O'REILLY®

Security and Frontend Performance

Breaking the Conundrum

Sonia Burney &
Sabrina Burney

Security and Frontend Performance

Breaking the Conundrum

Sabrina Burney and Sonia Burney

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Security and Frontend Performance
by Sonia Burney and Sabrina Burney

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Virginia Wilson and Brian Anderson
Production Editor: Colleen Cole
Copyeditor: Charles Roumeliotis
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

January 2017: First Edition

Revision History for the First Edition
2017-01-13: First Release

978-1-491-97215-1

[LSI]

Table of Contents

CHAPTER 1

Understanding the Problem

More often than not, performance and security are thought of as two separate issues that require two separate solutions. This is mainly due to the implications posed behind various performance and security products. We typically have either security solutions or performance solutions, but rarely solutions that offer both.

As technology has advanced, so have our attackers, finding newer and better ways to impact both the performance and security of a site. With this in mind, it has become even more critical to come up with solutions that bridge the gap between security and performance. But how do we do that?

We need to shift the focus to what we can do at the browser by leveraging various frontend techniques, such as web linking and obfuscation, versus solely relying upon the capabilities of a *content delivery network* (CDN) or the origin. We can take advantage of all the new and emerging frontend technologies to help provide a secure and optimal experience for users—all starting at the browser.

Challenges of Today: Rise of Third Parties

Many of the recent web-related concerns stem from an increase in web traffic as well as an increase in security attacks. More specifically, many of these concerns arise due to the presence of embedded third party content. Third party content is a popular topic due to the many risks involved, including both the potential for site performance degradation as well as the introduction of security vulnera-

bilities for end users. Let's discuss some of these issues in detail before diving into techniques to address them.

Web Traffic

The latest trends suggest an accelerated increase in overall web traffic; more and more users access the Web through mobile and desktop devices. With the growth in web traffic and ultimately bandwidth, end users continue to demand improved browsing experiences such as faster page loads, etc. Keeping that in mind, we not only need to adapt our sites to handle additional user traffic, but we need to do so in an optimal way to continue delivering an optimal browsing experience for the end user.

One of the higher profile frontend issues arising today is *single point of failure*. By definition, single point of failure is a situation in which a single component in a system fails, which then results in a full system failure. When translated to websites, this occurs when a single delayed resource in a page results in blocking the rest of the page from loading in a browser. Generally, blocking resources are responsible for this type of situation due to a site's dependency on executing these resources (i.e. JavaScript) before continuing to load the rest of the page. Single point of failure is more likely to occur with third party content, especially with the increase in web traffic and the obstacles in trying to deliver an optimal experience for the end user.

Attacks on the Rise

While web traffic continues to grow, security threats continue to increase as well. Many of these threats are motivated by financial means or for the purposes of harvesting data, while others execute *distributed denial of service* (DDoS) or spamming attacks to bring down origin web infrastructures.[1]

When discussing security, many different areas can be targeted during an attack, including the end user and/or the origin infrastructure. While security at the origin is important, providing a secure browsing experience for the end user is equally important and is

1 "Takeaways from the 2016 Verizon Data Breach Investigations Report", David Bisson, accessed October 13, 2016, *http://www.tripwire.com/state-of-security/security-data-protection/cyber-security/takeaways-from-the-2016-verizon-data-breach-investigations-report*.

now the focus as security threats continue to rise. Given the guarantee of a secure experience in the browser, end users are more likely to return to a site without having to worry about compromised content affecting their experiences.

As the effort to support increased web bandwidth and security threats continue, so does the need to adapt our sites to handle the increased load in an optimal and secure way for the end user.

Today, attackers are targeting vendor-related content due to the fact that proper security measures are not always in place and verified with third party content. From Alexa's Top 100 domains, pages on average fetch 48 embedded first party resources while fetching 62 embedded third party resources. Based on these numbers, we can see how heavily reliant websites are on third party content—including fonts, images, stylesheets, etc. Because of this dependency, websites are exposed to vulnerabilities like single point of failure and the potential for delivering malicious content to end users.

Technology Trends

Based on the latest issues, we need solutions to bridge the gap and address both performance concerns as well as security holes at the browser level—and some of the latest technologies do just that. Taking a look at service workers and HTTP/2, these are both technologies aimed at improving the browsing experience; however, both of these methods are restricted to use over a secure connection (HTTPS). These technologies are ideal in demonstrating how solutions can improve both performance and security for any given website.

Other frontend techniques exist to help mitigate some of the security and performance vulnerabilities at the browser. Leveraging `<iframe>`, `Content-Security-Policy`, `HTTP Strict-Transport-Security`, and preload/prefetch directives prove to help protect sites from third party vulnerabilities that may result in performance degradation or content tampering.

Start at the Browser

The main idea behind all these technology trends and frontend techniques is to help provide a secure and optimal experience for the end user. But rather than focusing on what a content delivery network, origin, or web server can do, let's shift that focus to what the browser can do. Let's start solving some of these issues at the browser.

In the remaining chapters, we will go through each of the frontend techniques and technology trends mentioned at a high level in this chapter. We will review implementation strategies and analyze how these techniques help achieve an end user's expectation of a secure yet optimal experience, starting at the browser.

CHAPTER 2

HTTP Strict-Transport-Security

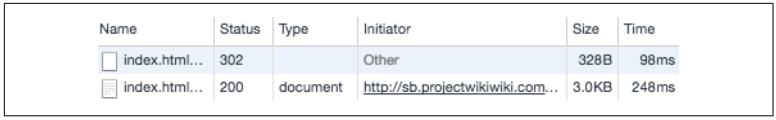
Based on recent conference talks and development in technology, we see that society is moving towards secure end user experiences. The examples of service workers and HTTP/2 working strictly over HTTPS demonstrates this movement; additionally, Google announced in late 2015 that their indexing system would now prefer indexing HTTPS URLs over HTTP URLs to motivate developers to move to HTTPS. Generally, the quick fix solution here is to enforce an HTTP to HTTPS redirect for the end user; however, this still leaves the end user open to *man-in-the-middle* (MITM) attacks in which a malicious end user can manipulate the incoming response or outgoing request over a nonsecure HTTP connection. Furthermore, the redirect method adds an additional request that further delays subsequent resources from loading in the browser. Let's explore how `HTTP Strict-Transport-Security` (HSTS), a frontend security technique, can address these issues.

What Is HSTS?

The `HTTP Strict-Transport-Security` (HSTS) header is a security technique that enforces the browser to rewrite HTTP requests into HTTPS requests, for a secure connection to the origin servers during site navigation. From HTTP Archive, 56% of base pages are using the `HTTP Strict-Transport-Security` technique and this number will continue to grow as HTTPS adoption continues to grow. Not only does this header provide browser-level security, but it also proves to be a frontend optimization technique to improve

the end user experience. By utilizing this header and the associated parameters, we can avoid the initial HTTP to HTTPS redirects so that pages load faster for the end user. As mentioned in *High Performance Websites* by Steve Souders, one of the top 14 rules in making websites faster is to reduce the number of HTTP requests. By eliminating HTTP to HTTPS redirects, we are essentially removing a browser request and loading the remaining resources sooner rather than later.

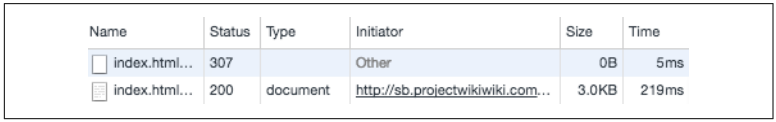
The example in Figure 2-1 demonstrates how the browser performs a redirect from HTTP to HTTPS, either using a redirect at a proxy level or at the origin infrastructure. The initial request results in a *302 Temporary Redirect* and returns location information in the headers, which directs the browser to request the same page over HTTPS. In doing so, the resulting page is delayed for the end user due to time spent and additional bytes downloaded.



Name	Status	Type	Initiator	Size	Time
index.html...	302		Other	328B	98ms
index.html...	200	document	http://sb.projectwikiwiki.com...	3.0KB	248ms

Figure 2-1. Redirect

When using the `Strict-Transport-Security` technique in Figure 2-2, we immediately see an improvement in delivery due to the elimination of the HTTP to HTTPS redirect for subsequent non-secure requests. Instead, the browser performs a *307 Internal Redirect* for subsequent requests and a secure connection is initialized with the origin infrastructure. From a frontend performance standpoint, initial header bytes are eliminated due to the removal of *302 Temporary Redirect* and the page is delivered sooner to the end user over HTTPS.



Name	Status	Type	Initiator	Size	Time
index.html...	307		Other	0B	5ms
index.html...	200	document	http://sb.projectwikiwiki.com...	3.0KB	219ms

Figure 2-2. Internal browser redirect with Strict-Transport-Security

NOTE

Keep in mind that an initial *302 Temporary Redirect* or *301 Permanent Redirect* is still needed to ensure the resulting HTTPS request returns the `Strict-Transport-Security` header; however, any subsequent requests will result in a *307 Internal Redirect* at the browser and will continue to do so until the time to live (TTL) expires, which is described in the next section.

