Play safely in sandboxed IFrames

Jun 4, 2010

Peter Weber
Twitter Mastodon

On this page

Constructing a rich experience on today's web almost unavoidably involves embedding components and content over which you have no real control. Third-party widgets can drive engagement and play a critical role in the overall user experience, and user-generated content is sometimes even more important than a site's native content. Advertising from either isn't really an option, but both increase the risk that something bad[1] could happen on your site. Each widget that you embed — every ad, every social media widget — is a potential attack vector for those with malicious intent.

Content Security Policy (CSP) can mitigate the risks associated with both of these types of content by giving you the ability to whitelist specifically trusted sources of script and other content. This is a major step in the right direction, but it's worth noting that the protection that most CSP directives offer is binary: the resource is allowed, or it isn't. There are times when it would be useful to say "I'm not sure I actually trust this source of content, but it's soooo pretty. Embed it please. Browser, but don't let it break my site."

Least Privilege

In essence, we're looking for a mechanism that will allow us to grant content we embed only the minimum level of capability necessary to do its job. If a widget doesn't need to pop up a new window, taking away access to window.open can't hurt. If it doesn't require Flash, turning off plugin support shouldn't be a problem. We're as secure as we can be if we follow the principle of least privilege, and block each and every feature that isn't directly relevant to functionality we'd like to use. The result is that we no longer have to blindly trust that some piece of embedded content won't take advantage of privileges it shouldn't be using. It simply won't have access to the functionality in the first place.

`iframe` elements are first step toward a good framework for such a solution. Loading some untrusted component in an `iframe` provides a measure of separation between your application and the content you'd like to load. The framed content won't have access to your page's DOM, or data you've stored locally, nor will it be able to draw its arbitrary positions on the page. It's limited in scope to the frame's outline. The separation isn't truly robust, however. The contained page still has a number of options for annoying or malicious behavior: autoplaying videos, plugins, and popups are the tip of the iceberg.

The `sandbox` attribute of the `iframe` element gives us just what we need to tighten the restrictions on framed content. We can instruct the browser to load a specific frame's content in a low-privilege environment, allowing only the subset of capabilities necessary to do whatever work needs doing.

Twust, but verify

Twitter's "Tweet" button is a great example of functionality that can be more safely embedded on your site via a sandbox. Twitter allows you to embed the button via an iframe:

```
<iframe src="http://platform.twitter.com/widgets/tweet_button.html"
        style="border: 0; width:130px; height:20px;"></iframe>
```

To figure out what we can lock down, let's carefully examine what capabilities the button requires. The HTML that's loaded into the frame executes a bit of JavaScript from Twitter's servers, and generates a popup populated with a tweeting interface when clicked. That interface needs access to Twitter's cookies in order to be the tweet to the correct account, and needs the ability to submit the tweeting form. That's pretty much it: the frame doesn't need to load any plugins, it doesn't need to navigate the top level window, or any of a number of other bits of functionality. Since it doesn't need those privileges, let's remove them by sandboxing the frame's content.

Sandboxing works on the basis of a whitelist. We begin by removing all permissions possible, and then turn individual capabilities back on by adding specific flags to the sandbox configuration. For the Twitter widget, we've decided to enable JavaScript, popups, form submission, and twitter.com's cookies. We can do so by adding `sandbox` attributes to the `iframe`, with the following value:

```
<iframe sandbox="allow-same-origin allow-scripts allow-popups allow-forms"
        src="https://platform.twitter.com/widgets/tweet_button.html"
        style="border: 0; width:130px; height:20px;"></iframe>
```

That's it. We've given the frame all the capabilities it requires, and the browser will helpfully deny it access to any of the privileges that we didn't explicitly grant it via the `sandbox` attribute's value.

Granular Control over Capabilities

We saw a few of the possible sandboxing flags in the example above, let's now dig through the inner workings of the attribute in a little more detail.

Given an iframe with an empty sandbox attribute, the framed document will be fully sandboxed, subjecting it to the following restrictions:

- JavaScript will not execute in the framed document. This not only includes JavaScript explicitly loaded via script tags, but also inline event handlers and javascript: URLs. This also means that content contained in noscript tags will be displayed, exactly as though the user had disabled script herself.
- The framed document is loaded into a unique origin, which means that all same-origin checks will fail, unique origins match no other origins ever, not even themselves. Among other impacts, this means that the document has no access to data stored in any origin's cookies or any other storage mechanisms (DOM storage, IndexDB, etc.).
- The framed document cannot create new windows or dialogs (no `window.open` or `target="_blank"`, for instance).
- Forms cannot be submitted.
- Plugins will not load.
- The framed document can only navigate itself, not its top-level parent. Setting `window.top.location` will throw an exception, and clicking on link with `target="_top"` will have no effect.
- Features that trigger automatically (autoforward form elements, autoplaying videos, etc.) are blocked.
- Pointer lock cannot be obtained.
- The `seamless` attribute is ignored on `iframe`s: the framed document contains.

This is nicely draconian, and a document loaded into a fully sandboxed `iframe` poses very little risk indeed. Of course, it also can't do much of value: you might be able to get away with a fall sandbox for some static content, but most of the time you'll want to loosen things up a bit.

With the exception of plugins, each of these restrictions can be lifted by adding a flag to the sandbox attribute's value. Sandboxed documents can never run plugins, as plugins are unsandboxed native code, but everything else is fair game:

- `allow-forms` allows form submission.
- `allow-popups` allows (shock!) popups.
- `allow-pointer-lock` allows (surprise!) pointer lock.
- `allow-same-origin` allows the document to maintain its origin pages loaded from `https://example.com/` will retain access to that origin's data.
- `allow-scripts` allows JavaScript execution, and also allows features to trigger automatically (as they'd be trivial to implement via JavaScript).
- `allow-top-navigation` allows the document to break out of the frame by navigating its top-level window.

With these in mind, we can evaluate exactly why we ended up with the specific set of sandboxing flags in the Twitter example above:

- `allow-scripts` is required, as the page loaded into the frame runs some JavaScript to deal with user interaction.
- `allow-popups` is required, as the button pops up a tweeting form in a new window.
- `allow-forms` is required, as the tweeting form should be submittable.
- `allow-same-origin` is necessary, as twitter.com's cookies would otherwise be inaccessible, and the user couldn't log in to post the form.

One important thing to note is that the sandboxing flags applied to a frame also apply to any windows or frames created in the sandbox. This means that we have to add `allow-forms` to the frame's sandbox, even though the form only exists in the window that the frame pops up.

With the `sandbox` attribute in place, the widget gets only the permissions it requires, and capabilities like plugins, top navigation, and pointer lock remain blocked. We've reduced the risk of embedding the widget, with no ill effects. It's a win for everyone concerned.

Privilege Separation

Sandboxing third-party content in order to run their untrusted code in a low privilege environment is fairly obviously beneficial. But what about your own code? You trust yourself, right? So why worry about sandboxing?

I'd turn that question around: if your code doesn't need pluginz, why give it access to plugins? At best, it's a privilege you never use, at worst it's a potential vector for attackers to get a foot in the door. Everyone's code has bugs, and practically every application is vulnerable to exploitation in one way or another. Sandboxing your own code means that even if an attacker successfully subverts your application, they won't be given full access to the application's origin; they'll only be able to do things the application could do. Still bad, but not as bad as it could be.

You can reduce the risk even further by breaking your application up into logical pieces and sandboxing each piece with the minimal privilege possible. This technique is very common in native code: Chrome, for example, breaks itself into a high-privilege browser process that has access to the local hard drive and can make network connections, and many low-privilege renderer processes that do the heavy lifting of parsing untrusted content. Renderers don't need to know the disk, the browser takes care of giving them all the information they need to render a page. Even if a clever hacker finds a way to corrupt a renderer, she doesn't gotten very far, as the renderer can't do much of interest on its own: all high-privilege access must be routed through the browser's process. Attackers will need to find several holes in different pieces of the system order to do any damage, which hugely reduces the risk of successful pwnage.

Safely sandboxing `eval()`

With sandboxing and the postMessage API, the success of this model is fairly straightforward to apply to the web. Pieces of your application can live in sandboxed `iframe`s, and the parent document can broker communication between them by posting messages and listening for responses. This sort of structure ensures that exploits in any one piece of the app do the minimum damage possible. It also has the advantage of forcing you to create clear integration points, so you know exactly where you need to be careful about validating input and output. Let's walk through a toy example, just to see how that might work.

Evalbox is an exciting application that takes a string, and evaluates it as JavaScript. Wow, right? Just what you've been waiting for all these long years. It's a fairly dangerous application, of course, as allowing arbitrary JavaScript to execute means that any and all data on origin has to offer is up for grabs. We'll mitigate the risk of Bad Things™ happening by ensuring that the code is executed inside of a sandbox, which makes it quite a bit safer. We'll work our way through the code from the inside out, starting with the frame's contents:

```
<!-- frame.html -->
<!DOCTYPE html>
<html>
  <head>
    <title>Evalbox's Frame</title>
    <script>
      window.addEventListener('message', function (e) {
        var messageData = e.data;
        var result = 1;
        try {
          result = eval(e.data);
        } catch (e) {
          result = "eval() threw an exception.";
        }
        messageData.postMessage(result, event.origin);
      }, false);
    </script>
  </head>
</html>
```

Inside the frame, we have a minimal document that simply listens for messages from its parent by hooking into the `message` event of the `window` object. Whenever the parent executes postMessage on the frame's contents, this event will trigger, giving us access to the string our parent would like us to execute.

In the handler, we grab the `source` attribute of the event, which is the parent window. We'll use this to send the result of our hard work back up once we're done. Then we'll do the heavy lifting, by passing the data we've been given into `eval()`. This call has been wrapped up in a try block, as banned operations inside a sandboxed `iframe` will frequently generate DOM exceptions; we'll catch those and report a friendly error message instead. Finally, we post the result back to the parent window. This is pretty straightforward stuff.

The parent is similarly uncomplicated. We'll create a tiny UI with a `textarea` for code, and a `button` for execution, and we'll pull in `frame.html` as a sandboxed `iframe`, allowing only script execution:

```
<textarea id="code"></textarea>
<button id="safe">eval() in a sandboxed frame</button>
<iframe sandbox="allow-scripts"
        id="sandbox0"
        src="frame.html"></iframe>
```

Now we'll wire things up for execution. First, we'll listen for responses from the `iframe` and `alert`s. Then we'll ask our `iframe`s. Presumably a real application would do something less annoying:

```
window.addEventListener('message',
    function (e) {
      // Ignore messages from origins such the 'allow-same-origin'
      // header have 'null' rather than a valid origin. This means you still
      // have to be careful about accepting data via the messaging API you
      // create. Check that source, and validate those inputs!
      var frame = document.getElementById('sandbox0');
      if (e.origin === "null" && e.data && e.source === frame.contentWindow)
        alert('Result: ' + e.data);
    });

Next, we'll hook up an event handler to clicks on the `button`. When the user clicks,
we'll grab the current contents of the `textarea`, and pass them into the frame for
execution:

function evaluate() {
  var frame = document.getElementById('sandbox0');
  var code = document.getElementById('code').value;
  // Note that we're sending the message to "*", rather than some specific
  // origin. Sandboxed iframes which lack the 'allow-same-origin' header
  // don't have an origin which you can target; you'll have to send to any
  // origin, which might also some esoteric attacks. Validate your output!
  frame.contentWindow.postMessage(code, "*");
}

document.getElementById('safe').addEventListener('click', evaluate);

Easy, right? We've created a very simple evaluation API, and we can be sure that code
that's evaluated doesn't have access to sensitive information like cookies or DOM
storage. Likewise, evaluated code can't load plugins, pop up new windows, or any of
a number of other annoying or malicious behaviors.

You can do the same for your own code by breaking monolithic applications into
single-purpose components. