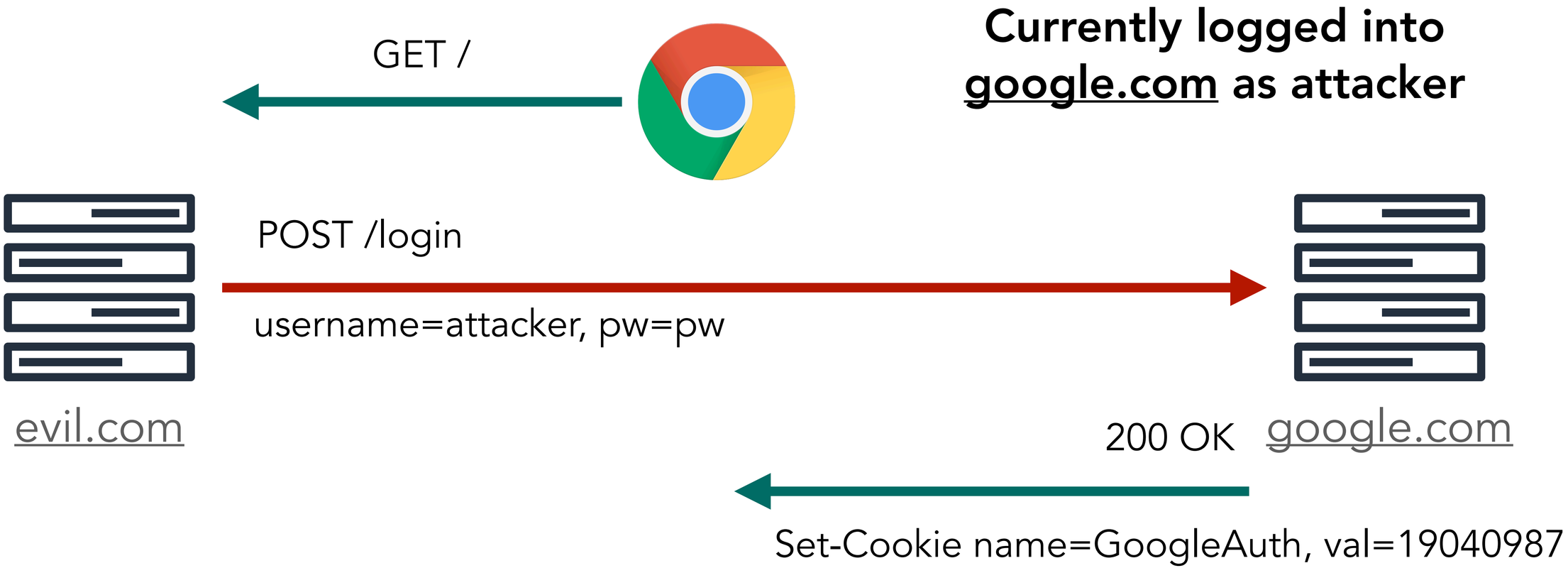
Currently not logged into google.com

google.com

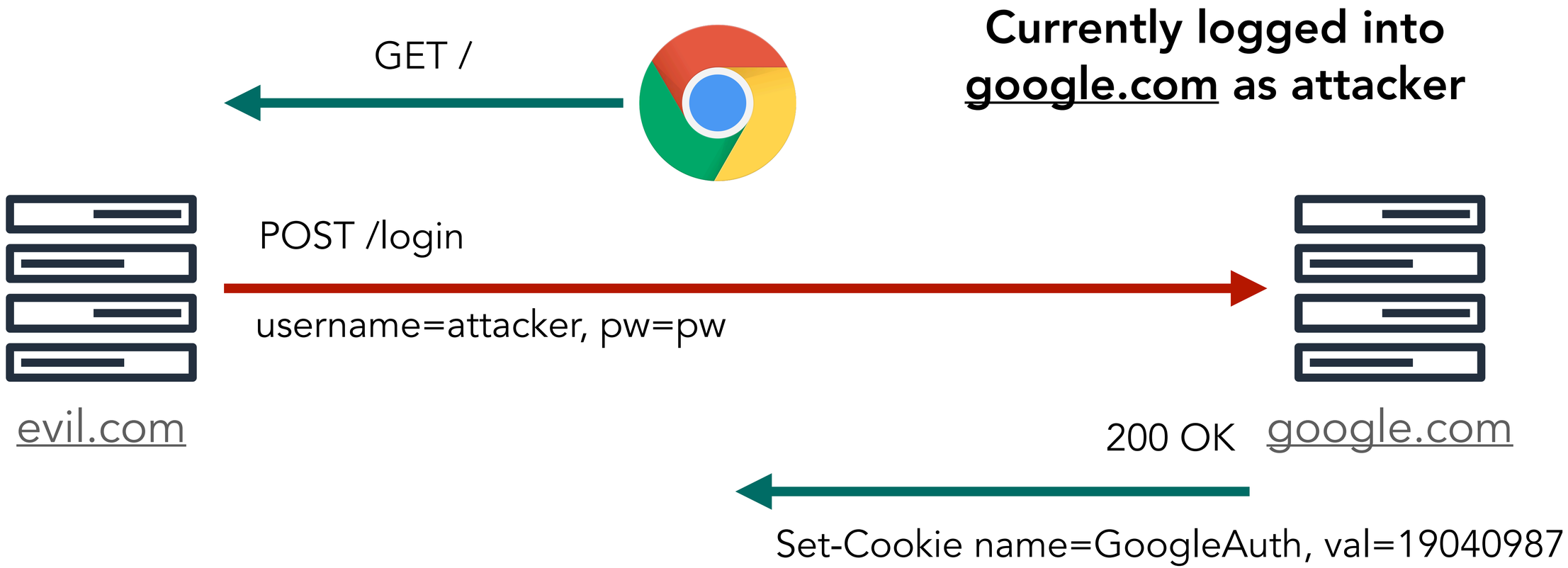
Login CSRF (Special Case)



Login CSRF (Special Case)