The Parameters

In order to take advantage of the `Strict-Transport-Security` header from both a performance and security point of view at the browser, the associated parameters must be utilized. These parameters include the `max-age`, `includeSubDomains`, and `preload` directives:

```
Strict-Transport-Security:
      _max-age_=_expireTime_
      [; _includeSubDomains_]
      [; _preload_]
```

The `max-age` directive allows developers to set a *time to live* (TTL) on the browser's enforcement of a secure connection to a website through the security header. The optional `includeSubDomains` directive allows developers to specify additional pages on the same website domain to enforce a secure connection. Lastly, the optional `preload` directive allows developers to submit sites to a `Strict-Transport-Security` list maintained on a per-browser basis to ensure that secure connections are always enforced. Preload lists have various requirements including a valid browser certificate and a full HTTPS site including subdomains.

Last Thoughts

With a simple security technique, we eliminate man-in-the-middle (MITM) attacks over a nonsecure connection. While this method proves successful, the security protocol should be investigated to ensure MITM attacks can be avoided over a secure connection as well. Several SSL and TLS versions have exploitable vulnerabilities that should be considered while moving to a secure experience and deploying this security enhancement.

As a basic frontend technique, reducing the number of redirects, by default, reduces the number of browser requests, which can help to improve page load times. We are essentially moving the redirect from a proxy or origin infrastructure level to the browser for the "next" HTTP request. As with any additional headers, developers are often worried about the size of requests being downloaded in browser. The latest HTTP/2 provides header compression, which reduces the size of requests. Additionally, for nonchanging header values, HTTP/2 now maintains header state without having to resend duplicate headers during a session. Given these new benefits, we can safely utilize additional security techniques such as `Strict-Transport-Security`, without affecting overall page delivery performance. While a single HTTP header serves as a great example of bridging the gap, we will cover other techniques such as `Content-Security-Policy` to address both performance and security concerns in a similar manner.

CHAPTER 3

iFrame and Content-Security-Policy

As mentioned in Chapter 1 of this book, the latest trends include the rise in third party popularity, which exposes end users to several risks associated with this type of content. These risks mainly concern single point of failure and the potential for delivering compromised content to end users. Because third party content is not under developer control, we must utilize alternate methods to address both performance and security concerns. We can implement `<iframe>` and `Content-Security-Policy` techniques to improve the frontend browsing experience while enforcing security at the browser, specifically when embedding third party content.

Third Party Risks

Third party content is essentially untrustworthy content for obvious performance and security concerns. Let's take a look at a sample third party resource that is often used in Google Ad implementations. Generally, the resource in Example 3-1 would be included in the base page of a wesbsite.

Example 3-1. Google resource

```
<script
      async
      type="text/javascript"
      src="https://www.googletagservices.com/tag/js/gpt.js">
</script>
```

Not only does the browser load the initial resource, but the browser continues to load subsequent embedded third party resources in Figure 3-1, which can lead us to the mentioned vulnerabilities.

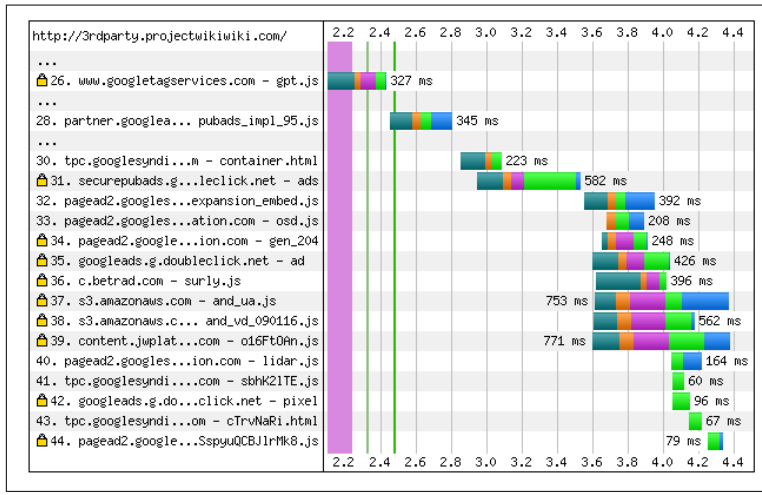


Figure 3-1. Third party waterfall

At any point, these additional resources in Figure 3-1 can fail, slow the page load, or become compromised and deliver malicious content to end users. Major headlines indicate how often both ad content and vendor content are compromised and this trend will only continue to grow as web traffic continues to grow. With the evolving nature of third party content, we as developers cannot prevent these situations from happening, but we can better adapt our sites to handle these situations when they occur.

The Basics: <script>

The `<script>` tag is the most common technique to load a script resource via HTML, for both first party content and third party content. From a performance point of view, this tag allows for the potential of single point of failure. With the latest version of HTML (HTML5), we are introduced to the script paramaters `async` and `defer`, which can help avoid single point of failure by delaying download and execution of low-prioty content such as Google Ad content. However, these parameters cannot always be used because of the lack of browser support or the fact that we can not delay the execution of the scripts for business reasons. From a security point of view, this tag provides third party providers with access to sites by executing scripts without restrictions. Overall, using the `script` tag for third party content is not the most optimal solution when it comes to trying to bridge the gap between frontend optimizations and security reenforcement at the browser level.

Improving Frontend Performance

Let's review techniques that could help address performance concerns, especially around the issue of single point of failure.

<script> Versus <iframe>

The `<iframe>` tag has been around for quite some time and developers are well aware of the pitfalls associated with using this tag. More specifically, iframes can block the onload event, block resource downloads, and they are one of the most DOM-expensive elements. But even with these pitfalls, there are ways to avoid some of these negative effects using newer methods or technologies such as HTTP/2 and dynamic iframes, which will be addressed below. From Alexa's Top 100 domains, 65% of sites use the `<iframe>` tag today so we find that developers continue to use this tag despite the downfalls. The reason for continued usage is due to the benefits associated with this tag—mainly when used with third party content.

While not commonly referred to as a frontend optimization technique, `<iframe>` provides performance benefits and, at the same time, enhances security at the browser as described briefly below. We will

dive into techniques to achieve the following points as the chapter progresses:

- Avoid delaying first party resource downloads by using a separate connection for third party content.
- Avoid single point of failure as a result of failed third party content.
- Protect end users by blocking malicious or known content from being delivered.

The `<iframe>` tag allows developers to safeguard first party content and the end user's browsing experience while loading and executing third party content in an optimal way.

While using dynamic iframes in Example 3-2, we can eliminate single point of failure issues as well as restrict access to a site using additional security enhancements discussed in the next section.

Example 3-2. Dynamic <iframe>

```
<script type="text/javascript">
      var myIframe=document.createElement("IFRAME");
      myIframe.src="about:blank";
      myIframe.addEventListener('load',
        function (e) {
           var myElement=document.createElement("SCRIPT");
           myElement.type="text/javascript";
           myElement.src="//3rdparty.com/object.js";
           myIframe.contentDocument.body.appendChild(myElement);
        }, false);
      document.body.appendChild(myIframe);
</script>
```

The JavaScript above dynamically creates an iframe that loads a third party resource, while avoiding some of the previously mentioned pitfalls of the `<iframe>` tag. Additionally, we gain the benefit of loading this third party resource on a separate connection to avoid slowing down first party resource downloads in the case of delayed content.

<script> and Content-Security-Policy

`Content-Security-Policy` (CSP) is a security enhancement which is available for use both as an HTTP header as well as a HTML

`<meta>` tag. This particular policy is used to safeguard sites from third party providers by whitelisting known or safe domains, which can help in the situation where compromised or unknown content is executing scripts on a site. The results can potentially impact the end user's browsing experience from both a performance and security point of view. From Alexa's Top 100 domains, 2% of sites are using the `Content-Security-Policy` header today, which is relatively low due to the maintenance required for using this policy. `Content-Security-Policy` usage will continue to grow in popularity due to its benefits, especially with third party or unknown content that has the potential of being compromised.

Using `Content-Security-Policy` with the `<script>` tag, we can restrict access and eliminate single point of failure from compromised content delivered through unknown third party vendors. The code samples in Examples 3-3 and 3-4 demonstrate the pairing of both techniques.

Example 3-3. <script>

```
<script
      async
      type="text/javascript"
      src="http://3rdparty.com/object.js">
</script>
```

Example 3-4. Content-Security-Policy header and tag

```
Content-Security-Policy:
      default-src 'self';
      img-src 'self' https://*.google.com;
      script-src 'self' http://3rdparty.com;
      style-src 'self' https://fonts.googleapis.com;
      font-src 'self' https://themes.googleusercontent.com;
      frame-src 'self' http://3rdparty.com

<meta http-equiv="Content-Security-Policy" content="
      default-src 'self';
      img-src 'self' https://*.google.com;
      script-src 'self' http://3rdparty.com;
      style-src 'self' https://fonts.googleapis.com;
      font-src 'self' https://themes.googleusercontent.com;
      frame-src 'self' http://3rdparty.com;">
```

As mentioned, `Content-Security-Policy` can be used in the HTML `<meta>` tag option or as an HTTP header option. The combi-

nation of `<script>` and `Content-Security-Policy` further enforces security at the browser while preventing compromised content from delaying page loads.

<script> Versus <iframe> Versus CSP

Let's compare each of the above methods with the basic `<script>` tag under the conditions of single point of failure as shown in Figures 3-2 and 3-2. In doing so, we observe that the basic `<script>` tag with a delayed third party resource can in fact delay the rest of the page from loading in the browser. In the sample Figure 3-2 filmstrip, the browser attempts to load the third party resource using `<script>` until the per-browser specified timeout kicks in.

Figure 3-2. WebPageTest visually complete comparison

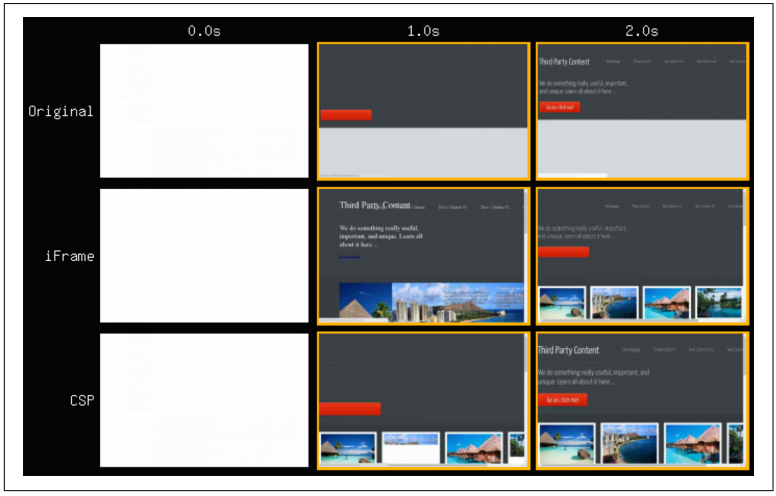


Figure 3-3. WebPageTest visually complete comparison zoom

Taking a closer look at Figure 3-3, the alternate methods result in faster page loads so that the end user can continue site usage without waiting for the delayed resource. When using the `<script>` tag in conjunction with `Content-Security-Policy`, compromised third party content will no longer result in single point of failure due to the security policies in place—a script from an unknown source

does not belong to the whitelisted set of domains and is now blocked. When replacing the `<script>` tag with the `<iframe>` tag, we observe that the browser attempts to load the delayed third party resource while continuing to load the rest of the page for the end user.

Reenforcing Security at the Browser

Let's look into how the previously mentioned techniques, `<iframe>` and `Content-Security-Policy`, can not only help eliminate certain performance issues but reenforce end user security at the browser level.

Sandboxing

By definition, the sandboxing concept involves separating individual running processes for security reasons. Within the web development world, this technique allows developers to execute third party content with additional security measures in place, separate from first party content. With that, we can restrict third party domains from gaining access to the site and end users.

As mentioned, developers are well aware of the `<iframe>` tag; however, HTML5 introduced a new `sandbox` parameter shown in Example 3-5 that provides an additional layer of security at the browser while maintaining the performance benefits associated with the `<iframe>` tag. In a similar way, `Content-Security-Policy` provides us with the same method of sandboxing third party content through use of the header or `<meta>` tag equivalent as shown in Example 3-6.

Using the `sandbox` attribute alone, we can prevent scripts and/or plugins from running, which can be useful in the situation of loading third party images for example.

Example 3-5. <iframe>

```
<iframe src="http://3rdparty.com/object.html" sandbox></iframe>
```

Example 3-6. Content-Security-Policy header and tag

```
Content-Security-Policy:
      default-src 'self';
```

```
      img-src 'self' https://*.google.com;
      style-src 'self' https://fonts.googleapis.com;
      font-src 'self' https://themes.googleusercontent.com;
      frame-src 'self' http://3rdparty.com;
      sandbox
```

```
<meta http-equiv="Content-Security-Policy" content="
      default-src 'self';
      img-src 'self' https://*.google.com;
      style-src 'self' https://fonts.googleapis.com;
      font-src 'self' https://themes.googleusercontent.com;
      frame-src 'self' http://3rdparty.com;
      sandbox">
```

We are also given flexibility in how third parties can execute content on first party websites by using a whitelist method, which allows developers to protect the end users by specifying exactly what third parties can display or control when loading content. The `sandbox` attribute has the following options:

- `allow-forms`
- `allow-modals`
- `allow-orientation-lock`
- `allow-pointer-lock`
- `allow-popups`
- `allow-popups-to-escape-sandbox`
- `allow-same-origin`
- `allow-scripts`
- `allow-top-navigation`

Third party content often comes in the form of JavaScript objects, which require certain allowances to be able to execute functionality as desired. Given that situation, we can `allow-scripts` but continue to prohibit popups or plugins from running as shown in Examples 3-7 and 3-8.

Example 3-7. <iframe>

```
<iframe
      src="http://3rdparty.com/object.html"
      sandbox="allow-scripts">
</iframe>
```

Example 3-8. Content-Security-Policy header and tag

```
Content-Security-Policy:
      default-src 'self';
      img-src 'self' https://*.google.com;
      script-src 'self' http://3rdparty.com
      style-src 'self' https://fonts.googleapis.com;
      font-src 'self' https://themes.googleusercontent.com;
      frame-src 'self' http://3rdparty.com;
      sandbox allow-scripts
```

```
<meta http-equiv="Content-Security-Policy" content="
      default-src 'self';
      img-src 'self' https://*.google.com;
      script-src 'self' http://3rdparty.com;
      style-src 'self' https://fonts.googleapis.com;
      font-src 'self' https://themes.googleusercontent.com;
      frame-src 'self' http://3rdparty.com;
      sandbox allow-scripts">
```

Sandboxing allows developers to restrict certain functionalities per third party resource embedded in a page. With `<iframe>`, we are able to avoid a scenario where single point of failure delays other resource downloads and, with the new `sandbox` directive, we are able to provide an additional layer of security at the browser.

Inline Code

When working with certain third parties such as analytics vendors, developers are often given segments of code to insert inline into the base pages of websites as shown in Example 3-9.

Example 3-9. Analytics code sample

```
<script type="text/javascript">
      (function () {
         var s = document.createElement("script"),
            el =  document.getElementsByTagName("script")[0];
         s.async = true;
         s.src = (
                  document.location.protocol ==
                  "https://3rdparty.com/analytics.js";
         el.push.parentNode.insertBefore(s, el );
       })();
</script>
```

In doing so, not only are we introducing the risk of single point of failure by allowing these resources to be downloaded without pre-

caution, but we are also providing full site access to these third parties without security measures in place. Third party content can never be guaranteed so developers must ready sites to adapt to these situations when they occur.

HTML5 introduced another parameter for the `<iframe>` tag, which is known as the `srcdoc` attribute. The `srcdoc` attribute allows developers to load inline code within the `<iframe>` tag. In doing so, the `srcdoc` and `sandbox` attributes can both be used as in Example 3-10 to prevent single point of failure and provide an additional layer of security for the end user.

Example 3-10. srcdoc and sandbox

```
<iframe
  srcdoc='<script type="text/javascript">
       (function () {
         var s = document.createElement("script"),
            el =  document.getElementsByTagName("script")[0];
         s.async = true;
         s.src = (
                  document.location.protocol ==
                  "https://3rdparty.com/analytics.js";
         el.push.parentNode.insertBefore(s, el );
       })();
    </script>'
  sandbox="allow-scripts">
</iframe>
```

Embedding inline code via the `<iframe>` tag ensures that third party content, such as analytics code shown above, will not affect the remaining resource downloads. Again, using this method, we have that ability to avoid single point of failure protect end users at the browser in case of compromised content.

Referrer Policies

End users today demand both an optimal and secure experience; security for the end user includes providing a certain level of privacy so developers must come up with ways to fulfill these expectations. The concept of referrer policies is a relatively new concept; however, it's a simple one that enforces a security measure to protect end user information. In looking at an example of navigating to a site requiring login information, end users generally enter their information and the site redirects the user to his/her account with a URL that includes information about that end user (Example 3-11).

Example 3-11. Login redirect URL

```
URL: http://profile.example.com?uname=myusername&time=1472756075
```

Generally, many sites, such as social media sites, include third party content, which is subsequently loaded with the current URL being used as a Referrer header for these third parties as shown in Example 3-12. In doing so, we are essentially leaking information about the end user and his/her session to these third parties so privacy is no longer guaranteed.

Example 3-12. Third party resource

```
URL: http://3rdparty.com/abc.js
Referrer: http://profile.example.com?uname=myusername&time=1472756075
```

Referrer policies provide developers with the ability to provide a level of privacy that end users expect. Both the `<iframe>` tag and `Content-Security-Policy` techniques include new (and experimental) parameters that help achieve these goals.

The `<iframe>` tag includes a new attribute called `referrerpolicy`, which allows developers to control how first party URLs are ingested by third party providers. Similarly, the `Content-Security-Policy` technique includes a referrer policy attribute called `referrer`, which essentially provides the same level of privacy options as provided by the iframe `referrerpolicy` option. Both of these techniques can be used with several different options depending on the use case.

`no-referrer`
: Browser will not send first party URL in Referrer.

`no-referrer-when-downgrade`
: Browser will not send first party URL in Referrer if protocol is downgraded from HTTPS to HTTP.

`origin`
: Browser will send first party origin domain in Referrer.

`origin-when-cross-origin`
: Browser will send first party origin domain in Referrer under cross origin conditions.

`unsafe-url`
: Browser will send full first party URL in Referrer.

Using the referrer policy technique with the `<iframe>` as shown in Example 3-13 provides developers with a way to avoid potential single point of failure situations while adding the benefit of privacy for the end user. Additionally, the referrer policy technique with `Content-Security-Policy` can be applied in both the HTTP header form as well as the `<meta>` tag equivalent as shown in Example 3-14. `Content-Security-Policy` also provides the same level of privacy, while protecting end users from compromised or unknown content.

Example 3-13. <iframe>

```
<iframe
      src="http://3rdparty.com/object.html"
      referrerpolicy="no-referrer">
</iframe>
```

Example 3-14. Content-Security-Policy header and tag

```
Content-Security-Policy:
      default-src 'self';
      img-src 'self' https://*.google.com;
      script-src 'self' http://3rdparty.com
      style-src 'self' https://fonts.googleapis.com;
      font-src 'self' https://themes.googleusercontent.com;
      frame-src 'self' http://3rdparty.com;
      referrer origin
```

```
<meta http-equiv="Content-Security-Policy" content="
      default-src 'self';
      img-src 'self' https://*.google.com;
```

```
script-src 'self' http://3rdparty.com;
style-src 'self' https://fonts.googleapis.com;
font-src 'self' https://themes.googleusercontent.com;
frame-src 'self' http://3rdparty.com;
referrer origin">
```

These techniques enforce a different flavor of security by focusing on the browser and protecting the end user, while still delivering third party content in an optimal way.

Because this technique is still in the experimental phase, caution should be used before implementing the referrer policy techniques through `<iframe>` and `Content-Security-Policy` due to variable browser support.

Last Thoughts

Overall, both the `<iframe>` tag and `Content-Security-Policy` techniques prove to be useful in situations that result in performance issues and/or security issues. More specifically, the newly introduced directives including `sandbox`, `srcdoc`, `referrerpolicy`, and `referrer` allow developers to improve the frontend user experience in a secure manner.

As mentioned in the beginning of this chapter, `Content-Security-Policy` is often overlooked due to the required maintenance and the added bytes to requests when downloaded in a browser. Again, with HTTP/2, we are given header compression and maintained header state, which allows more room for utilizing security techniques such as `Strict-Transport-Security` and `Content-Security-Policy`. We have other ways in which we can utilize `Content-Security-Policy` along with other frontend optimization techniques to achieve similar security and performance benefits, which will be explored in the next chapter.

CHAPTER 4

Web Linking

Web linking is a technique aimed at improving the frontend user experience by delivering prioritized resources faster to the end user through the use of the Link header or `<link>` tag. The Link technique provides the `rel` attribute with various options including `dns-prefetch`, `preconnect`, `prerender`, `prefetch`, and `preload`. While all of these techniques improve page load performance, we will focus on `prefetch` and `preload`.

Prefetch and Preload

The `prefetch` technique, as shown in Example 4-1, forces the browser to load low-priority resources that might be needed on the next page navigation. While this technique should be used with caution, we can see how the frontend user experience is improved with predetermined resource downloads and faster navigation page load.

Example 4-1. <link> tag and header

```
<link rel="prefetch" href="/dir/common.js">

Link: </dir/common.js>; rel=prefetch
```

The `preload` technique, as shown in Example 4-2, forces the browser to load high-priority resources that are needed on current page navigation. This technique should be used for resources deemed critical (required for page render and major site functional-

ity) to achieve an improved frontend user experience without degrading site performance.

Example 4-2. <link> tag and header

```
<link rel="preload" href="/dir/styles.css">

Link: </dir/styles.css>; rel=preload
```

By definition alone, these web linking techniques prove to be useful in improving overall page load from a frontend point of view.

Where Does Security Fit In?

The `Link` technique provides the `AS` attribute, which allows us to specify exactly what kind of resource the browser is loading:

```
AS
      media
      script
      style
      image
      worker
      embed
      object
      document
      font
```

The `AS` attribute provides a way to apply security policies on a per-resource type basis by using the `Link` technique in Example 4-3 along with the `Content-Security-Policy` technique in Example 4-4.

Example 4-3. <link> tag and header

```
<link rel="preload" href="/dir/styles.css" as="style">
<link rel="prefetch" href="/dir/common.js" as="script">

Link: </dir/styles.css>; rel=preload; as=style
Link: </dir/common.js>; rel=prefetch; as=script
```

Example 4-4. Content-Security-Policy header and tag

```
<meta
      http-equiv="Content-Security-Policy" content="
      default-src 'self';
      style-src 'self' https://fonts.googleapis.com;
      script-src 'self' http://3rdparty.com"
>

Content-Security-Policy:
      default-src 'self';
      style-src 'self' https://fonts.googleapis.com;
      script-src 'self' http://3rdparty.com
```

Last Thoughts

While pairing `Link` and `Content-Security-Policy` techniques, we are able to improve page delivery while applying distinct security measures to particular types of resources, such as JavaScript objects and stylesheet objects. All resources are not created equal so they should not be treated equal with a global security policy. Script type resources may require more security measures versus style type resources, and so, the `AS` attribute provides a method to associate policies on a per-resource type basis.

CHAPTER 5

Obfuscation

Obfuscation is a common frontend optimization technique that provides both security and performance benefits at the browser when applied to inline or external code. From a security point of view, developers are able to conceal implementation logic in order to avoid allowing attackers to reverse engineer code. From a performance point of view, obfuscation can reduce the size of code (page weight), implying that the browser will load and execute this content faster. Code obfuscation is a basic example of how we, as developers, can implement solutions that provide benefits in both security and performance areas. While the technique of obfuscating code has been around for quite some time, other areas of sites such as resource URLs can utilize this to further enhance frontend security and performance.

Learn from Our Attackers

To explore other applications, let's discuss how attackers deliver malicious payloads today. Origin infrastructures will often have a web application firewall implemented to block certain types of application layer attacks such as directory traversal attempts for sensitive files (Example 5-1). Attackers will then obfuscate their payloads in order to avoid detection from preset firewall rules, and in doing so, they are able to deliver malicious payloads to origin servers as intended (Example 5-2).

Example 5-1. Original URL

```
http://www.example.com/test?file=/../etc/passwd
```

Example 5-2. Obfuscated payload

```
Decimal Encoded: http://www.example.com/test?file=&#47;&#101;&#116;
&#99;&#47;&#112;&#97;&#115;&#115;&#119;&#100;

URL Encoded: http://www.example.com/test?file=%2F..%2Fetc%2Fpasswd
```

In Example 5-2, obfuscation is applied in the form of different types of encodings including decimal and URL encoded payloads. Let's use the same concept of obfuscating payloads and apply this to other areas of sites—more specifically, let's apply obfuscation to embedded third party URLs.

Alternative Application: URL Obfuscation

Why obfuscate third party URLs? Attackers will often target vendor content in order to bring down a site either through compromising content or degrading site performance. Many times, companies do not enforce security or performance measures for third party partners largely due to the focus being on first party content and protecting origin infrastructures. Developers need a way to mask the sources of these third party resources in an attempt to deter attackers from targeting vendor content in the first place.

Concept

Stepping away from the traditional application of obfuscation, we will now proxy and obfuscate third party URLs. Under normal circumstances, the browser parses a page and fetches third party resources from a third party provider as shown in Example 5-3. If we rewrite third party URLs to use a first party URL and obfuscated path/filename as shown in Example 5-4, the flow will change with the introduction of a reverse proxy. The reverse proxy has been introduced in Figure 5-1 to provide a way to interpret obfuscated requests, which can be done through use of Varnish, Apache mod_rewrite functionality, any other reverse proxies that would allow request rewrites, or simply a content delivery network. The browser will now parse a page and fetch obfuscated content, and the reverse proxy will then interpret the obfuscated request and fetch

third party content on behalf of the end user's browser. In doing so, we will achieve both enhanced security and performance at the browser.

Example 5-3. Original path

```
<img src="http://image.3rdparty.com/images/header_banner.jpg"/>
```

Example 5-4. New path

```
<img src="http://firstparty.example.com/fY4dCrWhjwjIu8Wkdhfjhg"/>
```

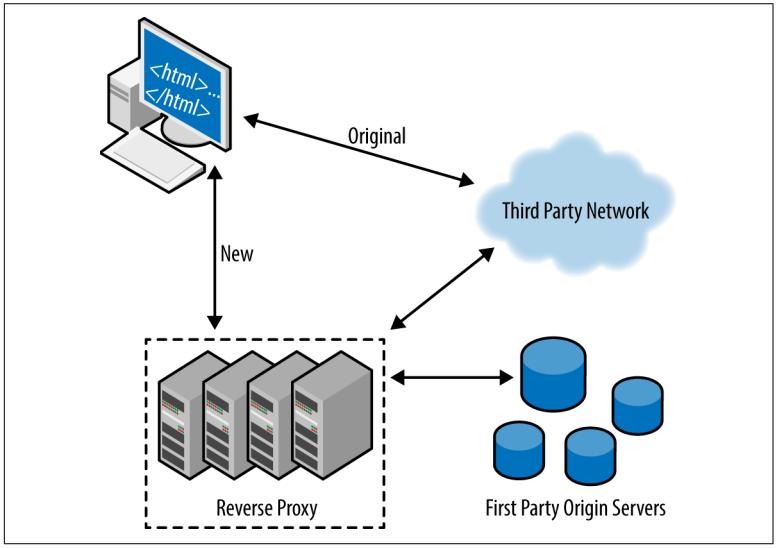


Figure 5-1. Third party workflow

URL Obfuscation Benefits

The URL obfuscation technique improves overall delivery performance and provides an additional layer of security for the end user, including benefits in areas such as privacy and resource download time.

Privacy

Because we are now proxying third party requests, we are able to enforce an additional layer of privacy by masking end users from third party providers who may not be as trusted as some of the more popular vendors. Additionally, we're concealing the source of third party content, which will deter attackers from targeting the providers and negatively impacting sites from both a security and performance point of view.

Single Point of Failure

As mentioned, third party content exposes us to the risks of single point of failure, which can happen through compromised content or simply delayed content. Using the method of proxying and obfuscating third party content provides more developer control to handle these situations, by allowing strict monitoring and disaster-recovery implementations in order to adapt to these situations when they occur. More specifically, developers can implement measures to monitor how long it takes to fetch a resource and, if the resource takes too long, serve alternate content to avoid single point of failure. Additionally, company security measures can be applied to proxied content in the same way first party content is secured so we are able to enhance security as well.

Improved Delivery Time

Let's dive into how URL obfuscation can provide various frontend performance benefits when it comes to actual resource downloads, especially with content classified as third party.

Caching

Introducing a reverse proxy into the flow provides an additional layer of caching for third party content, which ultimately brings resources closer to end users.

DNS Lookup and Connection

With the widely used HTTP/1.1, browsers open a single connection per unique hostname. Under the conditions of a page loading several resources from different third party domains, we observe in Figure 5-2 how much time is spent in `DNS Lookup` and `Connection`

in the browser due to HTTP/1.1 properties. `DNS Lookup` is the time spent to perform a domain lookup while `Connection` is the time spent when the browser initiates a connection to the resolved domain address. Both of these metrics contribute to how long a browser will take to load a resource.

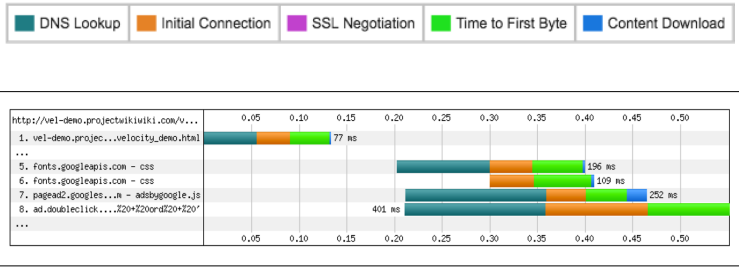


Figure 5-2. HTTP/1.1 browser conditions

If we apply URL obfuscation to all third party resources in a page, we observe in Figure 5-3 a reduction in overall `DNS Lookup` and `Connection` in the browser. Proxying these resources under the same first party hostname allows us to eliminate the browser opening many different connections.

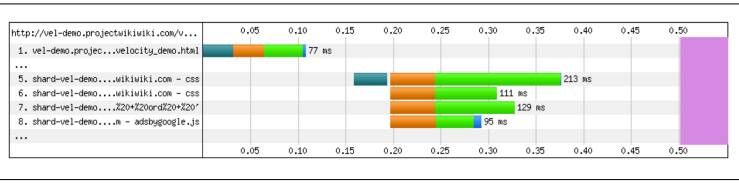


Figure 5-3. HTTP/1.1 browser conditions with URL obfuscation

With the recently introduced HTTP/2, browsers open a single connection per unique origin server. Additionally, we are introduced to the idea of multiplexing in that a browser can open a single connection to fetch multiple resources at the same time. Keeping the features of HTTP/2 in mind, we can couple HTTP/2 with URL obfuscation to provide even more of a reduction in overall `DNS Lookup` and `Connection` in the browser as shown in Figure 5-4. This is mainly due to the multiplexing feature in that a single connection is now the most optimal approach with HTTP/2.

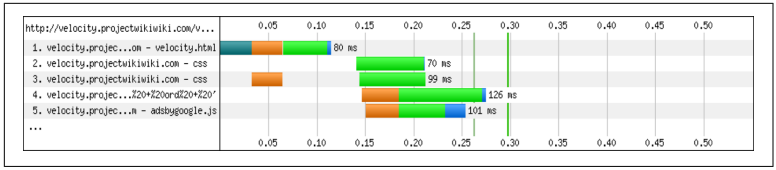


Figure 5-4. HTTP/2 browser conditions with URL obfuscation

Content-Security-Policy

Recall that `Content-Security-Policy` is a security technique aimed at whitelisting known third party domains, while prohibiting unknown third party domains from accessing and executing content on a site. When used as a header, it can grow large and often becomes harder to maintain as shown in Examples 5-5 and 5-6. Not only are we at risk of a large header delaying resource download time, but we are essentially exposing the sources of third party content as shown in the following examples. As mentioned earlier, attackers will target vendor content in order to bring a site down. That being said, we need to ensure information about vendor content is concealed as much as possible.

Example 5-5. Content-Security-Policy

```
Content-Security-Policy:
      default-src * data: blob:;script-src *.facebook.com
      *.fbcdn.net *.facebook.net *.google-analytics.com
      *.virtualearth.net *.google.com 127.0.0.1:*
      *.spotilocal.com:* 'unsafe-inline' 'unsafe-eval'
      fbstatic-a.akamaihd.net fbcdn-static-b-a.akamaihd.net
      *.atlassolutions.com blob:; style-src * 'unsafe-inline'
      data:;connect-src *.facebook.com *.fbcdn.net
      *.facebook.net *.spotilocal.com:* *.akamaihd.net
      wss://*.facebook.com:* https://fb.scanandcleanlocal.com:*
      *.atlassolutions.com attachment.fbsbx.com
      ws://localhost:* blob:
```

Example 5-6. Content-Security-Policy

```
Content-Security-Policy:
      script-src 'self' *.google.com *.google-analytics.com
      'unsafe-inline' 'unsafe-eval' *.gstatic.com
      *.googlesyndication.com *.blogger.com
      *.googleapis.com uds.googleusercontent.com
      www-onepick-opensocial.googleusercontent.com
      www-bloggervideo-opensocial.googleusercontent.com
```

```
      www-blogger-opensocial.googleusercontent.com
      *.blogspot.com; report-uri /cspreport
```

Coupling the technique of URL obfuscation with `Content-Security-Policy`, we can overcome the associated risks of large header size and content exposure as shown in Example 5-7. From a security point of view, proxying third party content masks the original source and deters attackers from targeting vendor content. From a frontend performance point of view, grouping third party content under a smaller set of first party hostnames can decrease the size of the `Content-Security-Policy` header, which is delivered on a per-resource download basis.

Example 5-7. Original and replacement

```
Content-Security-Policy:
      script-src 'self' *.google.com *.google-analytics.com
      'unsafe-inline' *.gstatic.com *.googlesyndication.com
      *.blogger.com *.googleapis.com uds.googleusercontent.com
      www-onepick-opensocial.googleusercontent.com
      www-bloggervideo-opensocial.googleusercontent.com
      www-blogger-opensocial.googleusercontent.com *.blogspot.com;
      report-uri /cspreport

Content-Security-Policy:
      script-src 'self' *.obf1.firstparty.com 'unsafe-inline'
      *.obf2.firstparty.com ; report-uri /cspreport
```

Thinking ahead with HTTP/2, we are also introduced to header compression, which avoids sending similar headers back and forth across the same connection. Instead, headers are maintained on a session basis to avoid the resending of duplicate headers. With this feature, we are also given a restriction in that header compression is provided up to 4K. For any headers that exceed this limit, the original header will continue being resent over the same connection, which can have an effect on the overall download time of resources. While using the `Content-Security-Policy` header, we are at risk of exceeding the header compression limit due to how large this header can become. With the coupling of HTTP/2 and URL obfuscation, we are essentially reducing the size of the `Content-Security-Policy` header by proxying and grouping third party resources. In doing so, developers can take advantage of HTTP/2 header compression due to the reduction in `Content-Security-Policy` header size.

Last Thoughts

Overall, we can see how proxying and obfuscating third party content provides benefits in a number of different areas. These areas include enabling faster browsing experiences, adapting to single point of failure situations, and utilizing the `Content-Security-Policy` technique without exposing sources of vendor content. With this technique, we address common concerns when dealing with third party content while improving both frontend delivery as well as browser-level security.

CHAPTER 6

Service Workers: An Introduction

Service workers have become progressively popular over the last couple of years. Last year, the talk was all about what service workers are, now it's about how we can use them. How do they help solve some of the common performance and security concerns with the modern browser? The next few chapters will discuss various applications of service workers, in an effort to bring further performance and security gains to the browser. Before diving into different use cases, let's briefly go over what service workers are and the architecture behind how they work so that we can better understand why they help bridge the gap between security and performance.

What Are Service Workers?

Textbook definition of a service worker: a script that is run by your browser in the background, apart from a webpage.[1] It's essentially a shared worker that runs in its own global context and acts as an event-driven background service while handling incoming network requests.

As displayed in Figure 6-1, a service worker lives in the browser while sitting in between the page/browser and the network, so that we can intercept incoming resource requests and perform actions

1 "Using Service Workers," Mozilla Development Network, accessed October 13, 2016, *https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers*

based on predefined criteria. Because of the infrastructure, service workers address the need to handle offline experiences or experiences with terrible network connectivity.

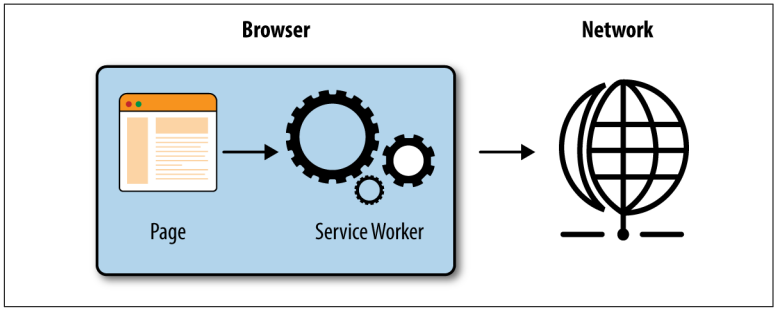


Figure 6-1. Service worker architecture

To get service workers up and running, certain event listeners need to be implemented, as displayed in Figure 6-2.

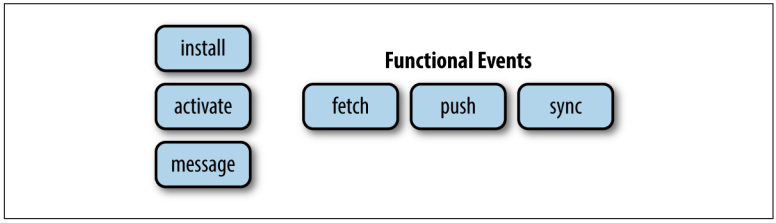


Figure 6-2. Events

These event listeners include the install and activate events, which allow developers to set up any necessary infrastructure, such as offline caches, prior to resource handling. Once a service worker has been installed and activated on the first page request, it can then begin intercepting any incoming resource requests on subsequent page requests using functional events. The available events are fetch, sync, and push; we will focus on leveraging the fetch event in the next few chapters. The fetch event hijacks incoming network requests and allows the browser to fetch content from different sources based on the request or even block content if desired.

Gotchas!

As with any new technology, there are of course several caveats.[2] Knowing these will help gain insight as to why service workers can provide both a secure and optimal experience for the end user.

- Service workers must be served over HTTPS. Given the ability to hijack incoming requests, it's important that we leverage this functionality over secure traffic to avoid man-in-the-middle attacks.
- Service workers are not yet supported in all browsers.
- The browser can terminate a service worker at any time. If the service worker is not being used or has been potentially tampered with, the browser has the ability to put it into sleep mode to avoid impacting website functionality from both a security and performance perspective.
- The service worker has no DOM access, which suggests there is less risk of code or objects being injected by an attacker.
- Fetch events are limited to a certain scope: the location where the service worker is installed.
- Fetch events do not get triggered for `<iframe>`, other service workers, or requests triggered from within service workers (helps to prevent infinite loops of event handling).

In the next few chapters, we will discuss these caveats in relation to specific service worker implementations.

2 "Service Workers: An Introduction," Matt Gaunt, accessed October 13, 2016, *http://www.html5rocks.com/en/tutorials/service-worker/introduction*

CHAPTER 7

Service Workers: Analytics Monitoring

Let's jump right into the first application of service workers, analytics monitoring.

Performance Monitoring Today

Over the last few years, third party performance monitoring tools have gained a lot of traction due to all the benefits they provide for most businesses. The reason these tools have had a significant impact on businesses is because of all the powerful data they are able to provide. More specifically, these tools are able to monitor performance by collecting timing metrics, which are then correlated to various revenue growth or even conversion business metrics. Businesses are then able to make logical decisions based on these key performance indicators (KPIs) to improve the end user's experience overall and gain that competitive edge over other businesses.

Many popular performance monitoring tools exist today, as shown in Figure 7-1; some are in-house, but the majority are third party components that are able to collect and present the data in a logical way.

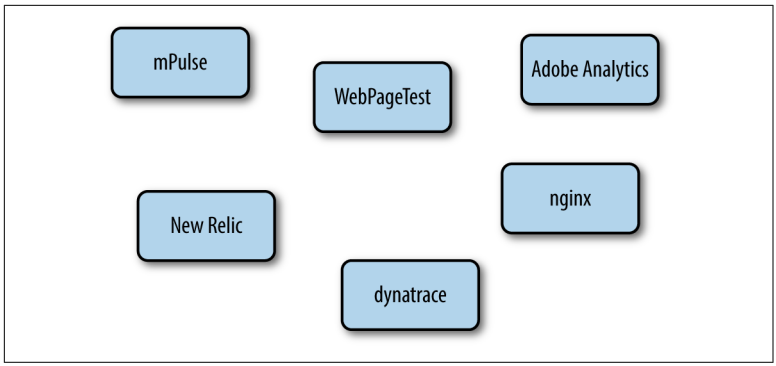


Figure 7-1. Performance monitoring tools

Each of these has unique capabilities bundled with the data and metrics they provide for measuring performance. Some measure performance over time versus others that measure single snapshots or even request transactions. And of course, some take the approach of real user monitoring (RUM) versus simulated traffic performance monitoring.

So what do third party analytics monitoring tools have to do with service workers? Typically, these tools expose their services via REST APIs. Given this approach, these tools are unable to track and provide data for offline experiences. As we advance in technology, year by year, it is important to note that we are constantly coming up with new ways to provide new types of experiences for our end users. If performance metrics have that much of an impact on the business and on the end users, then it's critical that we provide that data for offline experiences.

Track Metrics with Service Workers

The `navigator.connect` API enables third party analytics platforms to register service workers that expose APIs to track metrics, for both online and offline experiences. As long as the services are available/implemented by service workers, we can use the `navigator.connect` API to connect to third party platforms.

In Example 7-1, note that we need to define two event handlers. We need to first successfully install the service worker and connect to the third party platform via the `activate` event. Then, during each fetch event, we can log and save pertinent metrics.

Example 7-1. Connect to third party API

```
self.addEventListener('activate', function(event) {
     event.waitUntil(
         navigator.services.connect('https://thirdparty/services/
                                     analytics', {name: 'analytics'}));
});

self.addEventListener('fetch', function(event) {
        navigator.services.match({name: 'analytics'}).then(
                port.postMessage('log fetch'));
});
```

These service workers are then able to report these metrics when connectivity is reestablished so that they can be consumed by the service. Numerous implementation strategies exist for reporting the metrics; for example, we can leverage background sync so that we do not saturate the network with these requests once the user regains connectivity.

Where Do Performance and Security Fit In?

Why is leveraging service workers for third party analytics a potentially better solution than what we have today?

Many third party analytics tools that provide RUM data require injection of a particular blocking script in the `<head>` of a base page. This script cannot run asynchronously and cannot be altered in any way. The problem is that placement of a third party blocking script at the beginning of the page may delay parsing or rendering of the HTML base page content, regardless of how small or fast it is. Mpulse and Adobe Analytics are good examples of tools that require blocking JavaScript at the `<head>` of the base page. By introducing third party content earlier in the page, the site is more susceptible to single point of failure or even script injection, if that third party content is compromised or the third party domain is unresponsive. Generally, more popular third party performance tools are reliable, but there are some with security holes, or that cause performance degradation to the first party site.

Service workers remove the piece of monitoring code from the initial base page that collects and beacons out data to the third party platforms. By placing the `connect` JavaScript logic in the installation event handler, we have less blocking client-side JavaScript that can run asynchronously, which reduces the risk for single point of fail-

ure or script injection attacks. Thus there is no impact to the parsing or rendering of the HTML content.

Last Thoughts: Now Versus the Future

If service workers can help third party performance monitoring tools go "unnoticed" to the site and end user, why are they not being leveraged everywhere already? As with any new technology, coming up with a standard takes time—time to vet, time to find the holes, and time to gain popularity. Also note that third party platforms need to expose their services to service workers. Many timing APIs have yet to include this functionality, such as Google's Navigation Timing API. Given their infancy, and the reasons mentioned above, service workers still have a long way to go before becoming part of the "standard."

CHAPTER 8

Service Workers: Control Third Party Content

Now let's broaden the scope from third party analytics tools to all third party content. More specifically, let's discuss how to control the delivery of third party content.

Client Reputation Strategies

When we talk about "control" with reference to unknown third party content, what often comes to mind are backend solutions such as client reputation strategies, *web application firewalls* (WAFs), or other content delivery network/origin infrastructure changes. But with the increased usage of third party content, we need to ensure that we offer protection not only with these backend strategies, but also to our end users starting at the browser. We want to make sure requests for third party content are safe and performing according to best practices. So how do we do that? Let's leverage service workers to control the delivery of third party content based on specific criteria so that we avoid accessing content that causes site degradation or potentially injection of malicious content not intended for the end user.

Move to Service Worker Reputation Strategies

Note the simple service worker diagram in Figure 8-1. The service worker's fetch event intercepts incoming network requests for any

JavaScript resource and then performs some type of check based on a predefined list of safe third party domains, or using a predefined list of known bad third party domains. Essentially, the fetch event uses some type of list that acts as a whitelist or blacklist.

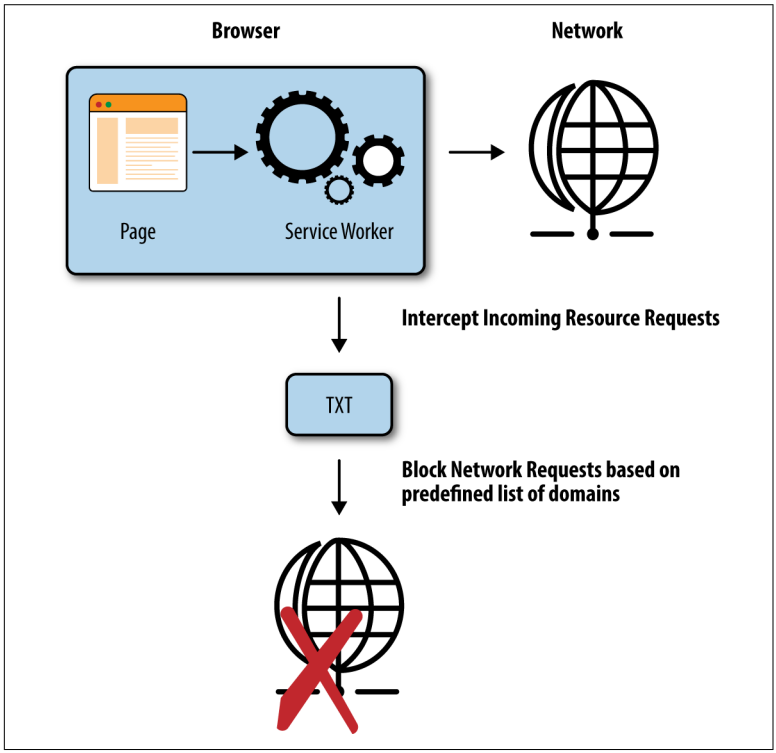


Figure 8-1. Service worker diagram

Let's take this solution and build on it to make it an adaptive reputation solution. In addition to the whitelist/blacklist, let's come up with logic that can use the forward and block mechanisms of the service worker. Specifically, let's use a counter and a threshold timeout value to keep track of how many times content from a third party domain exceeds a certain fetch time. Based on that, we can configure the service worker to block a third party request if a resource from a specific domain has exceeded the threshold X amount of times.

A Closer Look

First, we need to handle the installation and activation of the service worker so that we make the initial list of third parties accessible upon worker interception of any incoming resource requests (Example 8-1).

Example 8-1. Activate service worker

```
self.addEventListener('activate', function(event) {
  if (self.clients && clients.claim) {
    clients.claim();
  }

  var policyRequest = new Request('thirdparty_urls.txt');

  fetch(policyRequest).then(function(response) {
    return response.text().then(function(text) {
      result=text.toString();
    });
  });
});
```

During the `activate` event, we can set up a connection to some list of third party domains. The text based file could live at the origin, at a content delivery network, at a remote database, or at other places that are accessible to the service worker.

Now, as shown in the pseudocode in Example 8-2, when a resource triggers a fetch event, limited to JavaScript only in this example, we can configure the service worker to block or allow the resource request based on two conditions: the whitelist/blacklist of third party domains and the counter/threshold adaptive strategy.

Example 8-2. Fetch event handler pseudocode

```
self.addEventListener('fetch', function(event) {

  // only control delivery of JavaScript content
  if (isJavaScript) {
    // determine whether or not the third party
    // domain is acceptable via a whitelist
    isWhitelisted = match(resource,thirdpartyfile)
    if (isWhitelisted) {
      getCounter(event, rspFcn);
      var rspFcn =  function (event){
        if (flag > 0){
```

```
          // if we have exceeded the counter
          // block the request OR serve from an offline cache
        }
       }
     }
    else{
     // send the request forward
     // if the resource has exceeded a fetch time
     if (thresholdTimeoutExceeded) {
       updateCounter(event.request.url);
     } // else do nothing

     // add resource to offline cache
     cache.add(event.request.url);
    }
   }
   else {
    event.respondWith(fetch(event.request));
   }
});
```

Analysis

Note the following method in particular:

```
getCounter(event, rspFcn);
```

This method fetches the current state of the counter for a third party domain. Remember that, for each fetch event, we can gather a fetch time for each resource. But the counter needs to be maintained globally, across several fetch events, which means we need to be able to beacon this data out to some type of data store so that we can fetch and retrieve it at a later time. The implementation details behind this method have not been included but there are several strategies. For the purposes of the example, we were able to leverage Akamai's content delivery network capabilities to maintain count values for various third party domains.

Upon retrieving the counter value, we have a decision to make as seen in the implementation:

```
updateCounter(event.request.url);
```

- If we have exceeded the number of times the third party content hit the threshold timeout for fetch time, as indicated by the counter value, then we can either block the request from going forward OR we can serve alternate content from an offline

cache. (An offline cache needs to be set up during the installation event of a service worker.)

- If we have NOT exceeded the counter, then we send the request forward and record whether or not it has exceeded the fetch time on this run, in this case, if it has exceeded 500 milliseconds. If the resource hit our predefined threshold value, then we can update the counter using the `updateCounter` method.

Again, the implementation details for this method have not been included, but you will need to be able to beacon out to a data store to increment this counter. If the resource did not hit the threshold value, then there is no need to update the counter. In both cases, we can store the third party content in the offline cache so that if the next time a fetch event gets triggered for the same resource, we have the option to serve that content from the cache.

Sample code

Example 8-3, shows a more complete example for the pseudocode in Example 8-2.

Example 8-3. Fetch event handler

```
self.addEventListener('fetch', function(event) {
  // Only fetch JavaScript files for now
  var urlString = event.request.url;

  if(isJavaScript(urlString) && isWhitelisted(urlString)) {
    getCounter(event, rspFcn);
    var rspFcn =  function (event){
      if (flag > 0){
        // If counter exceeded, retrieve from cache or serve 408
        caches.open('sabrina_cache').then(function(cache) {
          var cachedResponse =
              cache.match(event.request.url).then(function(response)
              {
              if(response) {
                console.log("Found response in cache");
                return response;
              } else{
                console.log("Did not find response in cache");
                return (new Response('', {status: 408,
                        statusText: 'Request timed out.'}));
              }
          }).catch(function() {
            return (new Response('', {status: 408,
```

```
                statusText: 'Request timed out due to error.'}));
      });
      event.respondWith(cachedResponse);
    });
  } else{
    Promise.race([timeout(500), fetch(event.request.url,
      {mode: 'no-cors'})]).then(function(value){
      if(value=="timeout"){
        console.log("Timeout threshold hit, update counter");
        updateCounter(event.request.url); // use promises here
      } else console.log("Timeout threshold not reached,
            retrieve request, w/o updating counter");
      // If counter not exceeded (normal request)
      // then add to cache
      caches.open('sabrina_cache').then(function(cache) {
        console.log("Adding to cache");
        cache.add(event.request.url);
      }).catch(function(error) {
        console.error("Error" + error);
        throw error;
      });
    });
  }};} else { event.respondWith(fetch(event.request)); }});
```

Last Thoughts

There are numerous ways to implement the `getCounter` and `updateCounter` methods so long as there exists the capability to beacon out to some sort of data store. Also, Example 8-3 can be expanded to count the number of times a resource request has exceeded other metrics that are available for measurement (not just the fetch time).

In Example 8-3, we took extra precautions to ensure that third parties do not degrade performance, and do not result in a single point of failure. By leveraging service workers, we make use of their asynchronous nature, so there is a decreased likelihood of any impact to the user experience or the DOM.

NOTE

Just like JavaScript, service workers can be disabled and are only supported on certain browsers. It is important to maintain fallback strategies for your code to avoid issues with site functionality.

The main idea behind this implementation is to avoid any unnecessary performance degradation or potential script injection by only allowing reputable third party content that meets the criteria that we

set in place. We are essentially moving security and performance to the browser, rather than relying solely on backend client reputation, WAFs, and other content delivery network/origin infrastructure solutions.

CHAPTER 9

Service Workers: Other Applications

Using service workers to control the delivery of third party content or even monitor third party performance is critical. But what about first party resources? Or frontend techniques to improve the performance and security with base page content in general? Service workers can be leveraged in many different ways, including through input validation and geo content control, which are discussed briefly below.

Input Validation

Input validation strategies typically involve client-side JavaScript, server-side logic, or other content delivery network/origin logic in an effort to not only prevent incorrect inputs or entries, but also to prevent malicious content from being injected that could potentially impact a site overall. The problem with some of the above strategies is that a site still remains vulnerable to attacks.

With client-side JavaScript, anyone can look to see what input validation strategies are in place and find a way to work around them for different attacks such as SQL injections, which could impact the end user's experience. With server-side logic or other content delivery network/origin features, the request has to go to the network before being validated, which could impact performance for the end user.

How can service workers mitigate some of these vulnerabilities? Let's use the service worker fetch handler to validate the input field and determine whether to forward or block a resource request. Of course service workers can be disabled, as with JavaScript, but it is up to the developer to put backup sever-side strategies in place as a preventative measure.

Benefits of using service workers:

- Remove the need to have the request go to the network, server, content delivery network, or origin, which removes additional validation delay.
- Reduce the risk of those requests being intercepted if we block them before even forwarding to the network.
- Service workers have no DOM access so malicious content is likely not going to be injected to change the page and how it validates form fields.

The below pseudocode in Example 9-1 helps demonstrate how to implement input validation via a service worker. The fetch event can catch the POST request and analyze the fields before submission.

Example 9-1. Fetch event handler for form submission

```
self.onfetch = function(event) {
    event.respondWith(
        // get POST request from form submission
        // analyze fields before submitting to network
        if input field is valid
                submit fetch(event.request)
        else block
    );
};
```

Geo Content Control

Delivering content to end users based on their specific geography has been critical to business growth. Advanced technology available at the content delivery network or origin has allowed businesses to target end users with content based on their geo locations. But what if we could make that determination at the browser, and then forward the request based on an end user's geo location? Service workers can help by leveraging the GeoFencing API, which allows web

applications to create geographic boundaries around specific locations.

Push notifications can then be leveraged when a user or device enters those areas. But being a browser-specific technique, there is the security concern in spoofing a geo location. With this in mind, it is critical to maintain server-side logic for comparison purposes, whether it exists at the origin or content delivery network, to ensure that geo location data is not tampered with.

This functionality is still relatively new because of the different caveats when accessing an end user's geo location. But the idea of moving this functionality to the browser, with possible access to an offline cache for geo-specific content, can help eliminate the need to make a decision at the content delivery network or origin, which could help improve performance.

A Closer Look

Let's take a look at Example 9-2. During service worker registration, different GeoFence regions would be added, along with any additional offline caches for content.

Example 9-2. Register event handler: Adding GeoFences

```
navigator.serviceWorker.register('serviceworker.js')
    .then((swRegistration) => {
    let region = new CircularGeofenceRegion({
        name: 'myfence',
        latitude: 37.421999,
        longitude: -122.084015,
        radius: 1000
    });
    let options = {
        includePosition: true
    };
swRegistration.geofencing.add(region, options).then(
    // log registration
);
// setup offline cache for geo-specific content
});
```

Once the service worker is active, it can start listening for users or devices entering or leaving the GeoFence we set up during registration (Example 9-3).

Example 9-3. GeoFence enter event listener

```
self.ongeofenceenter = (event) => {
    console.log(event.geofence.region.name);
    //if offline cache has region resources -> serve content
};
```

Because service workers can be disabled, having backend solutions in place is critical. As an additional preventative measure, content delivery networks or the origin can validate geo location and geo-specific content being served back to the end user.

Last Thoughts

Input validation and geo content control are just a couple more service worker applications, but the use cases and applications will continue to increase as we advance with this technology. The idea is to take backend solutions and bring them to the browser in an effort to mitigate some of the common security and performance issues we see today.

CHAPTER 10
Summary

Throughout this book, we have learned that a performance solution can be a security solution and a security solution can, in fact, be a performance solution. In the past and up until now, the majority of the focus has been on improving conditions at the origin, by looking at web infrastructure. Additionally, certain performance improvement techniques have been found to compromise security and vice versa, certain security techniques have been found to compromise performance. This is mainly due to business needs. End users demand an optimal browsing experience and they will continue to demand even faster and more secure browsing experiences. That being said, we need to develop solutions that help bridge the gap between security and performance, by bringing the focus to the browser.

We have discussed major trends and prominent issues, including the concept of single point of failure as well as the possibility of delivering compromised content to end users. As developers, it is important to recognize when these situations can occur so that we can better adapt our sites to handle unexpected behavior.

Much of the focus of this book has been on third party content due to the fact that third party providers are becoming increasingly popular as they are able to offload much of the work from companies' origin web infrastructures. End users are exposed to the many different risks mentioned throughout this book due to this, so we can see how the concept of bridging the gap at the browser is becoming increasingly important.

What Did We Learn?

Over the course of this book, we have explored several existing techniques as well as newer technologies to help achieve an optimal frontend experience that is also secure. Keep these simple yet powerful points in mind:

- *Avoid the HTTP→HTTPS redirect on every page request!*
  - Use the HTTP `Strict-Transport-Security` technique to cache these redirects and potentially configure browser preload lists to continue enforcing an initial secure connection.
- *Protect your sites from third party vulnerabilities*
  - `Sandbox`, `sandbox`, `sandbox`….and `srcdoc`! Utilize the new `<iframe>` directives introduced in HTML5 and corresponding `Content-Security-Policy` directives to better address third party concerns.
  - Explore the latest on referrer policies. While still experimental, adopting these practices in your sites will better ensure privacy for your end users.
- *Improve content delivery in a secure way*
  - Consider pairing `preload` and `prefetch` web linking techniques with `Content-Security-Policy` to gain a security enhancement in addition to a frontend optimization technique.
  - Deter attackers that target your vendor content by obfuscating the sources in an optimal way.
- *Explore service workers!*
  - While still considered *new*, explore the latest with service workers as they can be powerful especially when bringing security and performance enhancements to the browser.
  - Service workers provide more control including geo content control and input validation methods, as well as monitoring third party content (analytics code, ad content, etc.).

Last Thoughts

Remember to enhance techniques that exist today using the methods described throughout this book. Additionally, stay up-to-date with the latest technologies and look for newer ways to bring a secure and optimal experience to the end user.

While security can be a vague term, there are many different areas that are often dismissed. Origin web security is usually the focus, but it is important to consider the different flavors of security including privacy for end users, as well as the ability to conceal information from potentially malicious end users.

Compromising security for a performance solution and vice versa is no longer an option given the latest trends. Let's continue thinking about solutions that provide benefits in both areas as the need continues to increase.

About the Authors

Sonia Burney has a background in software development and has been able to successfully participate in many roles throughout her years at Santa Clara University and in the tech world. Every role, at every company, has driven her to learn more about the tech industry, specifically with regards to web experience and development. While Sonia's background consists of mostly software development roles within innovative teams/companies, her current role at Akamai Technologies now includes consulting and discovering new solutions to challenging problems in web experience—specifically, coming up with algorithms designed to improve the frontend experience at the browser. Outside of work, not only is she a dedicated foodie, but she enjoys traveling, running, and spending time with friends and family.

Sabrina Burney has worked in many different fields since graduating from Santa Clara University. She has a background in computer engineering and has always had a passion for technologies in the IT world. This passion stems from learning about newer tech being developed as well as enhancing tech that is already present and underutilized. While Sabrina currently works at Akamai Technologies, her experience inside and outside of Akamai includes roles in software development and web security, as well as more recently the web experience world. She is able to utilize her backgrounds in multiple fields to help improve the overall end user experience when it comes to navigating the Web. Sabrina's recent work is focused on third-party content and ways to improve the associated vulnerabilities and concerns—she has several patents pending in this subject area. Outside of work, she enjoys playing soccer with her fellow coworkers as well as traveling with her family.