Web Security

https://stevekinney.net/courses/security

Steve Kinney

The good part of the Web is *also* the bad part.

Even on it’s best day:

The Web is *weird*.

Very few—if any—of us think that security is *not* important.

In the *Real World*®,
it’s all about trade-offs.

"Oh, well we had to do `$thing` in order to get `$feature` working."

—Famous last words in a code review.

How do we know what we know?

If you're doing it well, then *no one notices* your hard work.

"I don't need to learn about web security. `$framework` just handles all of this stuff for me."

—Someone who is accidentally about to create a security hole.

So, let's *dig in*. In this course, we're going to cover…

So, let's *dig in*. In this course, we're going to cover…

Common *vulnerabilities* found in many web applications.

So, let's *dig in*. In this course, we're going to cover...

The ways in which attackers can *exploit* those vulnerabilities.

So, let's *dig in*. In this course, we're going to cover...

A *reasonably deep understanding* of the tools provided by the browser and best practices on the server to protect yourself from these vulnerabilities.

So, let's *dig in*. In this course, we're going to cover…

The *trade-offs* between security, user experience, and the complexity of our infrastructure.

You can be *anyone* on the Internet.

Cookies

Small pieces of data stored on the client-side to track and identify users.

Cookies

HTTP is *stateless*. Cookies are used to implement *sessions*.

Keeping track of the **currently authenticated user**.

Storing the contents of a **shopping cart**.

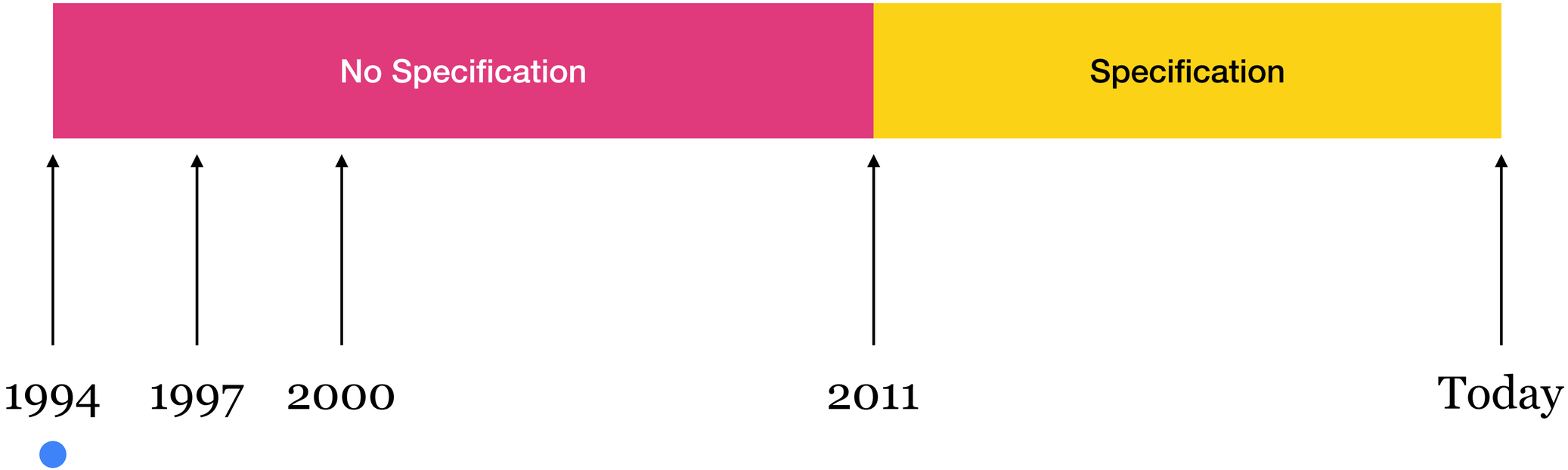
Or, more nefariously: **tracking users**.

Like many things on the web, the original implementation of cookies was *fast and loose*.

Cookies

A brief history.

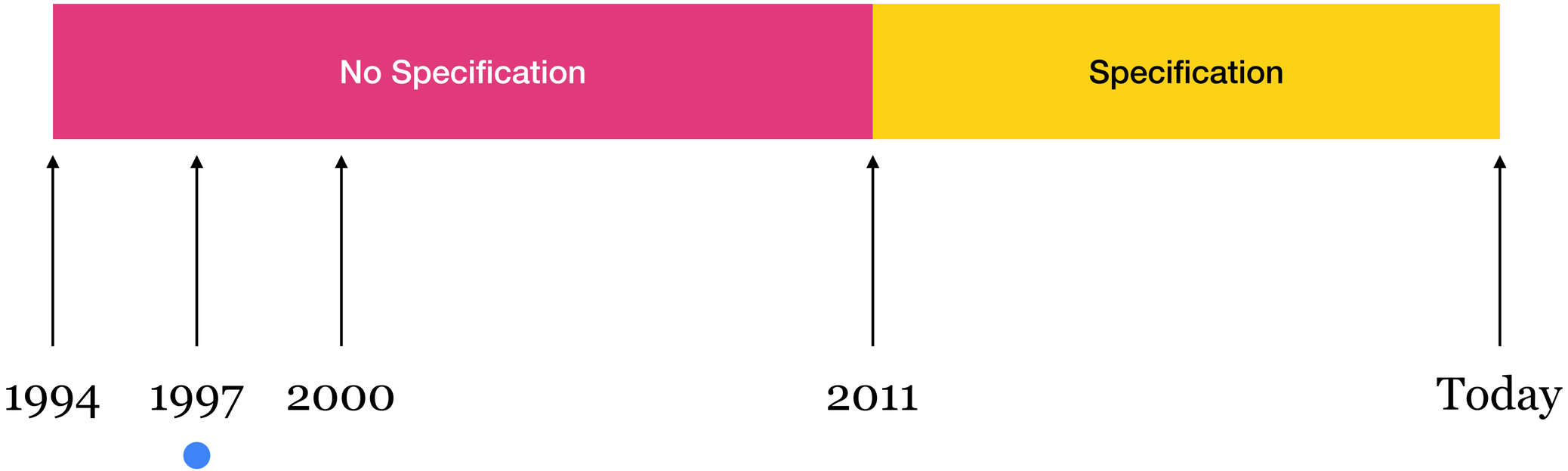
1994: Implemented by Netscape.



Cookies

A brief history.

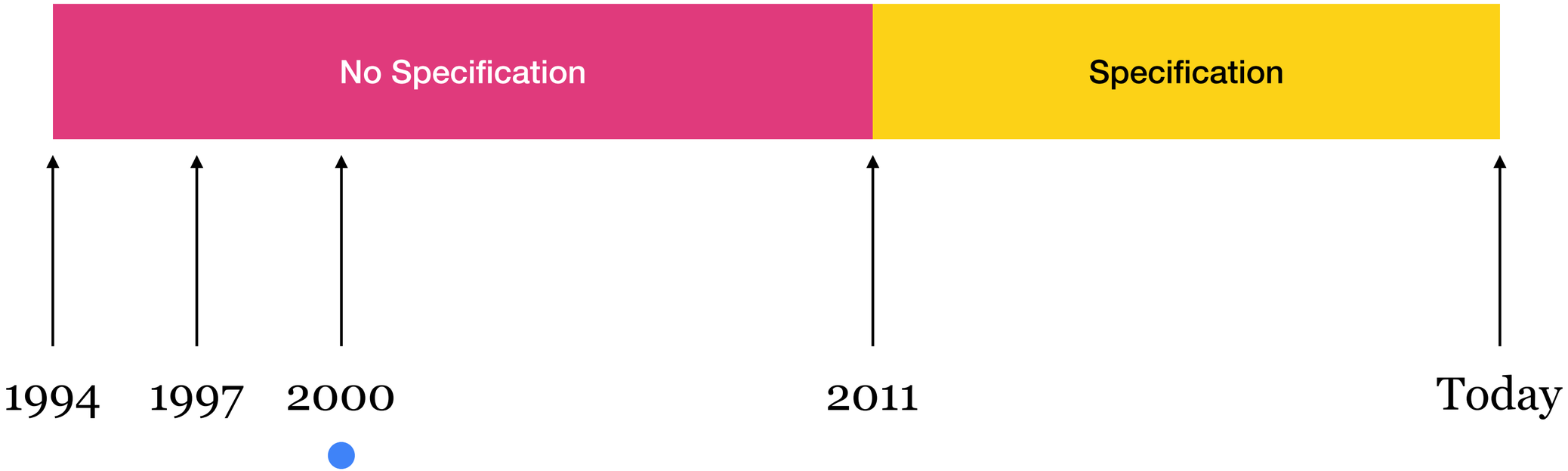
1997: First attempt to standardize.



Cookies

A brief history.

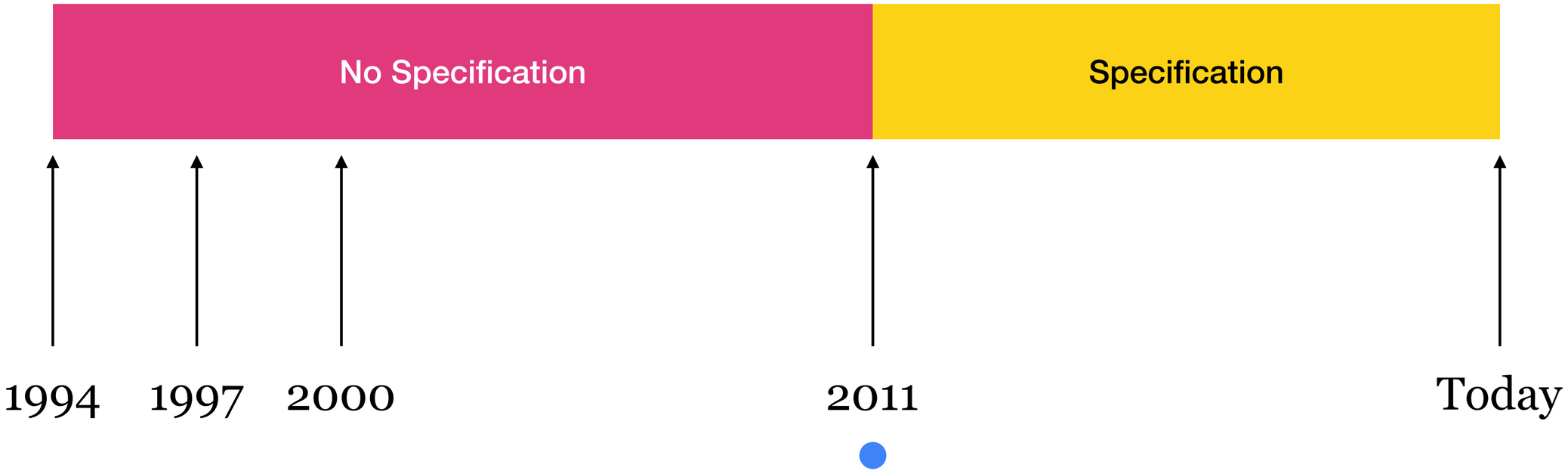
2000: “Cookie2”, a second attempt to standardize.