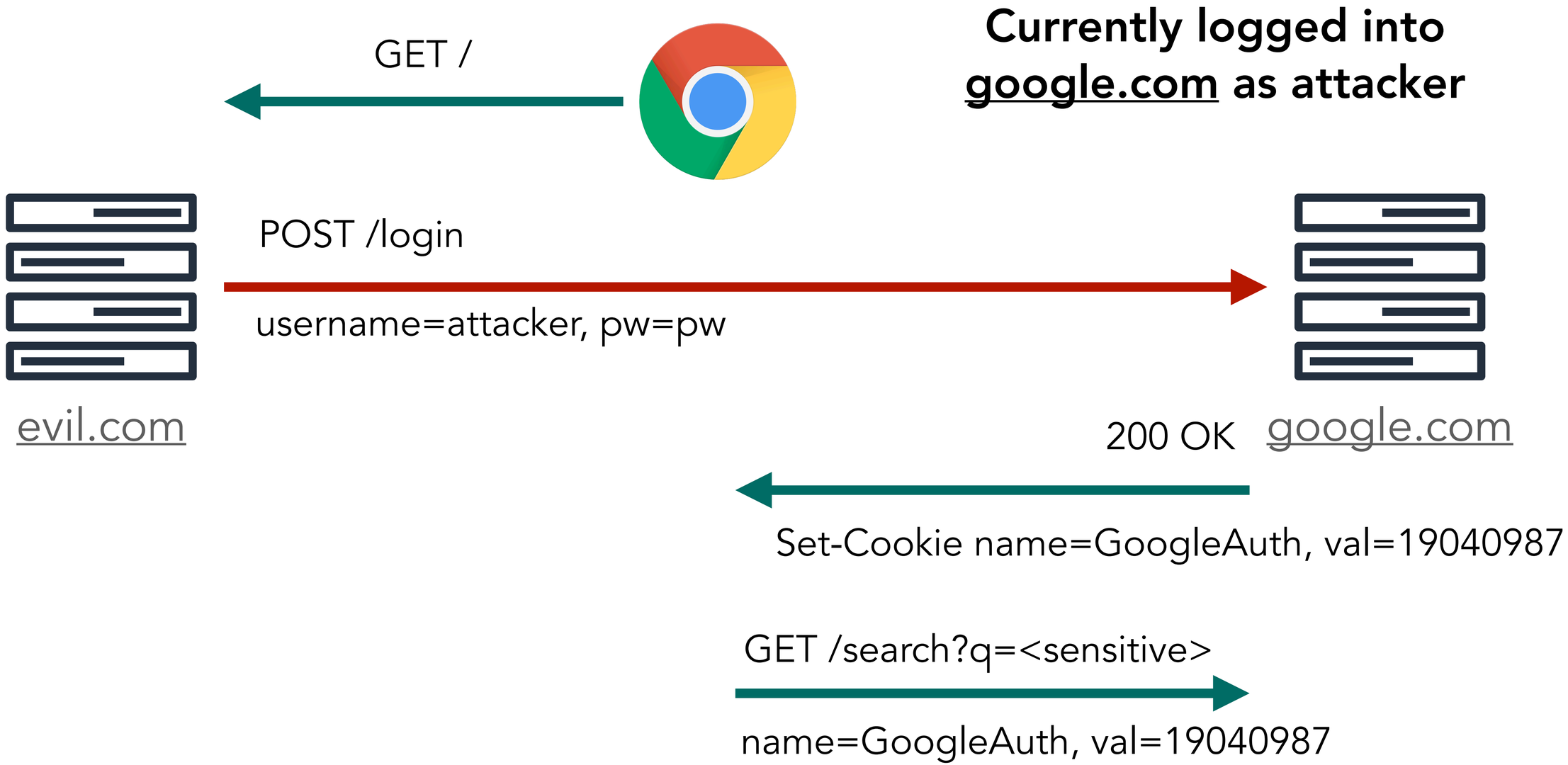


Login CSRF (Special Case)

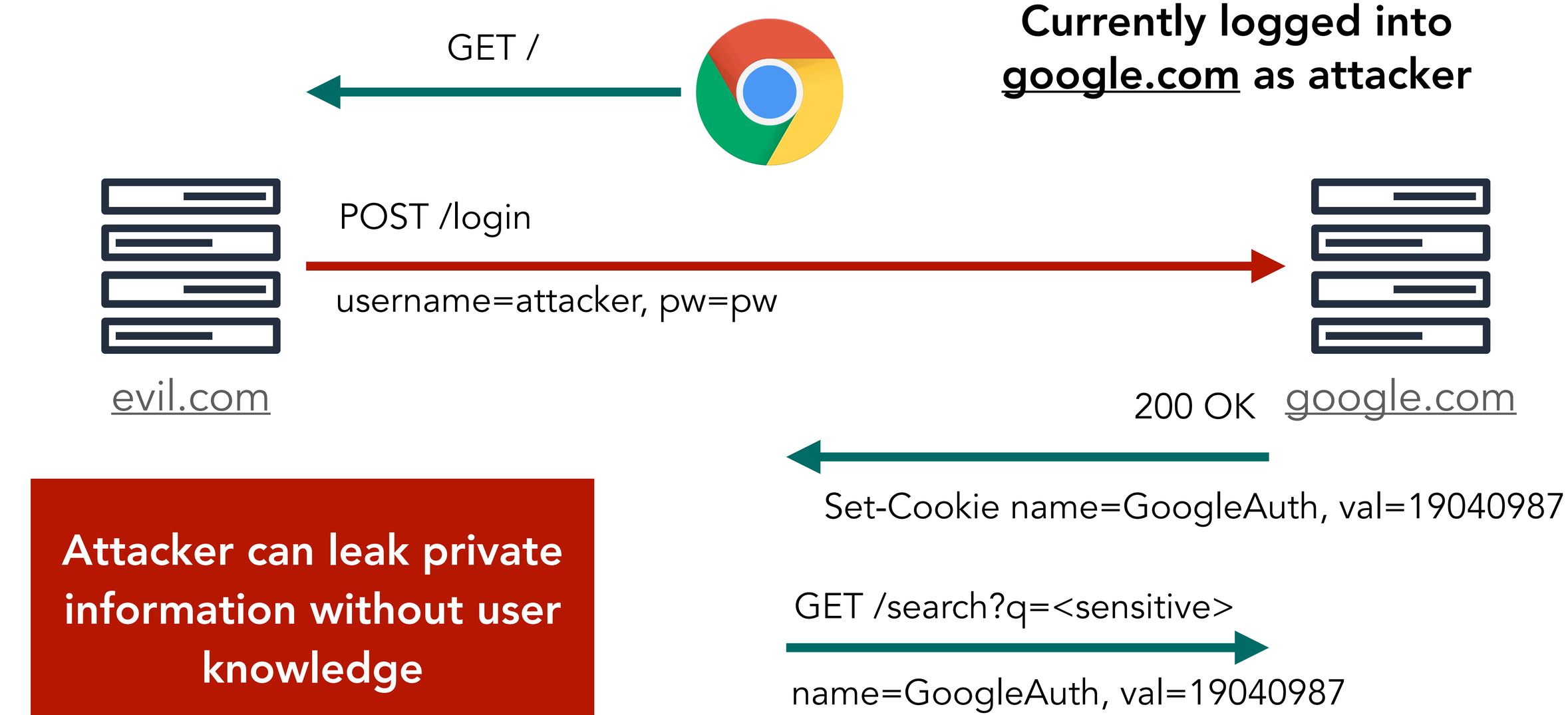


Why might an attacker want to do this?

Login CSRF (Special Case)



Login CSRF (Special Case)



Attacker can leak private information without user knowledge

CSRF Definition

- “Cross-site request forgery is an attack that forces an end-user to execute unwanted actions on a web application in which they’re currently authenticated” — OWASP
- **Deepak’s version:** CSRF lets you secretly masquerade as a user. Not good!
- Fundamentally enabled by cookie *side effects* (remind you of anything?)
  - Issue happens any place where the user’s browser has some kind of privileged access via the Web (not just cookies!)

Drive-by Pharming

- Home networks generally use “private” addresses (e.g., 192.168.X.X), only reachable inside the home
- Attack strategy
  - User visits malicious site. JavaScript scans home network looking for router (usually at 192.168.1.1)
  - Once JavaScript finds the router, can replace firmware or change DNS to attacker-controlled server
    - Many home routers have easily guessable passwords, e.g., admin:admin

CSRF Defenses

- How do we defend against these attacks? We need to ensure that **POST** is authentic — i.e., coming from a trusted page
  - Secret CSRF tokens
  - Referer/Origin Validation
  - SameSite Cookies

Secret CSRF Tokens

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

```
<form action="/login" method="post" class="form login-form">
<input type="hidden" name="csrf_token" value="434ec7e838ec3167efc04154205">
<input id="login" type="text" name="login" >
<input id="password" type="password">
<button class="button button--alternative" type="submit">Log In</button>
</form>
```

Secret CSRF Tokens

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

```
<form action="/login" method="post" class="form login-form">
<input type="hidden" name="csrf_token" value="434ec7e838ec3167efc04154205">
<input id="login" type="text" name="login" >
<input id="password" type="password">
<button class="button button--alternative" type="submit">Log In</button>
</form>
```

- Very commonly used defense against CSRF attacks. Any issues?

Secret CSRF Tokens

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

```
<form action="/login" method="post" class="form login-form">
<input type="hidden" name="csrf_token" value="434ec7e838ec3167efc04154205">
<input id="login" type="text" name="login" >
<input id="password" type="password">
<button class="button button--alternative" type="submit">Log In</button>
</form>
```

- Very commonly used defense against CSRF attacks. Any issues?
  - Implementation fails (server poorly checks, token only checked sometimes, etc.)
  - Token never rotated
  - Side channels for token validation (e.g., Spectre)

Defeating CSRF with the Referer header

- By default (usually), when the browser makes an HTTP request, it contains the *Referer*, aka the URL of the webpage that is making the request
  - Validation of the Referer header could easily defend against CSRF attacks
- Why does validation with the Referer header **not** work all the time?

Defeating CSRF with the Referer header

- By default (usually), when the browser makes an HTTP request, it contains the *Referer*, aka the URL of the webpage that is making the request
  - Validation of the Referer header could easily defend against CSRF attacks
- Why does validation with the Referer header **not** work all the time?
  - Fail-open: Allow requests where there is no Referer header
  - Fail-closed: Block requests where there is no Referer header

Extension: Origin header

Origin

Baseline Widely available

The HTTP `Origin` request header indicates the origin (scheme, hostname, and port) that *caused* the request. For example, if a user agent needs to request resources included in a page, or fetched by scripts that it executes, then the origin of the page may be included in the request.

Today's Defenses: SameSite Cookies

Set-Cookie

Baseline Widely available *

The HTTP `Set-Cookie` response header is used to send a cookie from the server to the user agent, so that the user agent can send it back to the server later. To send multiple cookies, multiple `Set-Cookie` headers should be sent in the same response.

`SameSite=<samesite-value>` Optional

Controls whether or not a cookie is sent with cross-site requests, providing some protection against cross-site request forgery attacks (CSRF).

SameSite Cookies

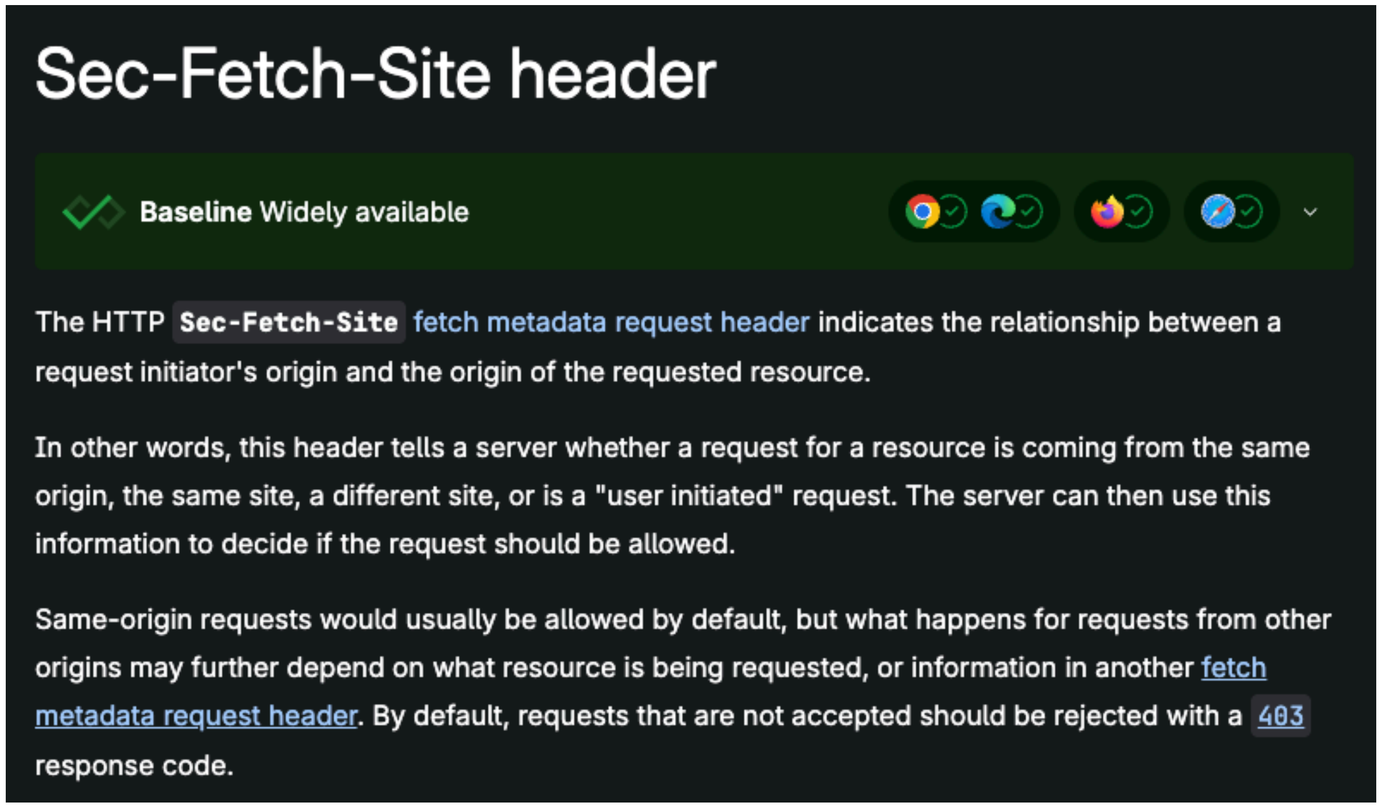
- Cookie option that prevents browser from sending a cookie along with cross-site requests
- **SameSite = Strict** Never send a cookie in a cross-site browsing context, even when following a regular link
- **SameSite = Lax** Session cookie is allowed when following a navigation link but blocks it in CSRF-prone request methods, like POST (default)
- **SameSite = None** Send cookies from any context
- Why might this not always work?

SameSite Cookies

- Cookie option that prevents browser from sending a cookie along with cross-site requests
- **SameSite = Strict** Never send a cookie in a cross-site browsing context, even when following a regular link
- **SameSite = Lax** Session cookie is allowed when following a navigation link but blocks it in CSRF-prone request methods, like POST (default)
- **SameSite = None** Send cookies from any context
- Why might this not always work?
  - Server has to trust browser to implement correctly. And they might not.

CSRF Defenses Today

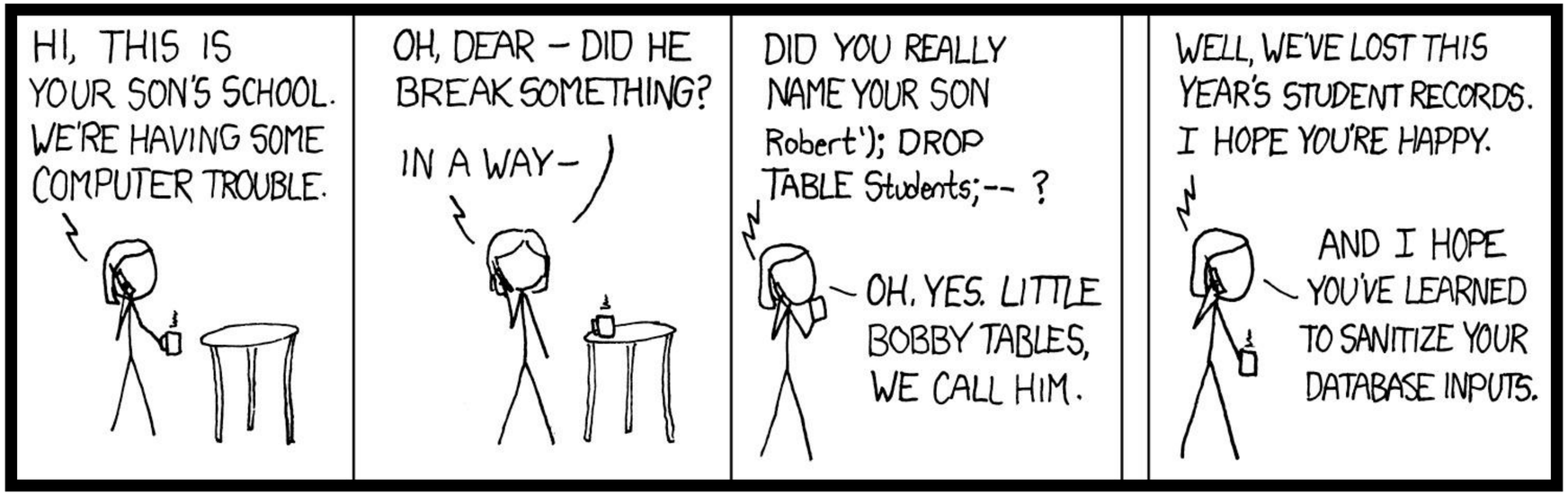
- Defense in depth — usually some combination of all three defenses
- New paradigm: Fetch metadata



https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Sec-Fetch-Site

SQL Injection (SQLi)

- Websites often rely on *databases* to properly function (e.g., SQL)
- Websites can be vulnerable to **command injection attacks** when using **user-provided data** to build SQL queries



SQL Basics

- Structured Query Language (SQL)
- Example
  - `SELECT * FROM users where username is kumarde and pw is ilovebooks`
- Other operators too
  - `AND, OR, NOT,` logical expressions
  - Two dashes `(--)` indicates a comment (until line end)
  - `;` is a statement terminator

Building SQL from user input

```
import sqlite3

conn = sqlite3.connect("users.db")
cursor = conn.cursor()

username = input("Username: ")
password = input("Password: ")

query = f"""
SELECT * FROM users
WHERE username = '{username}' AND password = '{password}'
"""

cursor.execute(query)
result = cursor.fetchone()
```

Building SQL from user input

```
import sqlite3

conn = sqlite3.connect("users.db")
cursor = conn.cursor()

username = input("Username: ")
password = input("Password: ")

query = f"""
SELECT * FROM users
WHERE username = 'kumarde' AND password = 'ilovebooks'
"""

cursor.execute(query)
result = cursor.fetchone()
```

username = kumarde

password = ilovebooks

Building SQL from user input

```
import sqlite3

conn = sqlite3.connect("users.db")
cursor = conn.cursor()

username = input("Username: ")
password = input("Password: ")

query = f"""
SELECT * FROM users
WHERE username = '{username}' AND password = '{password}'
"""

cursor.execute(query)
result = cursor.fetchone()
```

username = kumarde’

password = ilovebooks

What happens?

Building SQL from user input

```
import sqlite3

conn = sqlite3.connect("users.db")
cursor = conn.cursor()

username = input("Username: ")
password = input("Password: ")

query = f"""
SELECT * FROM users
WHERE username = 'kumarde'' AND password = 'ilovebooks'
"""

cursor.execute(query)
result = cursor.fetchone()
```

username = kumarde’

password = ilovebooks

What happens?

Building SQL from user input

```
import sqlite3

conn = sqlite3.connect("users.db")
cursor = conn.cursor()

username = input("Username: ")
password = input("Password: ")

query = f"""
SELECT * FROM users
WHERE username = 'kumarde'' AND password = 'ilovebooks'
"""

cursor.execute(query)
result = cursor.fetchone()
```

username = kumarde’

password = ilovebooks

What happens?

Program crash!

Building SQL from user input

```
import sqlite3

conn = sqlite3.connect("users.db")
cursor = conn.cursor()

username = input("Username: ")
password = input("Password: ")

query = f"""
SELECT * FROM users
WHERE username = '{username}' AND password = '{password}'
"""

cursor.execute(query)
result = cursor.fetchone()
```

username = admin

password = ‘ OR ‘1’=‘1

What happens?

Building SQL from user input

```
import sqlite3

conn = sqlite3.connect("users.db")
cursor = conn.cursor()

username = input("Username: ")
password = input("Password: ")

query = f"""
SELECT * FROM users
WHERE username = 'admin' AND password = '' OR '1'='1'
"""

cursor.execute(query)
result = cursor.fetchone()
```

username = admin

password = ‘ OR ‘1’=‘1

What happens?

Building SQL from user input

```
import sqlite3

conn = sqlite3.connect("users.db")
cursor = conn.cursor()

username = input("Username: ")
password = input("Password: ")

query = f"""
SELECT * FROM users
WHERE username = 'admin' AND password = '' OR '1'='1'
"""

cursor.execute(query)
result = cursor.fetchone()
```

username = admin

password = ‘ OR ‘1’=‘1

What happens?

Attacker can successfully login as an admin user.

SQLi Attack Variants

- ‘; drop table users --
  - Deletes the users table from the database
- Any set of SQL commands will do
  - Can read fields, find elements, write tables
- What is the fundamental design flaw that enables SQL injection?

SQLi Attack Variants

- ‘; drop table users --
  - Deletes the users table from the database
- Any set of SQL commands will do
  - Can read fields, find elements, write tables
- What is the fundamental design flaw that enables SQL injection?
  - Mixing code and data… just like in buffer overflow attacks

Defending against SQL attacks

- **Don’t mix code and data.** Instead, use **prepared statements**

```
import sqlite3

conn = sqlite3.connect("users.db")
cursor = conn.cursor()

username = input("Username: ")
password = input("Password: ")

query = """
SELECT * FROM users
WHERE username = ? AND password = ?
"""

cursor.execute(query, (username, password))
result = cursor.fetchone()
```

Defending against SQL attacks

- **Don’t mix code and data.** Instead, use **prepared statements**
- Every language supports prepared statements, allowing you to independently process query inputs and SQL inputs
  - The database handles all escaping
  - User input is never incorrectly treated as SQL
- **Prepared statements are the industry standard… use them.**

Cross Site Scripting (XSS)

- “Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites” – OWASP
- Where SQL injection is a piece of malicious code executed on the victim’s server…
- XSS is malicious code executed on a victim’s browser.

Cross Site Scripting (XSS)

- Key idea: Indirect attack on browser *via* a server
- Malicious content is injected via URL encoding (query parameters, form submission) and **reflected back** by the server in the response
- Browser then **executes code** server provided

Very simple XSS example

Web Server

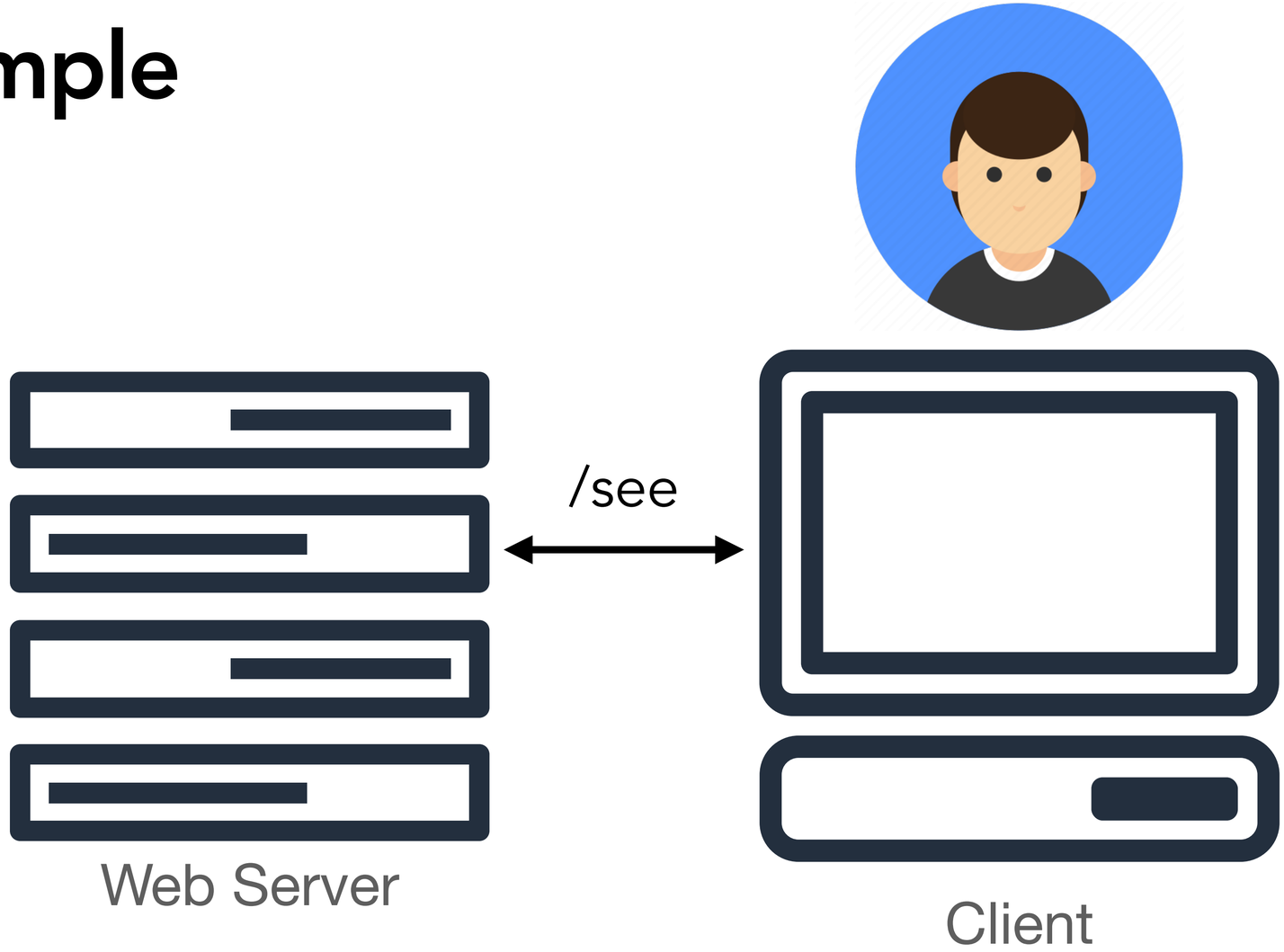
Client

Very simple XSS example

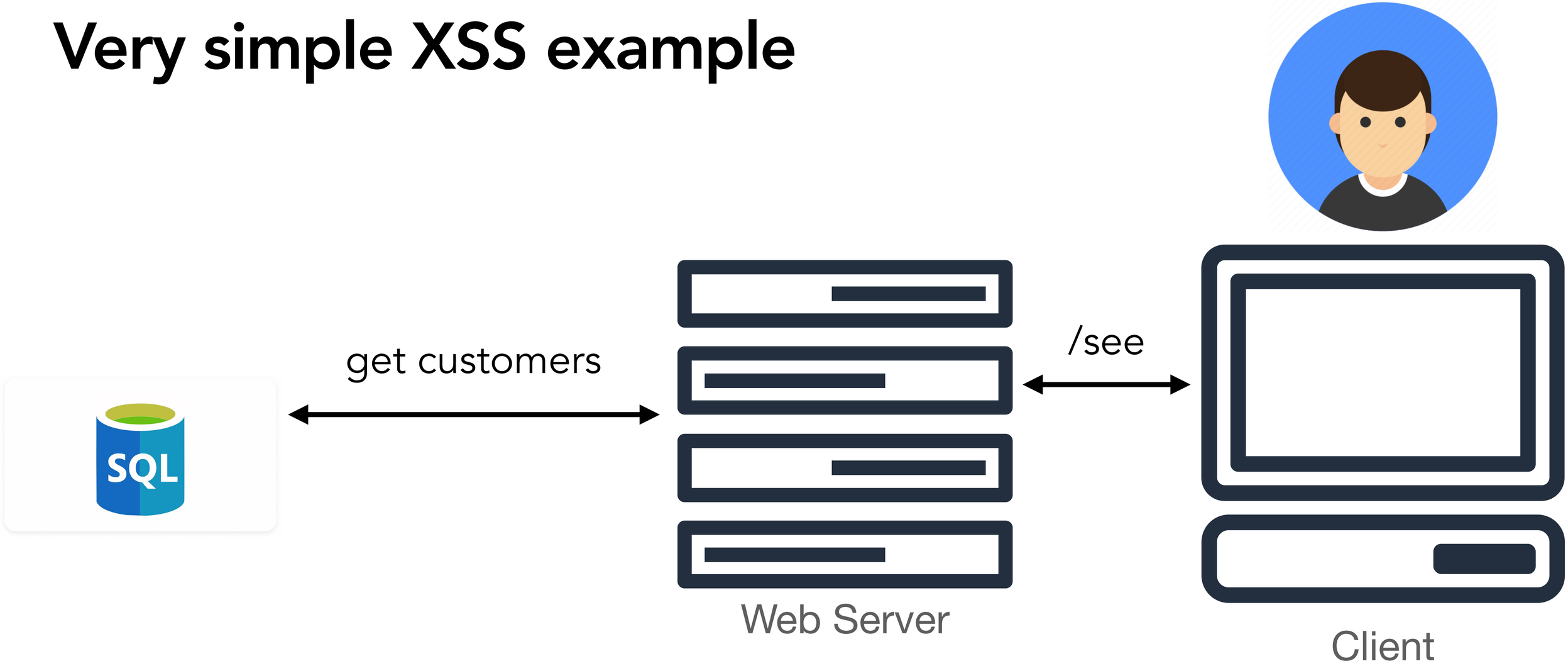
Web Server

Client

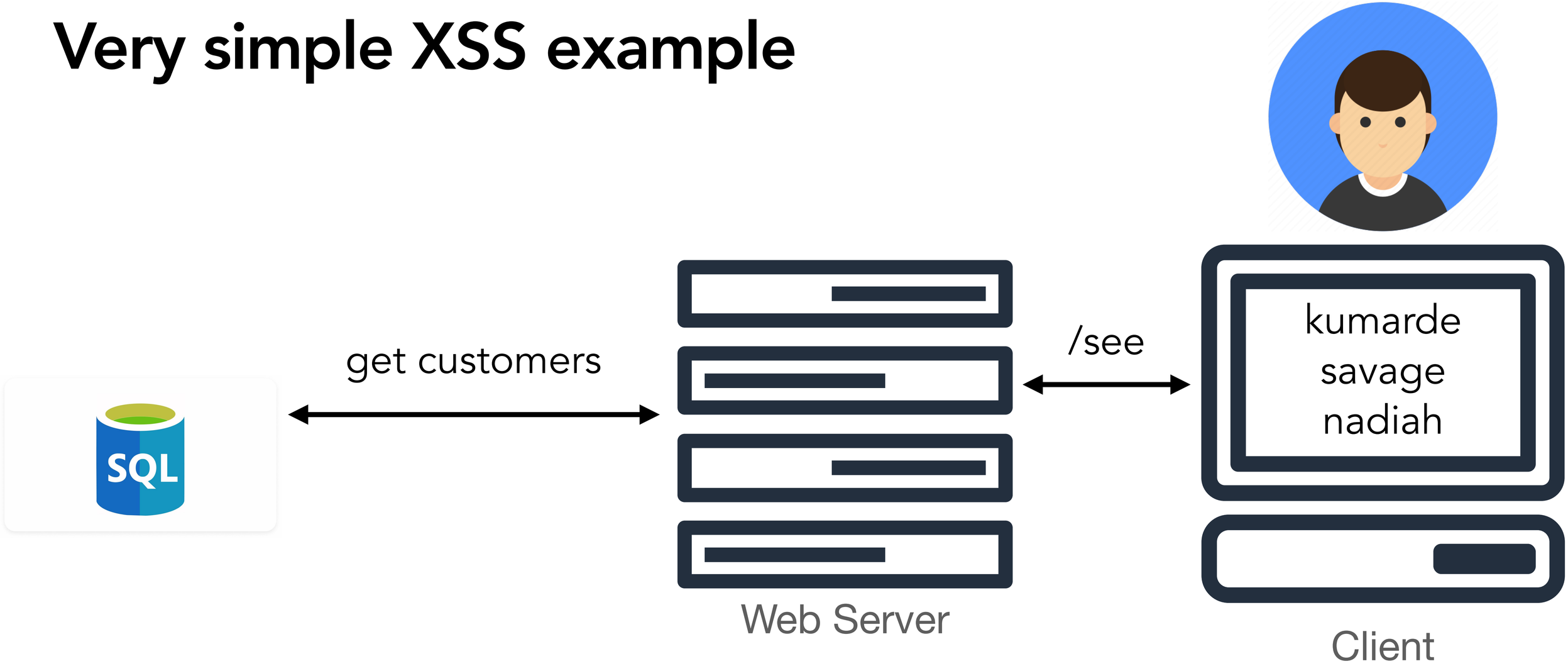
Very simple XSS example



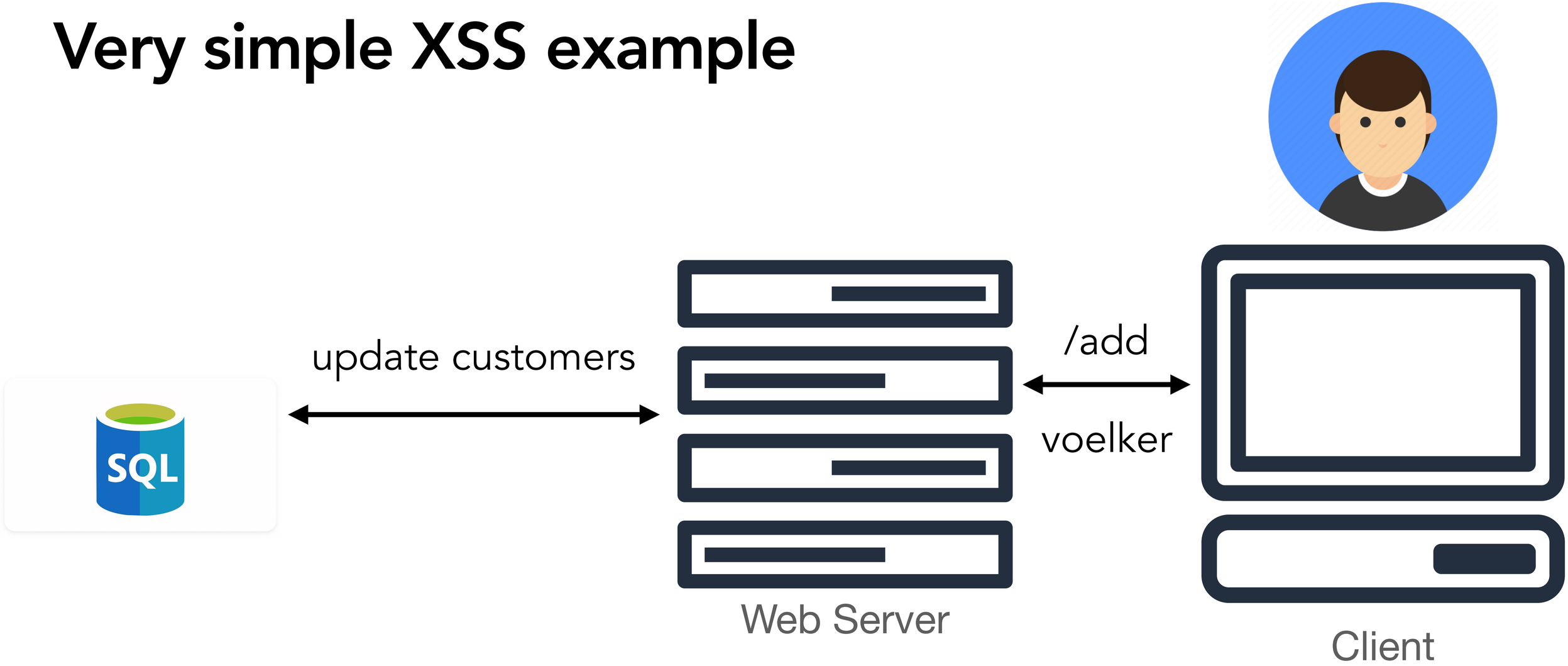
Very simple XSS example



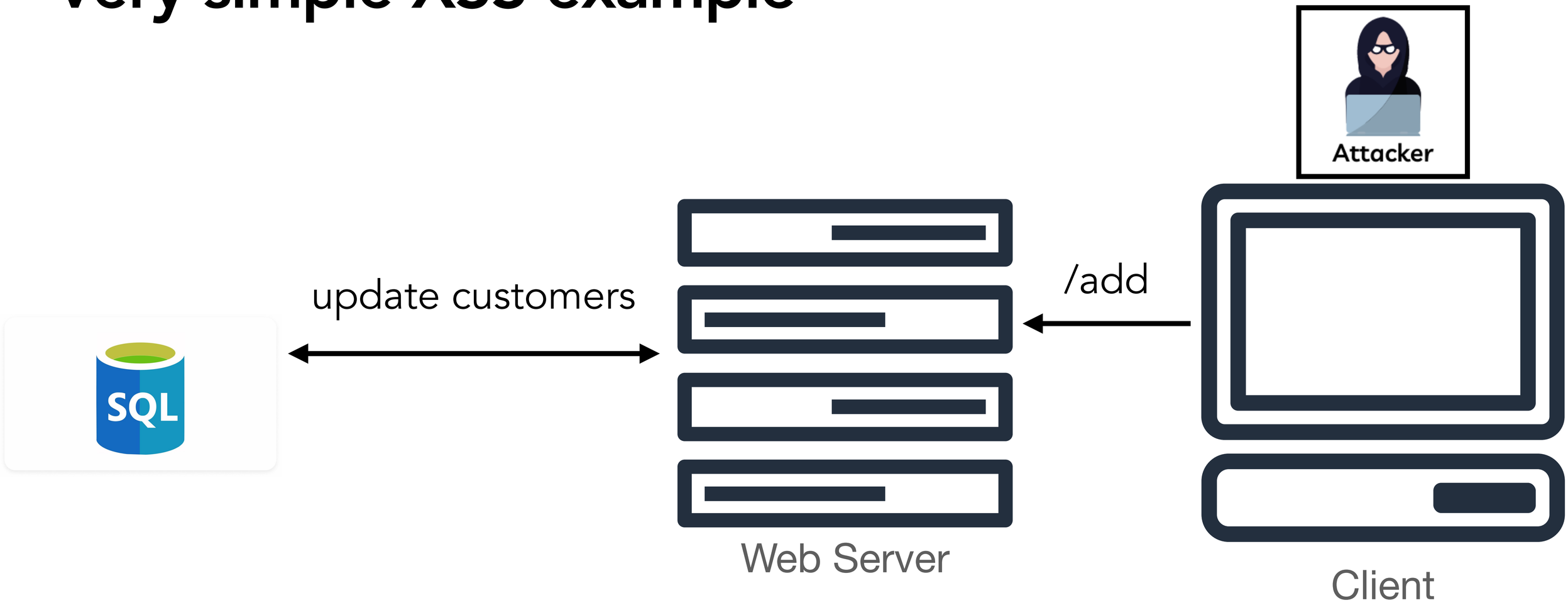
Very simple XSS example



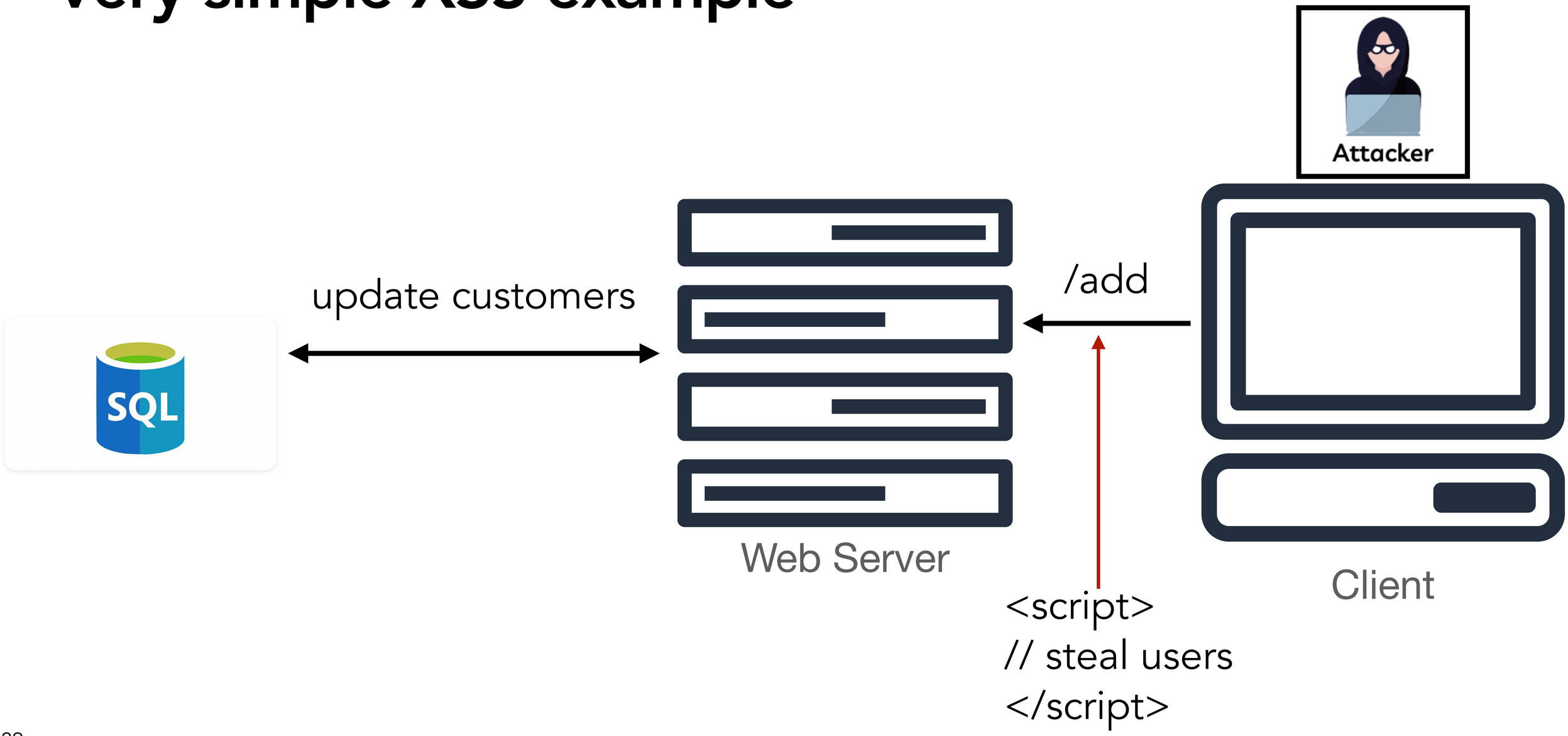
Very simple XSS example



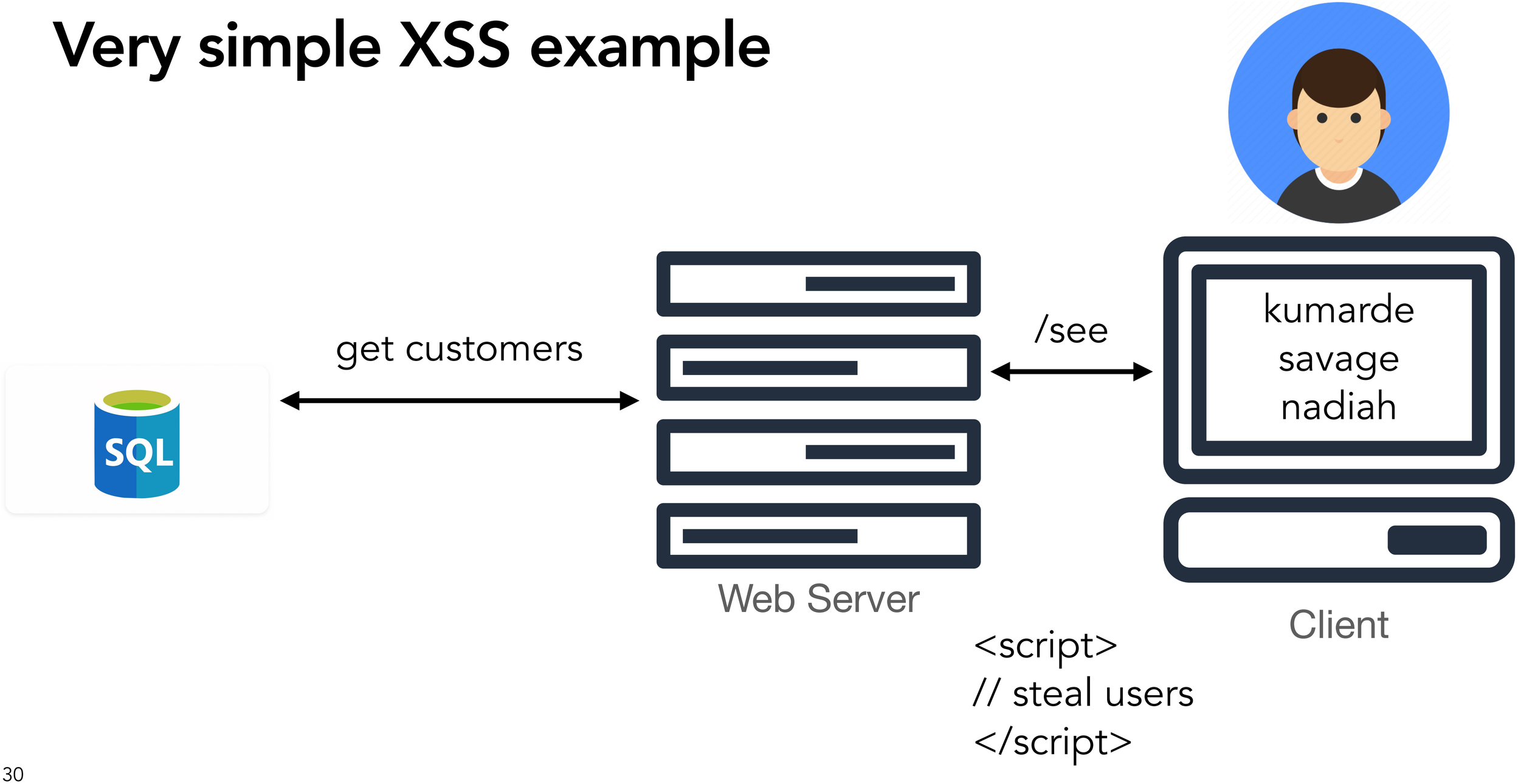
Very simple XSS example



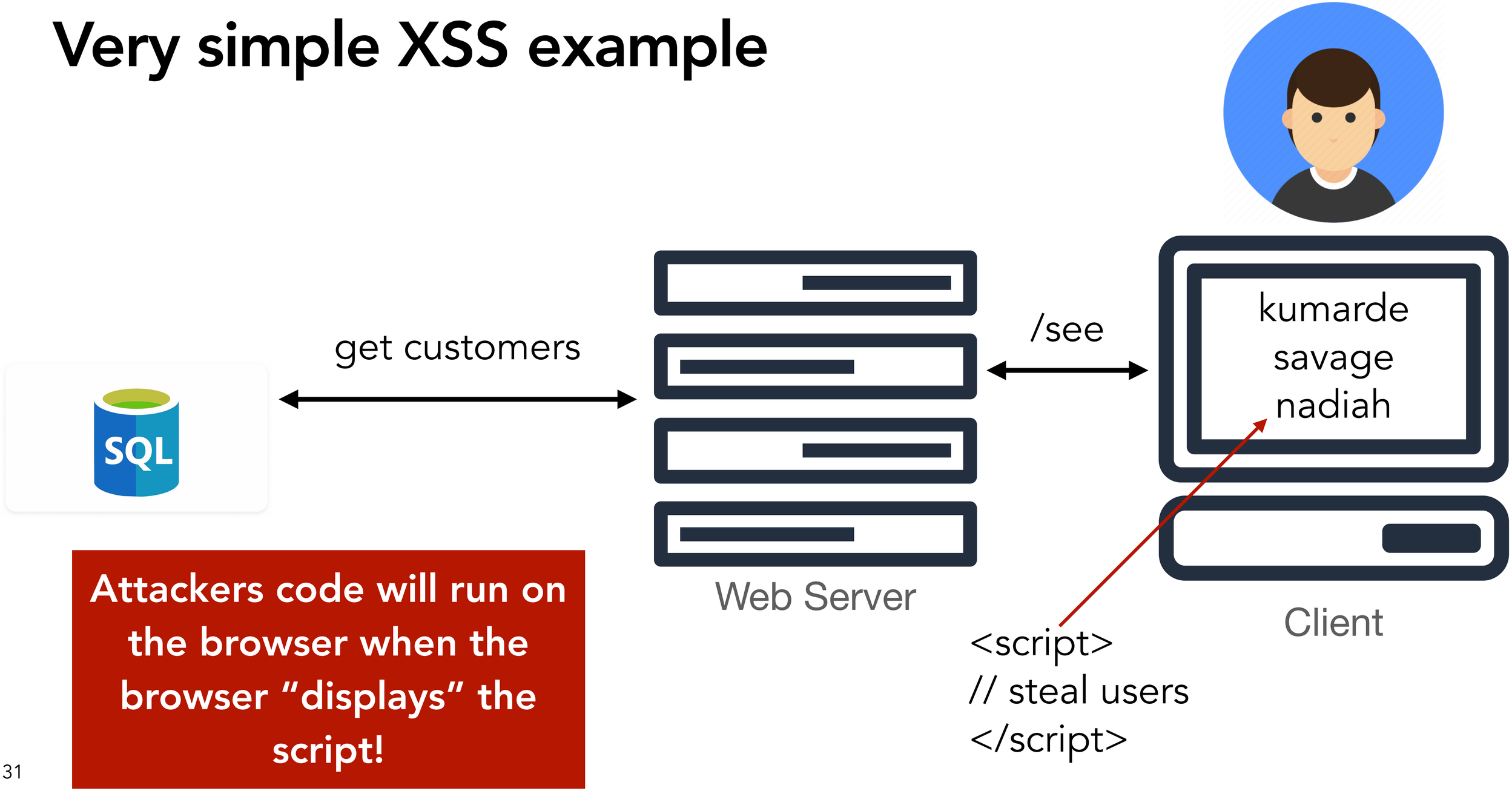
Very simple XSS example



Very simple XSS example



Very simple XSS example



Attackers code will run on the browser when the browser "displays" the script!

Preventing XSS: filtering

- Key problem: rendering raw HTML from user input
  - Let's just filter it!
- Very hard in practice.
  - Blocking "<" and ">" is not enough; lots of ways to get code to execute in a browser…
  - Event handlers, other tags, not just script tags…
- Example: filter out <script
  - <script src="…">
  - <scr<scriptip src="…">

Preventing XSS: Content Security Policy

- Content Security Policy eliminates XSS by specifying the domains that the browser should consider to be valid sources of executable scripts
  - `Content-Security-Policy: default-src 'self' (means content can only be loaded from exact same domain, no inline scripts)`
  - `Content-Security-Policy: default-src 'self', img-src *; media-src media1.com; script-src good.com`
- CSP is served via HTTP headers, or can be embedded in pages via meta HTML object in DOM
- **Modern standard defense against XSS attacks**

Recap + next time…

- Web is windy, twisted, complicated, and hard to reason about
  - Lots of growth in web comes from use cases, as those evolved, so too did security
- Evergreen lesson: **mixing code and data is bad**
  - Double evergreen lesson: Sanitize inputs, but don’t do it yourself (libraries will help you here)
- You’ll implement all these attacks in PA3
- Next time we’ll talk about web measurement and how you’re tracked on the web — two of my favorite topics!