Cookies

A brief history.

2011: Standardized as ***RFC 6265***.



Cookies

The server sets a cookie with an HTTP header.

```
Set-Cookie: username=bobbytables;
```

HTTP Header **Key** **Value**

Cookies

The browser sends back the cookie on subsequent requests.

```
Cookie: username=bobbytables;
```

HTTP Header **Key** **Value**

Cookies

Accessing cookies from the browser—*sometimes*.

DevTools - localhost:3000/profile

Elements | Console | Sources | Performance insights | Network | Performance | Memory | Application | Security | 3

Application
- Manifest
- Service workers
- Storage

Storage
- Local storage
- Session storage
- IndexedDB
- Cookies
  - http://localhost:3000
- Private state tokens
- Interest groups
- Shared storage
- Cache storage

Background services
- Back/forward cache
- Background fetch
- Background sync
- Bounce tracking mitigations

Filter | Only show cookies with an issue

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite	Par...	Prior...
username	bobbytables	localhost	/	Session	19					Medi...
password	papayawhip	localhost	/	Session	18					Medi...

Cookie Value Show URL-decoded

bobbytables

Cookie Attributes

Expiration Dates.

Expires: `Set-Cookie: username=bobbytables; Expires=Thu, 29 Feb 2024 17:45:00 GMT;`

Max Age: `Set-Cookie: username=bobbytables; Max-Age=2592000;`

Without some kind of expiration, cookies last for the duration of the *session*.

Cookies

How to delete a cookie.

You *can't*.

But, you *can* set it's expiration date to some time in the past.

And, this *will* delete the cookie.

I *don't* make the rules around here.

Cookie Attributes

Scoping.

The `Path` attribute indicates a URL path that must exist in the requested URL in order to send the `Cookie` header.

```
Set-Cookie: username=bobbytables; Path=/profile;
```

This *predates* a lot of the modern browser securities features and you're better off *not* relying on it for security.

Cookie Attributes

Scoping.

The `Domain` attribute specifies which server can receive a cookie. If specified, cookies are available on the specified server and its subdomains.

```
Set-Cookie: username=bobbytables;
Domain=.frontendmasters.com;
```

This would allow the cookie to be sent to *any* subdomain of `frontendmasters.com`.

Demonstration

Cookie Jar

Same Origin Policy

A security measure implemented by web browsers to restrict how documents or scripts loaded from one origin interact with resources from other origins.

Two resources from different sites shouldn’t be able to interfere with each other.

Protocol + Host + Port

Same Origin Policy

How it works.

The browser checks *three* things: the **protocol** (e.g. `https://` vs `http://`), the **domain** (e.g. `frontendmasters.com`) and the **port** (e.g. 443).

If those three things are the same, then the browser considers the two resources to have the **same origin**.

Same Origin Policy

Ways around it.

Cross Origin Resource Sharing (CORS)

JSON with Padding (JSONP)

Proxies

PostMessage API

WebSockets

`document.domain`

Cookies

Some common vulnerabilities.

Session Hijacking: Exploits active sessions to gain unauthorized access. Basically, the attacker can *become* the user as far as the server is concerned.

Cross-Site Scripting (XSS): A Malicious script injected via input fields or URLs. The script then accesses cookies and sends them to an attacker.

Cross-Site Request Forgery (CSRF): An innocent user is tricked into executing actions via a forged request, exploiting that user's authenticated session.

And now…

Session Hijacking

Privilege Escalation

A type of security exploit where an attacker gains elevated access to resources that are normally protected from an application or user.

Privilege Escalation

How it works.

Initial Access: The attacker gains limited access tot he application or system.

Discovering Weaknesses: The attacker identifies misconfigurations or vulnerabilities that can be exploited.

Exploiting Vulnerabilities: They then use these weaknesses to gain higher privileges than originally intended.

Gaining Control: With escalated privileges, they can access sensitive data or execute unauthorized actions.

Session Hijacking

Using a cookie value in an attempt to try to trick the server into thinking that you're someone that you're not.

Man-in-the-Middle Attacks

A type of attack where the attacker secretly intercepts and possibly alters the communication between two parties who believe they are directly communicating with each other.

Luckily, this has a pretty easy fix: *Use HTTPS*.

(We'll talk a little more about this later.)

Cookie Attributes

Security.

`HTTPOnly`: Prevents client-side scripts from accessing cookies.

`Secure`: Ensures cookies are sent over HTTPS only.

`SameSite`: Restricts how cookies are sent with cross-origin requests.

Cookie Attributes

The SameSite attribute.

It's *not* the same as the origin.

TLD+1: Two addresses are the SameSite if they have the same TLD plus one more level:

example.com

login.example.com

Exceptions: github.io and similar.

Injection Attacks

A class of vulnerabilities caused by the attacker sending malicious inputs to your application is homes of breaking out and causing some damage.

And now…

Other Types of Injection Attacks

Command Injection

A type of security vulnerability where an attacker can execute arbitrary commands on the host operating system via a vulnerable application.

Command Injection

How it works.

Target: Applications that pass unsafe user inputs to system shell commands.

User Input: Application receives input from a user.

Unsanitized Input: The user input is concatenated with a system command in a way that allows additional commands to be executed.

Shell Execution: The combined command is executed in the system command shell.

Malicious Payload: Attackers inject malicious commands, piggybacking on legitimate commands.

Command Injection

A somewhat silly example that fits on a slide.

```
import { exec } from 'child_process';

app.get('/files', (req, res) => {
    const command = `ls ${req.body}`;

    exec(command, (error, stdout, stderr) => {
        if (error) {
            return res.status(500).send(error.message);
        }

        if (stderr) {
            return res.status(500).send(stderr);
        }

        res.send(stdout);
    });
});
```

User Input: `test.txt; rm -rf /`

Command Injection

Remediation.

First: Just don't.

Second: Use a built-in Node module (e.g. `fs`) if you *really* need to.

Third: Use `execFile` if you *really* need to.

Fourth: Sanitize, allowlist, and use the principle of least privilege if you *really* need to.

File Upload Vulnerabilities

A class of security flaws that occur when a web application improperly handles the uploading of files.

Remote Code Execution

This one kind of does what it says on the tin. Remote Code Execution (RCE) is a vulnerability that allows an attacker to execute arbitrary code on a remote system.

Remote Code Execution

How it works.

Injection: RCE typically occurs when the user input is not properly sanitized.

Processing: The malicious input is processed by the server through `eval` functions, un-sanitized inputs to shell commands, or unsafe deserialization.

Execution: Malicious code is executed within the server environment, providing the attacker with control over the server resources.

Remote Code Execution

Steps to prevent bad things from happening.

Input Validation: Always validate and sanitize user inputs. Use strict data types and constraints.

Don't Do Dangerous Stuff: Avoid using `eval()`, `Function()`, `exec()`, and other potentially dangerous Node.js functions.

Use Security Libraries: Utilize libraries such as `DOMPurify` for sanitizing HTML. Use `sandboxed` environments like VM2 for executing untrusted code.

Principle of Least Privilege: Run services with the minimal required permissions. Do not run your application as a root user.

Update Your Stuff: Keep Node.js and all dependencies up-to-date to mitigate known vulnerabilities.

Cross-Site Request Forgery

A vulnerability that allows an attacker to make unauthorized requests on the user's behalf.

Side effects include: unauthorized actions like changing account settings, making purchases, and lots of other ***BAD STUFF™***.

Case Study

The Twitter (CSRF) Worm (2010)

Twitter has a URL that would allow you to create a post with a URL.

```
http://twitter.com/share/update?status=Your%20Message
```

Case Study

Netflix (2006)

Netflix had a `GET` endpoint that allowed you to add a DVD to your queue. It was as easy as...

```
<img src="http://www.netflix.com/JSON/AddToQueue?
movieid=70110672" />
```

Cross-Site Request Forgeries: Exploitation and Prevention

William Zeller* and Edward W. Felten*†

*Department of Computer Science

*Center for Information Technology Policy

†Woodrow Wilson School of Public and International Affairs

Princeton University

{wzeller,felten}@cs.princeton.edu

Case Study

The New York Times (2008)

The *New York Times* has a form on every article called *Email This*. This form made a `POST` request that was vulnerable to CSRF allowing the attacker to have the authenticated user send an email to the attacker, exposing their name and email address.

Case Study

ING Direct (2008)

The researchers also discovered CSRF vulnerabilities in ING's site that allowed an attacker to open additional accounts on behalf of a user and transfer funds from a user's account to the attacker's account.

Case Study

YouTube (2008)

They also discovered CSRF vulnerabilities in nearly every action a user can perform on YouTube.

Case Study

TikTok One-Click Account Takeover (2020)

A specific endpoint on TikTok, which handled password resets for accounts created via third-party applications, was susceptible to CSRF attacks.

The vulnerability could be exploited to change the user's password.

Cross-Site Request Forgery

Consider this example CSRF request.

```
POST /delete-account HTTP/1.1

Host: supercoolwebsite.com

Content-Type: application/x-www-form-urlencoded

Cookie: sessionId=de4db33f


delete=1
```

```
function createAndSubmitForm() {
  const form = document.createElement('form');

  form.method = 'POST';
  form.action = 'https://example.com/delete-account';
  form.style.display = 'none';

  document.body.appendChild(form);

  form.submit();
}

createAndSubmitForm();
```

Cross-Site Request Forgery

The three ingredients to the recipe.

A relevant action. There has to be something interesting to the attacker. Relevant actions can include: email or password changes, balance transfers, etc.

Cookie-based session handling. In a CSRF attack, the attacker is tricking *you* into accidentally performing action with your very legitimate session authentication without your knowledge.

No unpredictable parameters. The attacker needs to be able to guess what it ought to send to get the desired outcome.

The interesting part about CSRF attacks is that they don't *really* require any JavaScript.

The attacker also doesn’t need to access to *your* site.

Cross-Site Request Forgery

How does it work?

User Authentication: User logs into a web application and receives an authentication token (e.g. a cookie).

Malicious Site Visit: User visits a malicious website while still authenticated.

Malicious Request: The malicious website contains code that sends a request to the authenticated web application.

Unauthorized Action: The web application processes the request as if it were made by the user.

Demonstration

Implementing a CSRF Attack

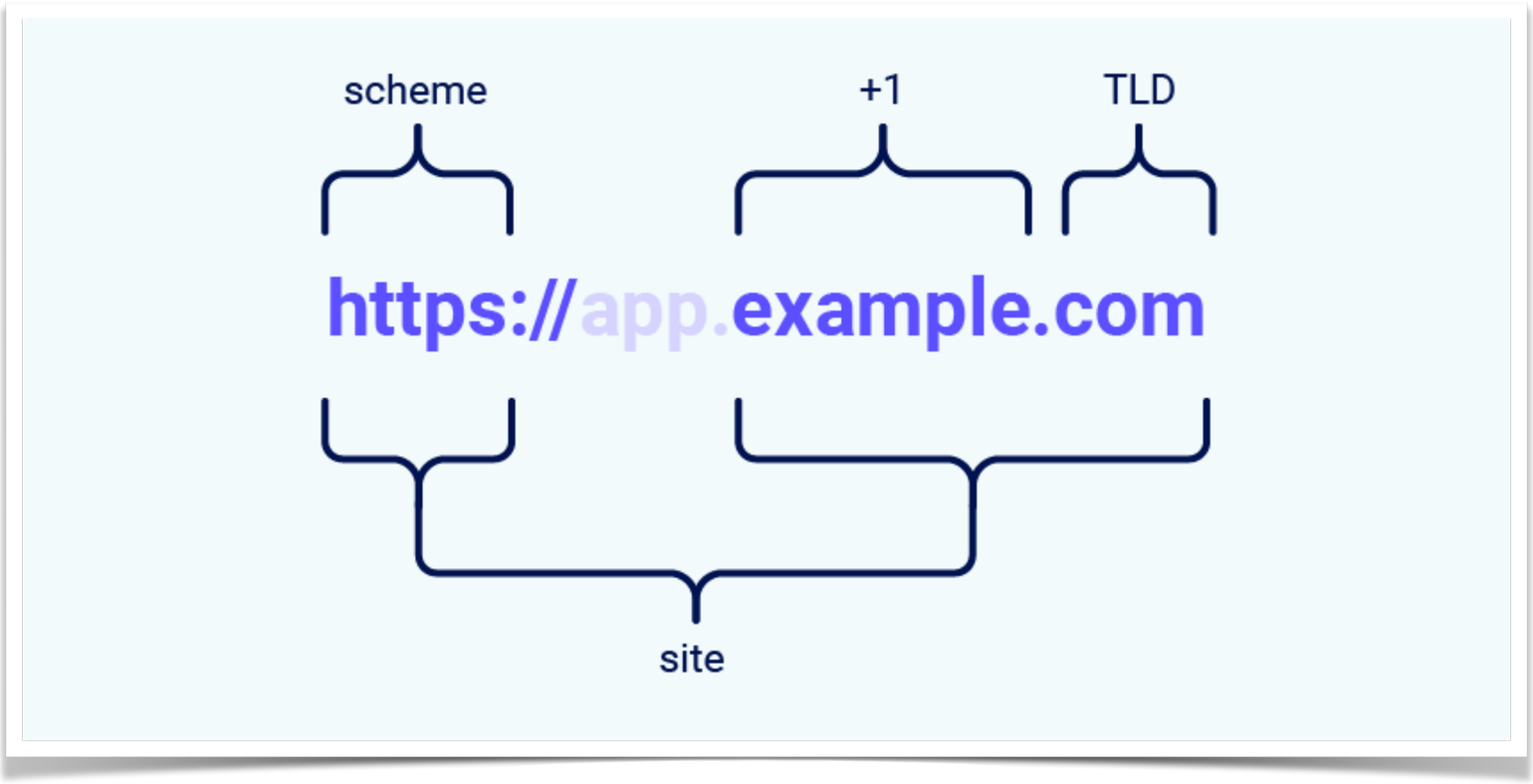
Cookie Attributes

The SameSite attribute.

`SameSite=None`: Always send the cookie.

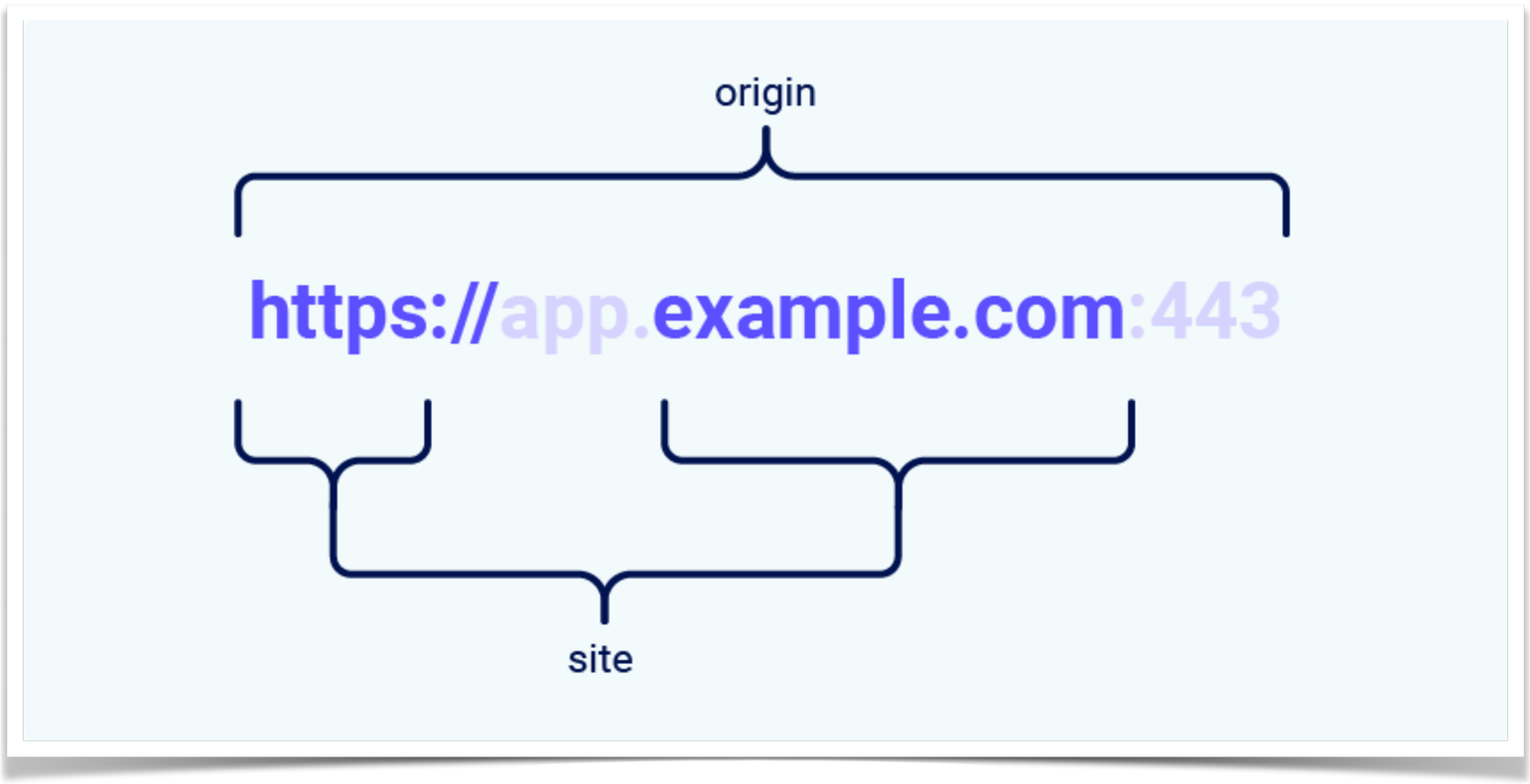
`SameSite=Lax`: Allows cookies to be sent with top-level navigations and if the request method is safe (e.g. `GET` or `HEAD` requests).

`SameSite=Strict`: *Only* include the cookie if the request is sent from the same site that set the cookie.



Source: https://portswigger.net/web-security/csrf/bypassing-samesite-restrictions

Site ≠ *Origin*



Source: https://portswigger.net/web-security/csrf/bypassing-samesite-restrictions

Top-Level Navigation

An action where a user initiates a change in the URL through direct methods like entering a URL in the address bar, selecting a bookmark, or clicking on a link that leads to another page, effectively changing the entire browser window's content to load the new page.

Top-Level Navigation

TL;DR—*Not* AJAX requests or iFrames.

Cookie Attributes

`Lax` versus `Strict`: A balance of trade-offs.

`SameSite=Lax` allows cookies to be sent with top-level navigations that are triggered by external sites.

This can be important for *legitimate* use cases (e.g., a link in an email or from another domain).

Cookie Attributes

`Lax` versus `Strict`: A balance of trade-offs.

While `SameSite=Strict` provides the highest level of security by restricting all cross-site cookie sharing (including through top-level navigations), it can disrupt user workflows.

For example, if a user follows a link from an email or another site to a `SameSite=Strict` site where they are normally logged in, they would find themselves unexpectedly logged out.

`SameSite=Lax` prevents this issue, providing a reasonable balance between preventing CSRF attacks and maintaining user session continuity across sites.

Cookie Attributes

`Lax` versus `Strict`: A balance of trade-offs.

Websites often interact with external services, affiliates, or rely on marketing and promotional campaigns that direct traffic to their sites through links. `SameSite=Lax` supports these business needs without sacrificing significant security, allowing services to operate smoothly across different domains.

Cookie Attributes

`SameSite=Lax` is now the default.

March 2019: Safari 12.1.

January 2020: Chrome 80.

June 2020: Firefox 79.

October 2020: Edge 86.

The *Two-Minute* Rule.

Cross-Site Request Forgery

Techniques for prevention.

Generate a unique token: Generate a token that validates that this is a legitimate request.

Use SameSite Cookies: Limit cookies to only working with requests that come from your domain.

Set up a CORS policy: Implement a strict Cross-Origin Resource Sharing (CORS) policy to disallow unauthorized domains.

Referer-based validation: This is typically less-than-effective.

CSRF Tokens

Being unpredictable as a defense mechanism.

A **CSRF token** is a random value generated by the server. If a request is either missing or has an invalid token, then the server will reject the request.

```
<input type="hidden"
name="csrf"
value="3964ccc5b64f54696134
3c57cf" required />
```

CSRF Tokens

Being unpredictable as a defense mechanism.

If you're using mostly AJAX requests—and *not* forms—you may choose to store the CSRF token in a `meta` tag.

```
<meta name="csrf-token"
content="3964ccc5b64f546961
343c57cf">
```

Per Session *vs.* Per Request

Request Body *vs.* Request Headers

CSRF Tokens

Some anti-patterns.

The value proposition of **CSRF tokens** is that they're *unpredictable*.

As soon as an attacker can guess the token, it's *basically worthless*.

PSA

Do *not* include your CSRF tokens in `GET` requests.

Referer-Based Validation

And, how to get around it.

See if you can get away with omitting it.

See if you can just hide the information somewhere else.

TL;DR—It doesn't hurt, but it's not enough on it's own.

Double-Signed Cookie Pattern

An alternative to CSRF tokens.

If you can't use a CSRF token, you might consider using a second cookie that is used to validate the user's session.

UX as a Security Mechanism

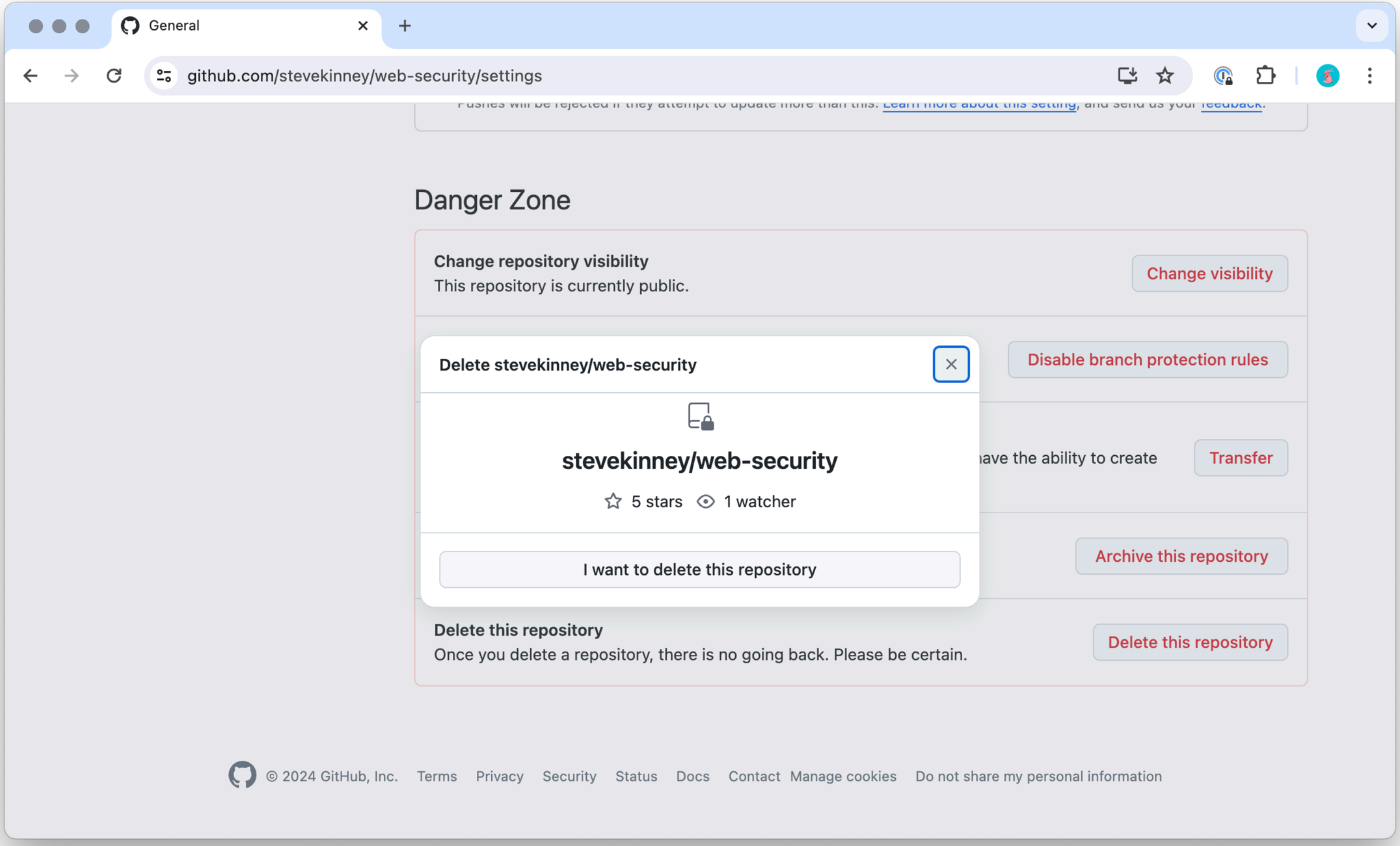
Or, making your users jump through additional hoops when it matters.

CSRF attacks are most effective when it's easy to perform the action. So, one very easy solution is to *make it harder* for users to do a thing.

Require them to **re-authenticate**.

Use **two-factor authentication** for important actions.

If you really hate your users, use a **CAPTCHA**.


General
github.com/stevekinney/web-security/settings
Danger Zone
Change repository visibility
This repository is currently public.
Change visibility
Delete stevekinney/web-security
stevekinney/web-security
5 stars
1 watcher
I want to delete this repository
Disable branch protection rules
Transfer
Archive this repository
Delete this repository
Once you delete a repository, there is no going back. Please be certain.
Delete this repository
© 2024 GitHub, Inc.
Terms
Privacy
Security
Status
Docs
Contact
Manage cookies
Do not share my personal information

General

github.com/stevekinney/web-security/settings

Pushes will be rejected if they attempt to update more than this. Learn more about this setting, and send us your feedback.

Danger Zone

Change visibility

Disable branch protection rules

have the ability to create — Transfer

Archive this repository

Delete this repository

Delete stevekinney/web-security

stevekinney/web-security

5 stars 1 watcher

Unexpected bad things will happen if you don't read this!

This will permanently delete the **stevekinney/web-security** repository, wiki, issues, comments, packages, secrets, workflow runs, and remove all collaborator associations.

I have read and understand these effects

General

github.com/stevekinney/web-security/settings

Pushes will be rejected if they attempt to update more than this. Learn more about this setting, and send us your feedback.

Danger Zone

Change repository visibility
This repository is currently public.

Change visibility

Delete stevekinney/web-security

stevekinney/web-security

5 stars 1 watcher

To confirm, type "stevekinney/web-security" in the box below

Delete this repository

Disable branch protection rules

have the ability to create

Transfer

Archive this repository

Once you delete a repository, there is no going back. Please be certain.

Delete this repository

“Oh, I only need to worry about CSRF if the user is logged in.”

—A well-meaning, if naïve software engineer.

Cross-Origin Resource Sharing

Cross-Origin Resource Sharing—or CORS, for short—is a security feature implemented by web browsers to restrict web pages from making requests to a different domain than the one that served the web page, addressing the issue of cross-origin requests.

Cross-Origin Resource Sharing

The reason for it’s existence.

CORS is designed to prevent malicious websites from accessing resources on another domain without permission.

It’s a safe way to get around the **Same Origin Policy**.

PSA

CORS is *not* a CSRF prevention mechanism.

PSA

`POST` requests from forms,
aren't covered by CORS.

Cross-Origin Resource Sharing

Simple requests aren't subject to CORS policies.

It's easier to describe what makes a request *not* simple.

Content-Type is something *other* than:

- `application/x-www-form-urlencoded`
- `multipart/form-data`
- `text/plain`

Request method is something *other* than `GET`, `POST`, or `HEAD`.

Cross-Origin Resource Sharing

The basic mechanics.

Browsers use preflight checks for complex requests (e.g., using methods like `PUT` or `DELETE`, or with custom headers) to ensure that the request is safe.

The server responds with specific HTTP headers (`Access-Control-Allow-Origin`, `Access-Control-Allow-Methods`, etc.) to specify whether the request is allowed.

Cross-Origin Resource Sharing

The headers.

Access-Control-Allow-Origin: This header specifies which domains are allowed to access the resources.

```
Access-Control-Allow-Origin: https://alloweddomain.com
```

Cross-Origin Resource Sharing

The headers.

Access-Control-Allow-Methods: Specifies the methods allowed when accessing the resource. This is used in response to a preflight request.

```
Access-Control-Allow-Methods: POST, GET, OPTIONS
```

Cross-Origin Resource Sharing

The headers.

Access-Control-Allow-Headers: Used in response to a preflight request to indicate which HTTP headers can be used when making the actual request

```
Access-Control-Allow-Headers: X-SUPER-CUSTOM-HEADER,
Content-Type
```

Cross-Origin Resource Sharing

The headers.

Access-Control-Allow-Credentials: Indicates whether or not the response to the request can be exposed when the credentials flag is true. If set to true, cookies and authorization headers are included in cross-origin requests.

```
Access-Control-Allow-Credentials: true
```

Cross-Origin Resource Sharing

Some anti-patterns.

Don't try to swap out headers that are subject to CORS (e.g. `PUT`, `PATCH`, `DELETE`) for simpler ones (e.g. `GET`, `POST`).

Cross-Origin Resource Sharing

Some anti-patterns.

Don't be fancy: Don't try to use wildcards to sub in a domain or support an array of origins. You might (read: *will*) opt out.

Only use a wildcard for your CORS policy if you *never* need to allow credentials.

Request Security Headers

Some additional headers set by the browsers.

Anything that starts with `sec-` is a header that JavaScript can't touch, which means you can assume it's safe and hasn't been tampered with.

Request Security Headers

sec-fetch-site

`cross-site`: The request initiator and the server hosting the resource have a different origin and site.

`same-site`: The request initiator have the same site—but, this *could* be a different origin.

`same-origin`: The request initiator and the server hosting the resource have the same origin.

`none`: The user did this. They entered a URL into the address bar or opened a bookmark or dragged a file into the browser window.

Request Security Headers

sec-fetch-dest

So, where's this request going?

`empty` if they used fetch().

`image` if it's an image.

`worker` if it was summoned by new Worker()

`document` for top-level navigations.

`iframe` for—umm—iframes.

Request Security Headers

Two additional headers.

`sec-fetch-user`: This a boolean (well, technically, it's only ever `true`), but it's included when a navigation request was triggered by the user.

`sec-fetch-mode`: One of the following: `cors`, `navigate`, `no-cors`, `same-origin`, `websocket`. You can take a lucky guess as to when these are set.

And now…

The Call Is Coming From Inside the House.

Cross-Site Scripting

Cross-Site Scripting (XSS) is a type of injection attack where malicious scripts are injected into otherwise benign, trusted websites.

This occurs when an attacker sends malicious code, generally in the form of a browser-side script, to a different end user.

Cross-Site Scripting

It comes in a few different flavors.

Stored: The malicious data is stored in the database or somewhere else on the backend.

Reflected: The malicious data is slid into the URL or query parameters.

DOM-based: The malicious data is input into the DOM (e.g. an input field that changes the page).

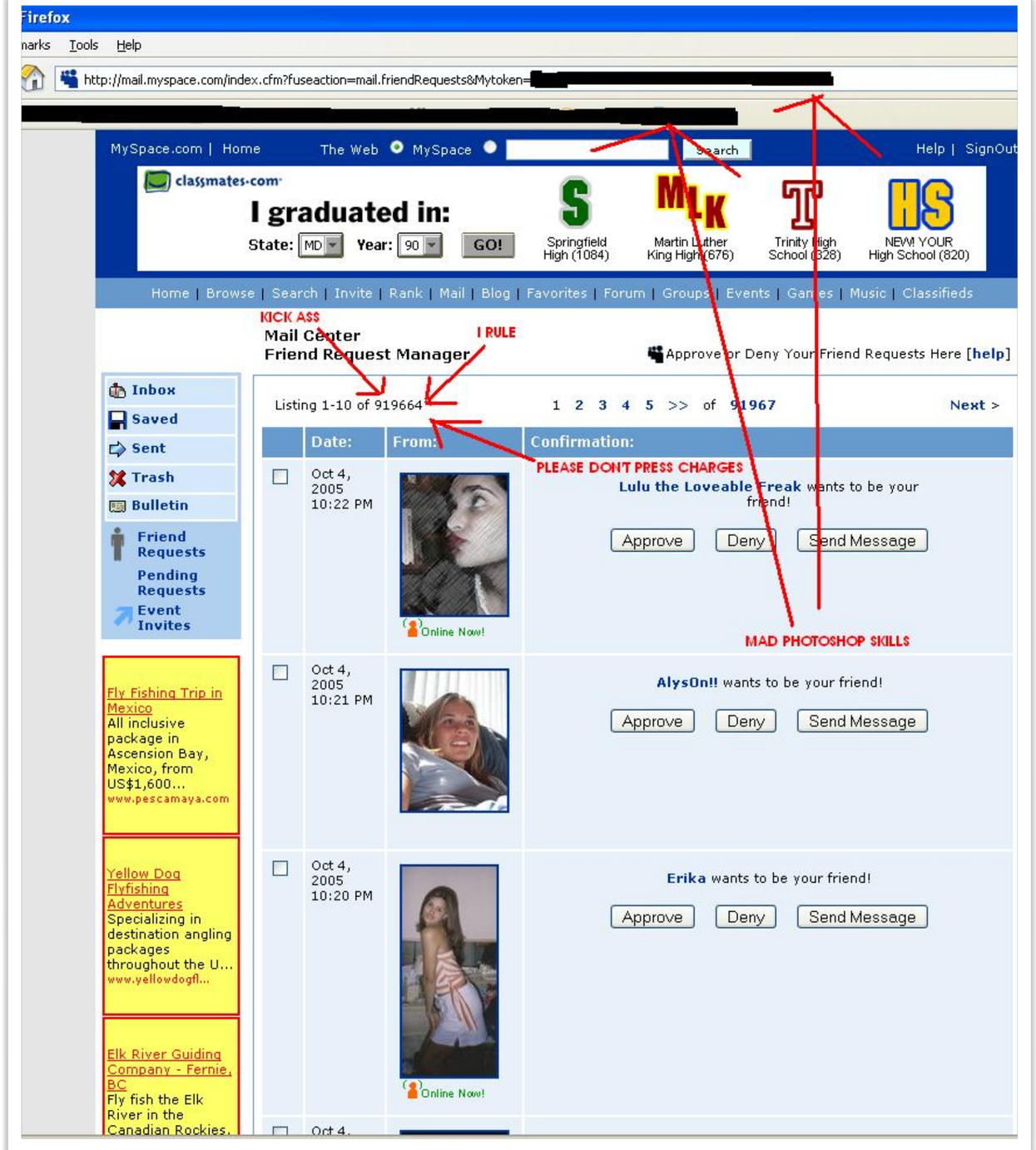
Cross-Site Scripting vs. *Cross-Site Request Forgery*

Case Study

The Samy Worm (2005)

A malicious script that injected the text "*but most of all, samy is my hero*" into a user's MySpace profile page and send Samy a friend request.

The payload would then plant itself on *that* user's profile page. Within just 20 hours, it spread to over a million users.



Source: https://samy.pl/myspace/

The Samy Worm

How it worked.

Some browsers allowed you to execute JavaScript from CSS styles.

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

The Samy Worm

How it worked.

But, in that previous example, you'd already used up single and double quotes. So, working with strings would be hard. But, you *could* hide the code in another attribute.

```
<div
  id="mycode"
  expr="alert('hah!')"
  style="background: url('javascript:eval(document.all.mycode.expr)')"
></div>
```

The Samy Worm

How it worked.

MySpace stripped out the word “javascript” from everywhere.

But, it *didn’t* strip out `java\nscript`.

And, browsers were totally cool with that new line jammed in there.

The Samy Worm

How it worked.

MySpace *also* stripped out any escaped quotes.

But, it didn't strip out ASCII character codes.

```
String.fromCharCode(34)
```

The Samy Worm

How it worked.

As you can imagine, MySpace was also not found of things like `innerHTML`. But, apparently not `eval`?

```
alert(eval('document.body.inne' + 'rHTML'));
```

From here, he had access to the full source of the page and the ability to make AJAX requests with the users cookie.

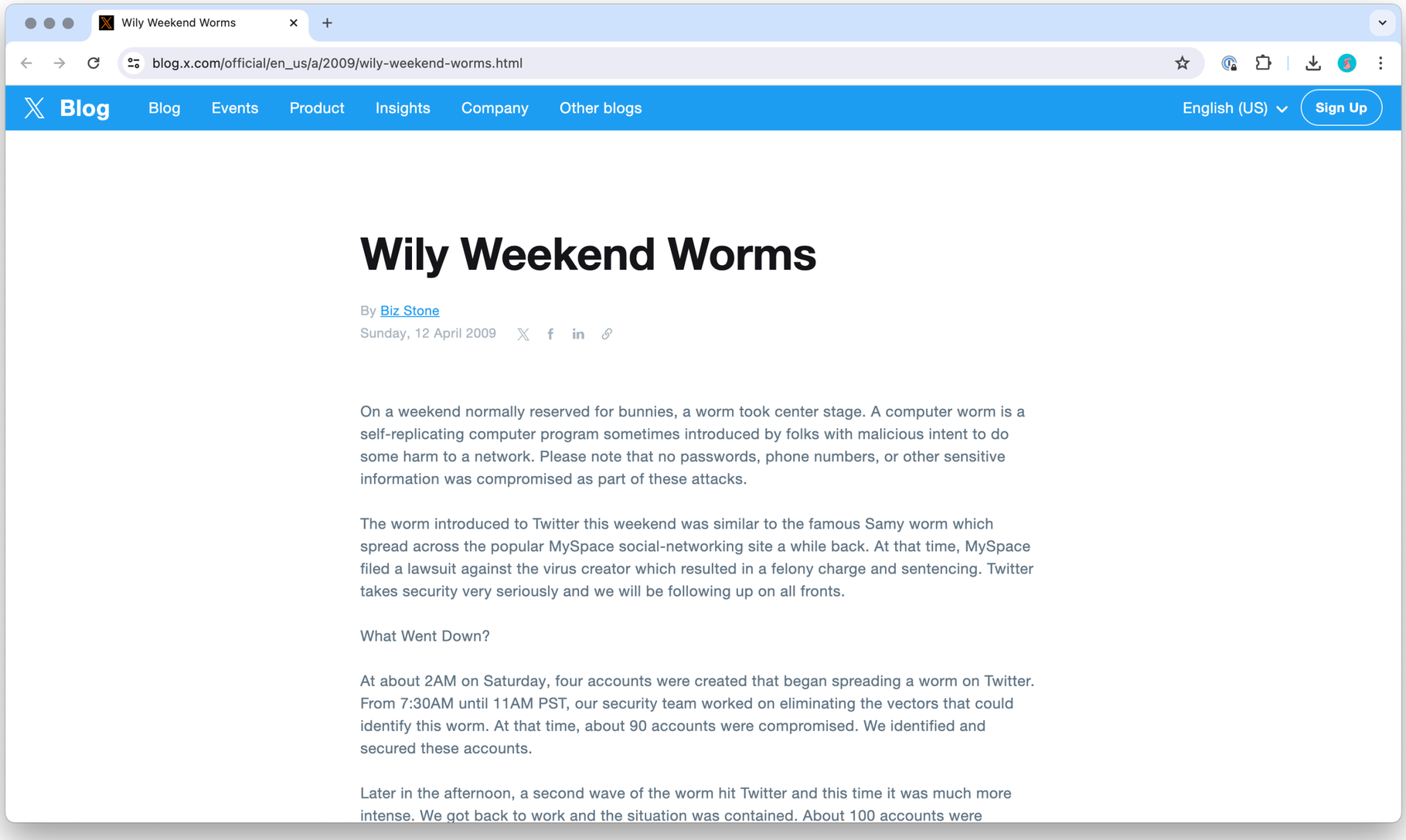
You can see the full source here: https://samy.pl/myspace/tech.html

Case Study

(Another) **Twitter Worm (2009)**

That time that someone used a stored XSS vulnerability in Twitter to create a worm that exploited the `onmouseover` event on links.

When users hovered over these links, the worm retweeted itself.



Source: https://blog.x.com/official/en_us/a/2009/wily-weekend-worms.html

The Twitter Worm

How it worked.

Let's say you tweeted something totally normal like, "http://frontendmasters.com is the best!"

You'd end up with something like this.

```
<a
  href="https://frontendmasters"
  class="tweet-url web"
  rel="nofollow">
  https://frontendmasters
</a> is the best!
```

The Twitter Worm

How it worked.

Ironically, using the @ character broke Twitter's HTML parser.

```
http://lol.no/@";onmouseover=";$
('textarea:first').val(this.innerHTML);$('.status-
update-form').submit();"class="modal-overlay"/
```

Now, if you hovered over that link, you'd end up posting the same content. *Whoops*.

Case Study

The TweetDeck Worm (2014)

A wildly simple example of how quickly an XSS vulnerability can spread.

*andy🚀 on X: "<script clas

x.com/derGeruhn/status/476764918763749376

Post

***andy🚀**
@derGeruhn

Follow

<script class="xss">$('.xss').parents().eq(1).find('a').eq(1).click();$('[data-action=retweet]').click();alert('XSS in Tweetdeck')</script> ❤️

10:36 AM · Jun 11, 2014

4.3K 59K 16K 162

Post your reply

Reply

Kent C. Dodds @kentcdodds · Jun 11, 2014

@derGeruhn @JuliaaMarieee I don't know whether you're actually retweeting this or fell victim to the vulnerability... Guessing the former...

1

https://x.com/derGeruhn

Case Study

eBay (2015–2016)

eBay had supported a `redirectTo` query parameter. The problem was that the pages you could redirect to didn't necessarily validate the query parameters in the URL that the user was redirected to—allowing for XSS attacks.

Case Study

McDonald's (2017)

We can blame a flaw in Angular's sandbox that allowed the attacker to execute code via a query parameter or we can blame the fact that McDonald's had made the interesting choice to decrypt passwords client-side.

Case Study

British Airways (2018)

There was a vulnerability in a JavaScript library called Feedify, which was used on the British Airway website.

The attack affected almost 500,000 customers of British Airways, of which almost 250,000 had their names, addresses, credit card numbers and CVV cards stolen

Case Study

Fortnite (2019)

Fortnite had a legacy page with serious vulnerabilities that went unnoticed. The vulnerability allowed an attacker to get access to the data of *all* Fortnite users.

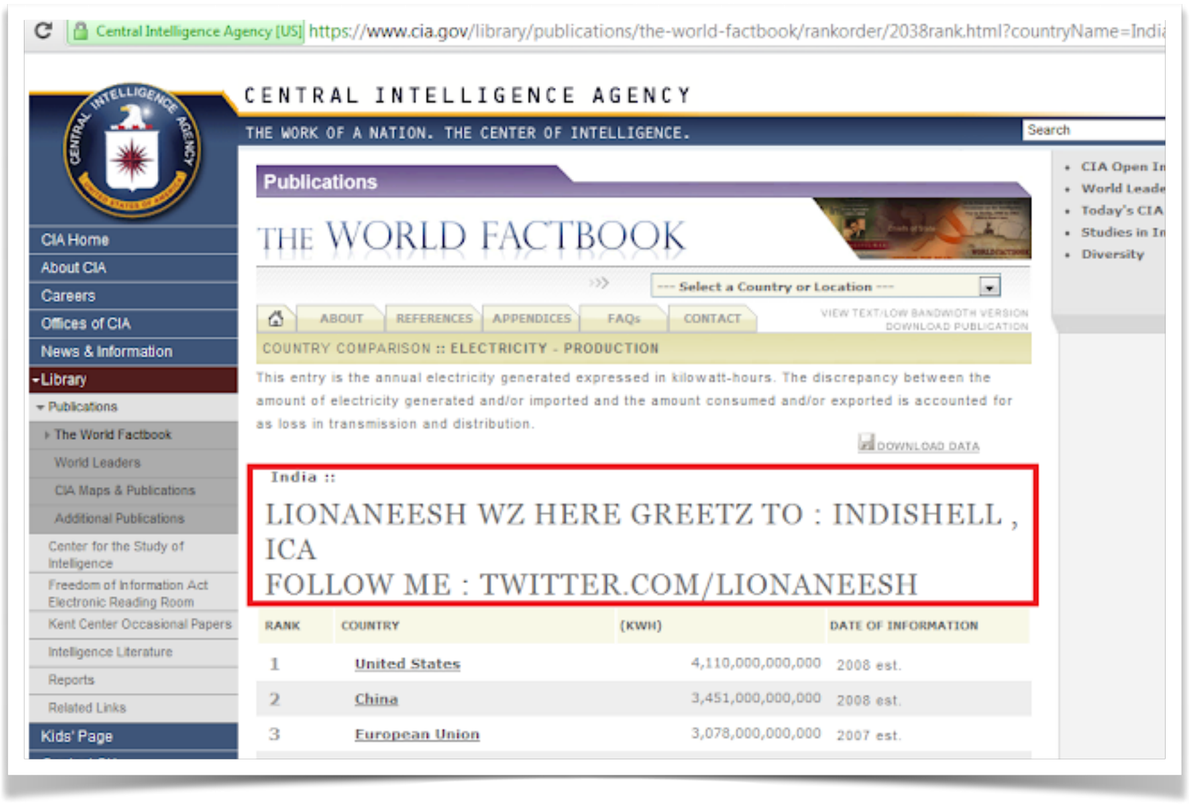
It’s unclear if the vulnerability was ever exploited.

Case Study

The Central Intelligence Agency (2011)

Yes. The **CIA**.

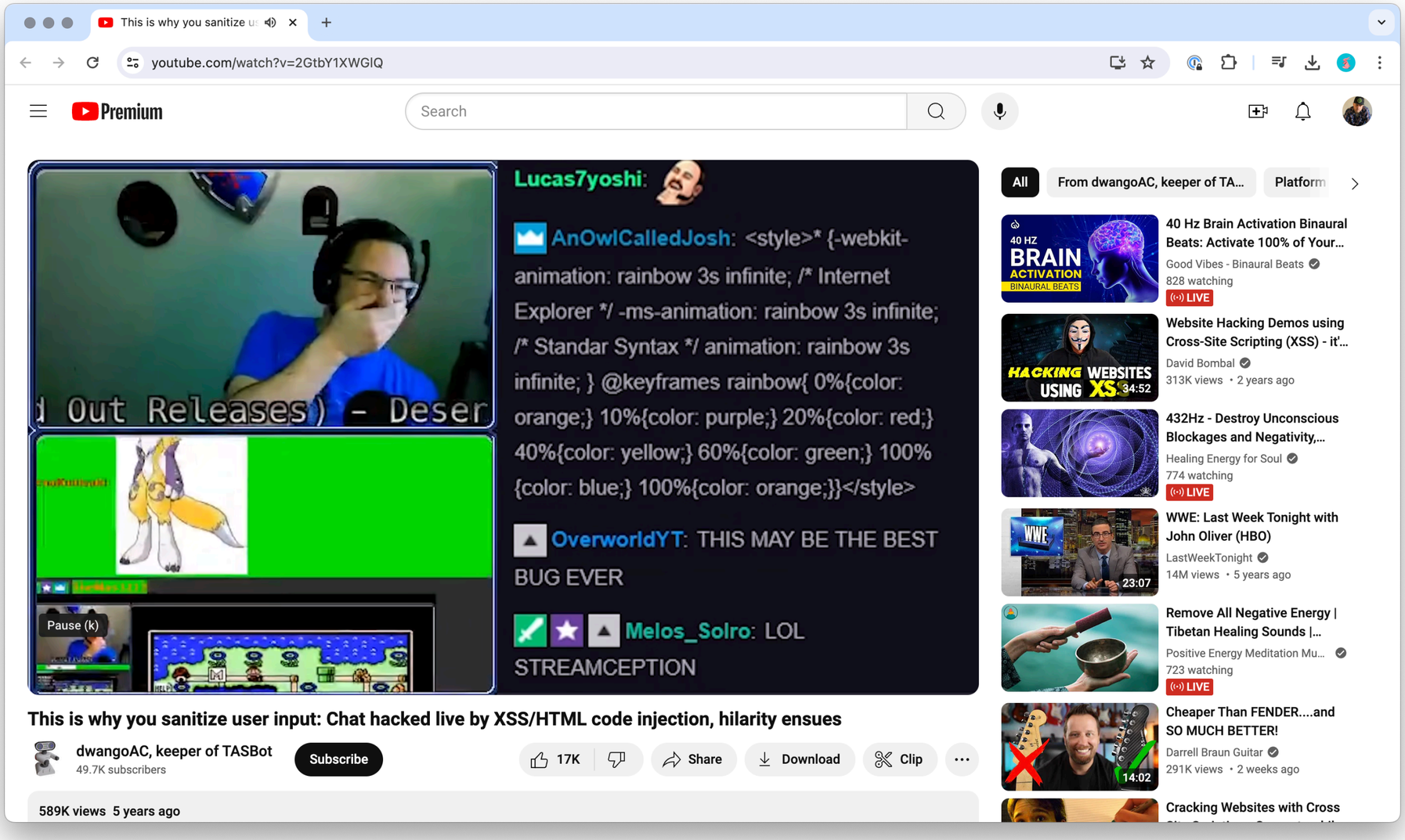
Source: https://thehackernews.com/2011/06/xss-attack-on-cia-central-itelligence.html



Case Study

GitLab (2024)

Tracked as CVE-2024-4835. This exploit took advantage of a flaw in their VS Code plugin that allowed an attacker to craft a malicious page to “exfiltrate sensitive user information.”



Source: https://www.youtube.com/watch?v=2GtbY1XWGIQ

Cross-Site Scripting

How it works.

Injection of a Malicious Script: An attacker injects malicious JavaScript (or other scripting languages) into a web application. This usually happens via user input (e.g. the comments section).

Execution of Client-Side Code: The malicious script runs in the context of the victim user's session, with the permissions of that user's browser.

Data Theft and Manipulation: Since the script executes as if it were part of the original website, it can steal cookies, session tokens, or other sensitive information.

Demonstration

Let’s play around.

Cross-Site Scripting

Best practices for mitigation.

Input Validation: Validate and sanitize all user inputs.

Output Encoding: Escape user-generated content before rendering it in the browser using context-appropriate escaping (HTML, JavaScript, URL, etc.).

Content Security Policy (CSP): Implement CSP headers to restrict sources from where scripts, styles, and other resources can be loaded.

Use Safe Methods: Avoid using functions that allow raw HTML input like `innerHTML` or `document.write`.

Libraries and Frameworks: Utilize established libraries and frameworks that auto-escape content and provide built-in protection mechanisms.

Safe Sink

A place where you can out data without the risk of it being vulnerable to XSS. For our purposes, these are places where the browser promises not to execute any code you give it.

Safe Sinks

As compared to unsafe sinks.

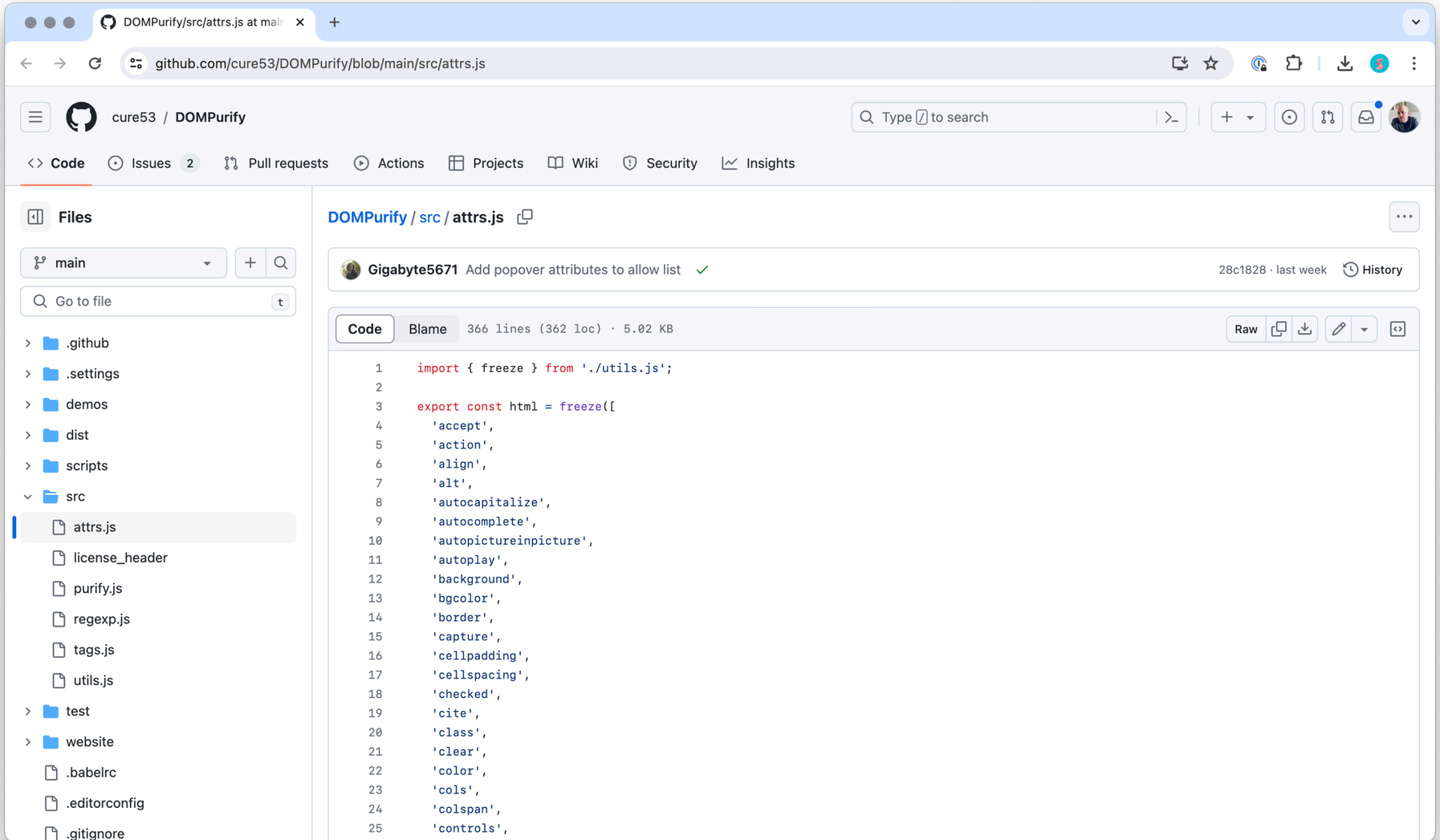
`element.textContent = randomUserInput` is a **safe sink**. The browser will not execute this code.

`element.innerHTML = randomUserInput` is *not* a safe sink. The will—*sometimes*—execute this code.

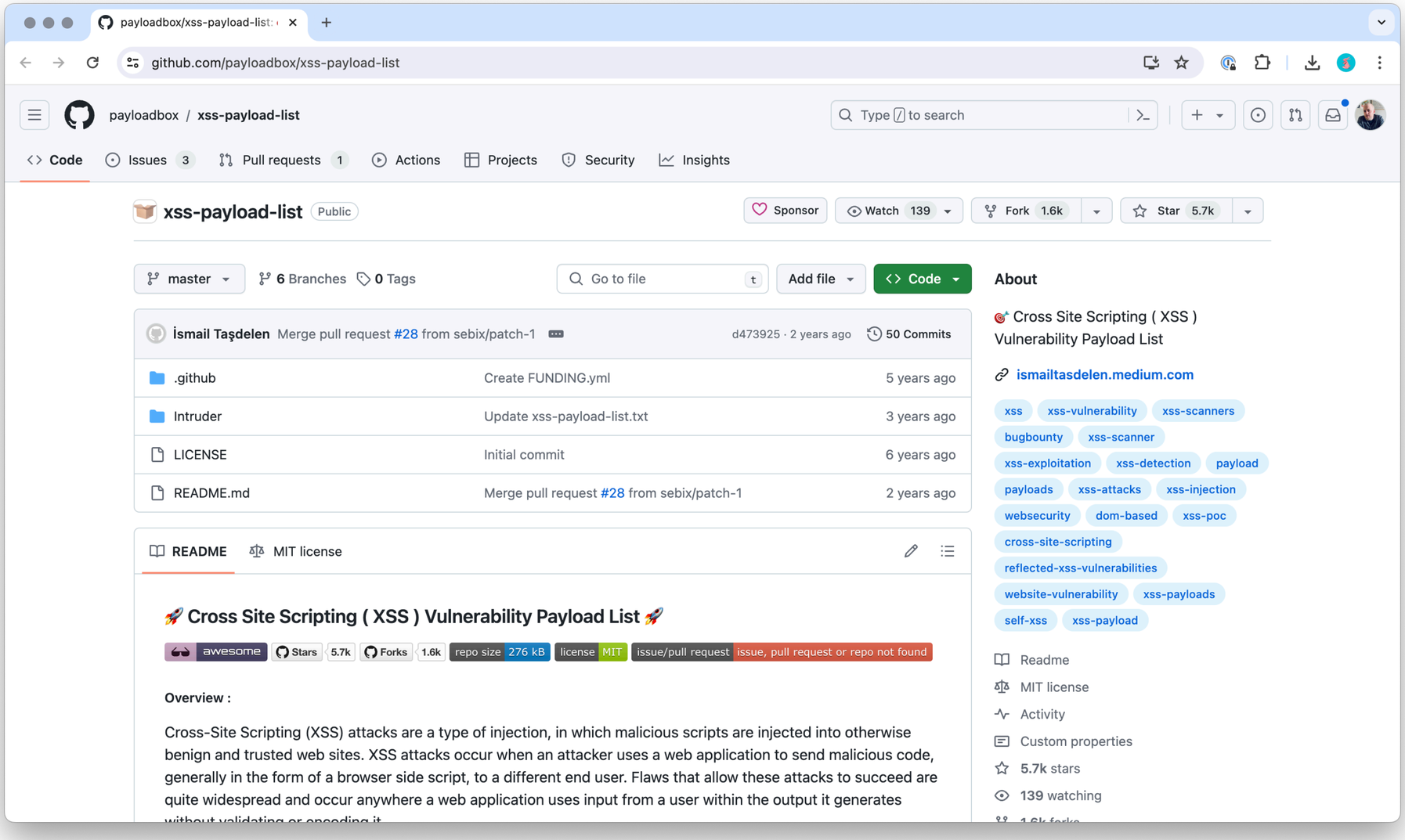
Safe Sinks

Some DOM methods are considered safe sinks.

```
element.textContent

element.insertAdjacentText

element.className

element.setAttribute

element.value

document.createTextNode

document.createElement
```



Source: https://github.com/cure53/DOMPurify/blob/main/src/attrs.js



Source: https://github.com/payloadbox/xss-payload-list

And now…

Helpful Tools That Can Be Bad if You Rely on Them Exclusively.

Content Security Policy

A security feature that helps prevent a range of attacks, including Cross-Site Scripting (XSS) and data injection attacks. CSP works by allowing web developers to control the resources the browser is allowed to load for their site.

CSP can be thought of a *second layer* defense.

Content Security Policy

How it works.

Allowlist Domains: Specify which domains are permitted to load resources such as scripts, styles, or images.

Directive-Based: Use various directives to control what content is allowed (e.g., `script-src`, `style-src`).

Report Violations: Optionally, configure CSP to report violations to a specified URI.

Content Security Policy

An example header.

```
Content-Security-Policy: script-src 'self' https://
trusted.cdn.com
```

Alternatively, you can use a `meta` tag in the head of your HTML document.

```
<meta http-equiv="Content-Security-Policy"
content="script-src 'self' https://trusted.cdn.com">
```

Demonstration

Using Helmet with Express.

Content Security Policy

In summary.

Yes, you should totally sanitize all user inputs and defensively code as best you possibly can. But, having a strong Content Security Policy gives you one extra layer of protection.

Purpose: Prevent XSS and data injection attacks.

Implementation: Define and apply CSP policies via HTTP headers or `<meta>` tags.

Allowlist: Restrict resource loading to trusted sources.

Enforcement: CSP policies can block or report unauthorized scripts.

Content Security Policy

Dealing with legacy applications.

So, what if you *can't* use a CSP because it will "*break stuff*."

The `Content-Security-Policy-Report-Only` will allow you to log CSP violations, while still allowing them to happen.

This will allow to collect data and prioritize which pages need some love.

PSA: DO NOT use `X-Content-Security-Policy` or `X-WebKit-CSP`.

Strict CSP

Really battening down the hatches.

We disallow inline styles and scripts using `unsafe-inline`.

We disallow eval and its friends using `unsafe-eval`.

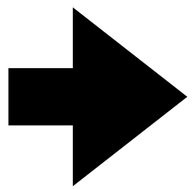
We only allow loaded resources from specific, highly-trusted resources.

We implement strict **nonce** or **hash-based** techniques to control script and style execution.

Nonce

A cute way of saying a “number used once.” It’s a token that we intend for one-time use.

By default, no sources are allowed.



```
Content-Security-Policy:
default-src ‘none';
script-src ‘nonce-rAnd0m’ ‘strict-dynamic';
base-uri ‘self';
block-all-mixed-content;
```

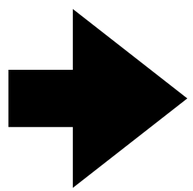
Scripts are only allowed if they match the nonce in the HTTP response header, and scripts dynamically added by these scripts are trusted.

```
Content-Security-Policy:
default-src ‘none';
script-src ‘nonce-rAnd0m’ ‘strict-dynamic';
base-uri ‘self';
block-all-mixed-content;
```

Restricts the base element, which can prevent injected scripts from changing the locations that resources are loaded from.

```
Content-Security-Policy:
default-src ‘none';
script-src ‘nonce-rAnd0m’ ‘strict-dynamic';
base-uri ‘self';
block-all-mixed-content;
```

Prevents loading any resources via HTTP on an HTTPS page.



```
Content-Security-Policy:
default-src ‘none';
script-src ‘nonce-rAnd0m’ ‘strict-dynamic';
base-uri ‘self';
block-all-mixed-content;
```

```
<script nonce="rAnd0m"></script>
```

```
<style nonce="rAnd0m"></style>
```

Using a Nonce for CSP

The pros and cons.

It's a small value, which means **your headers will be smaller** and you'll need to send fewer bytes over the wire.

You won't need to update anything if the content of your script files change.

You'll need to generate your pages programmatically. This is obviously *easier* if you're using server-generated pages.

Since the initial page has the nonce, you *cannot* **cache** the HTML.

Using a Hash for CSP

The alternative to using a nonce.

```
Content-Security-Policy: default-src ‘none'; script-src
‘sha256-/JqT3SQfawRcv/BIHPThkBvs0OEvtFFmqPF/lYI/Cxo='

<script src=“https://code.jquery.com/
jquery-3.7.1.min.js" integrity="sha256-/JqT3SQfawRcv/
BIHPThkBvs0OEvtFFmqPF/lYI/Cxo="
crossorigin="anonymous"></script>
```

Using a Hash for CSP

The ups and the downs.

You don't have to have any special infrastructure outside of generating a hash.

Depending on how many files you have, this can get *big*.

If you or anyone else *changes* the file, then it won't load since the hash is no longer valid.

That someone else could be your **CDN** or anyone in the middle trying to be helpful.

And now…

Fun With iFrames

Clickjacking

A malicious technique where an attacker tricks a user into clicking on something different from what the user perceives, effectively hijacking clicks meant for a legitimate webpage.

Clickjacking

How it works.

Embedding a Webpage: The attacker embeds the target webpage within an `iframe` on their malicious site.

Opacity & Positioning: The `iframe` is made invisible (e.g. the opacity is set to 0) or positioned behind other content.

Deceptive UI: The attacker places deceptive buttons or links on top of the invisible `iframe` elements.

User Interaction: The user *believes* they are interacting with the visible elements but are actually interacting with the concealed `iframe`.

Clickjacking

An example.

The Target: A banking site where a user logs in and clicks a button to transfer funds.

The Bad Actor: An attacker's site with an invisible `iframe` containing the embedded banking site.

The Trap: A visible button on the attacker's site saying "Click to Win a Prize!"

The Attack: Our friendly user clicks the button, which *actually* triggers the transfer funds button in the invisible iframe.

Clickjacking

Antidote: Use the `X-Frame-Options` Header.

DENY: Prevents the site from being displayed in `iframe` elements altogether.

SAMEORIGIN: Allows the site to be framed only by pages on the same origin.

```
<meta http-equiv="X-Frame-Options" content=“DENY”>
<meta http-equiv="X-Frame-Options" content=“SAMEORIGIN">

app.use((req, res, next) => {
  res.setHeader('X-Frame-Options', 'DENY');
  next();
});
```

Clickjacking

Antidote: Use a Content Security Policy.

Specify which origins are allowed to embed the content.

```
<meta http-equiv="Content-Security-Policy" content="frame-
ancestors 'self' https://trustedorigin.com”>
```

```
app.use((req, res, next) => {
  res.setHeader(
    'Content-Security-Policy',
    "frame-ancestors 'self' https://trustedorigin.com"
  );

  next();
});
```

Demonstration

Clickjacking

postMessage

`postMessage` is a method that allows for secure cross-origin communication between Window objects. It enables data exchange between a page and an `iframe` irrespective of their origins.

postMessage Vulnerability

A security issue arising when `postMessage` is misused or improperly validated. This can lead to data leakage, script injection, or cross-site scripting.

postMessage Vulnerabilities

How it works and what can go wrong.

The **Sender** uses `window.postMessage(message, targetOrigin)`.

The **Receiver** listens for messages with `window.addEventListener("message", someCallback, false)`.

Two Failure Modes: Accepting messages from any origin (e.g. `targetOrigin = '*'`) or trusting the payload (e.g. using the data without any sort of validation).

```
window.addEventListener('message', (event) => {
  if (event.origin !== 'https://place-that-i-trust.com') {
    return; // Do not process messages from untrusted origins.
  }

  const data = sanitize(event.data);
  // Do something with the data…
});
```

Data Encryption

A data encryption vulnerability occurs when sensitive data is either not encrypted or is inadequately encrypted, leading to potential data exposure.

Data Encryption

Common pitfalls.

Not encrypting: Sensitive information (like passwords, credit card numbers) is stored without encryption.

Encrypting, but poorly: Using outdated or broken encryption algorithms that can be easily decrypted.

Encrypting, but leaving the keys under the doormat: Poor handling of encryption keys, such as hardcoding keys in the code.

Encrypting, but too little and too late: Failing to encrypt data transmitted between the server and the client. *Why aren't you using HTTPS?*

Data Encryption

An ounce of prevention.

Encrypt stuff that needs encrypting: Always encrypt sensitive information using strong encryption algorithms (e.g., AES-256).

Encrypt it from end to end: Use Transport Layer Security (TLS) to encrypt data sent over the network. TL;DR—*use HTTPS*.

Hide your keys: Store encryption keys securely, not in the source code or environment variables.

Don’t roll your own crypto: Leverage well-maintained cryptographic libraries and frameworks.

PSA Don’t encrypt your passwords.

Demonstration

Hashing and salting passwords.

AWS Key Management Servic

docs.aws.amazon.com/kms/

aws | Search in this product | Contact Us | English | Create an AWS Account

AWS > Documentation > AWS KMS

AWS Key Management Service Documentation

AWS Key Management Service (AWS KMS) is an encryption and key management service scaled for the cloud. AWS KMS keys and functionality are used by other AWS services, and you can use them to protect data in your own applications that use AWS.

Developer Guide

Provides conceptual overviews of AWS Key Management Service and explains how to use it to protect data in your own applications that use AWS.

HTML | PDF

API Reference

Describes in detail the API operations for AWS Key Management Service. Also provides sample requests, responses, and errors for the supported web services protocols.

HTML | PDF

AWS KMS in the AWS CLI Reference

Describes the AWS Key Management Service commands that are available in the AWS Command Line Interface.

HTML

AWS KMS Cryptographic Details

Learn about the cryptographic operations that are run within AWS when you use AWS KMS.

HTML | PDF

Learn About AWS Cryptography Services

Cloud Key Management | Goo

cloud.google.com/security/products/security-key-management

Google Cloud | Overview | Solutions | Products | Pricing | Resources | Docs | Support | English | Console

Security | Home | Intel and expertise | SecOps | Cloud security | Security resources | Contact Us | Incident Response Assistance

Automatically provision and assign customer-managed encryption keys with Autokey.

Cloud Key Management

- Benefits
- Key features

Documentation

Use cases

- Support regulatory compliance
- Manage encryption keys via secure hardware

Cloud Key Management

Manage encryption keys on Google Cloud.

Go to console

- Deliver scalable, centralized, fast cloud key management
- Help satisfy compliance, privacy, and security needs
- Apply hardware security modules (HSMs) effortlessly to your most

VIDEO

Customer-managed encryption keys (CMEK)

34:12

e Cloud and

eys based on clear

Key Vault | Microsoft Azure

azure.microsoft.com/en-us/products/key-vault

Azure | Explore | Products | Solutions | Pricing | Partners | Resources | Search | Learn | Support | Contact Sales | Try Azure for free | Sign in

Key Vault

Safeguard cryptographic keys and other secrets used by cloud apps and services.

Try Azure for free | Create a pay-as-you-go account

Overview | Features | Security | Get started | Resources

- Increase security and control over keys and passwords
- Create and import encryption keys in minutes
- Applications have no direct access to keys
- Use FIPS 140-2 Level 2 and Level 3 validated HSMs
- Reduce latency with cloud scale and global redundancy
- Simplify and automate tasks for SSL/TLS certificates

Key Vault is now available free with an Azure free account. Start building today >

JSON Web Tokens

JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties.

For some reason, JWT is pronounced "*jot*."

JSON Web Tokens

The anatomy of a JWT.

Header: Contains metadata such as the type of token and the algorithm used for signing (`HS256, RS256`, etc.).

Payload: Contains the actual claims or data.

Signature: Ensures that the token hasn’t been tampered with.

Claims

Claims are name-value pairs. They carry information about a subject, like a user or device, securely. For instance, a claim could include a user's name or role.

You can use various data as claims, tailoring security and functionality for your application.

Claims

Types of claims.

Registered Claims: These are suggested, not mandatory, claims. They offer basic information. For example, "iss" reveals the JWT issuer.

Public Claims: Users can define these. Yet, to avoid conflicts, they should be registered with IANA or use unique URIs.

Private Claims: These are custom claims for specific uses, like sharing a user's permissions.

JSON Web Tokens

A super helpful example.


```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY
3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

JWTs Versus Session IDs

Storage of User Data

A JWT includes user data directly in the token. This data lists user ID, roles, permissions, and more. Importantly, it makes the server "stateless." Each request includes all the details needed to authenticate and authorize the user.

A session ID is a long, random string. It links to user data stored on the server. This data is often in memory or a database. When a request arrives, the server uses the session ID to find user information.

This method, however, makes the server "stateful."

JWTs Versus Session IDs

Scalability

JWTs are more scalable. They don't require server-side session storage.

This makes them perfect for distributed systems.

In these setups, users work with many servers in a cloud.

By skipping session storage, servers save resources. This also simplifies load balancing.

Scaling is harder with Session IDs. Every server that could serve a user's request must access their session data.

This often means using shared storage or replicating sessions. These methods can complicate load balancing and increase overhead.

JWTs Versus Session IDs

Security Considerations

JWTs can be stolen if stored insecurely.

They are saved in browsers, making them targets for XSS attacks.

Once issued, their information is fixed until they expire.

You can instantly revoke them by using a denylist.

Generally, session IDs are more secure because the data is stored on the server.

However, they are susceptible to session hijacking if the session ID is intercepted by an attacker.

Using `Secure` + `HttpOnly` cookies can mitigate this risk.

JWTs Versus Session IDs

Statelessness and Statefulness

JWTs create a stateless setup. Each request carries its needed information, aligning with RESTful API principles.

Session IDs need the server to keep track of states. This can complicate matters, but it offers precise control over session data.

They might be more bandwidth-efficient. Only a small ID is sent with each request.

JWTs Versus Session IDs

Expiration Management

The JWT controls its own expiration.

It's valid until it expires.

However, checks are needed to revoke it.

With session IDs, the server manages expiration.

It can quickly end or cancel a session. This method lets the user manage sessions better.

JSON Web Tokens

Algorithm confusion.

The Vulnerability: The vulnerability arises when JWT accepts weak or incorrect algorithms, notably `none` or switching from `HS256` to `RS256`.

An attacker can modify the token header to specify `alg: 'none'` and remove the signature, potentially bypassing the verification process.

Misconfiguration in the JWT library can lead to improper validation, especially when algorithms mix symmetric and asymmetric keys.

JWT Storage

Local Storage

Easy Access: Tokens can be easily accessed from JavaScript running in the browser, making it straightforward to manage tokens in client-side applications.

Persistence: Data stored in local storage persists even after the browser window is closed, facilitating persistent user sessions.

Vulnerable to XSS: If an attacker can execute JavaScript on the application, they can retrieve the JWTs stored in local storage.

No HttpOnly: Local storage does *not* support `HttpOnly` cookies, which means all stored data is accessible through client-side scripts.

JWT Storage

Session Storage

Tab Specific: Data is accessible only within the tab that created it, which provides some level of isolation.

Ease of Use: Similar to local storage, session storage is easy to use and integrates well with client-side scripts.

Limited Lifetime: Data in session storage is cleared when the tab or window is closed, which could be inconvenient for users who expect longer session times.

Just like local storage, they're **vulnerable to XSS.**

JWT Storage

Cookies

HttpOnly: Cookies can be configured as HttpOnly, making them inaccessible to JavaScript and thus protecting them from being stolen through XSS attacks.

Secure: Can be configured to be transmitted only over secure channels (HTTPS).

Domain and Path Scoping: Provides additional security settings, such as restricting the cookies to certain domains or paths.

CSRF Vulnerability: Unless properly configured with attributes like `SameSite`, cookies can be susceptible to CSRF attacks.

Size Limitations: Cookies are limited in size (around 4KB) and each HTTP request includes cookies, which could potentially increase the load times if not managed correctly.

JWT Storage

In memory (a.k.a *not* storing them)

XSS Safety: Storing JWTs in JavaScript variables can keep them out of the reach of XSS attacks, as long as the script itself is not compromised.

Fast Access: Tokens stored in memory can be accessed very quickly.

Lifetime: The token exists only as long as the page session lasts. It will be lost on page reloads or when navigating to a new page, which might not be suitable for all —honestly, *most*—applications.

JWT Best Practices

Use `HttpOnly` cookies for JWTs.

Include HttpOnly and Secure: Store JWTs in cookies with these attributes. This limits JavaScript access and ensures HTTPS-only transmission.

Add SameSite: Set the attribute to Strict or Lax to reduce CSRF risks. Lax allows some cross-site usage.

JWT Best Practices

Use short-lived JWTs and refresh tokens

Limit JWT Lifespan: Set JWTs to expire in 15 minutes to an hour. This reduces risk if a token is stolen.

Refresh Tokens: Keep sessions active with refresh tokens. Store them securely and use them to issue new access tokens.

JWT Best Practices

Do the stuff that we all know we're supposed to do.

Regular Updates: Update security systems, libraries, and frameworks to patch vulnerabilities.

SSL/TLS: Serve your site over HTTPS to secure data in transit.

CORS: Carefully set CORS to block unauthorized access.

JWT Best Practices

Control your token usage.

Avoid Sensitive Info: Don't store sensitive data in JWTs. They are easily decoded.

Define Scope and Audience: Clearly limit token usage to reduce risks.

JWT Best Practices

Monitor and audit them.

Log and Monitor Usage: Track token activities for security. Look out for unusual access patterns.

Security Audits: Regularly review your JWT handling, storage, and security.

Demonstration

Using a JWT

Strict-Transport-Security

A header that allows you to enforce that all of your traffic goes through HTTPS.

> “Be aware that inclusion in the preload list cannot really be undone. You can request to be removed, but it will take months for the deleted entry to reach users with a Chrome update and we cannot make guarantees about other browser vendors. Don't request inclusion unless you're sure that you can support HTTPS for the long term.”

—Words of wisdom Chrome’s HSTS preload site.

Fin.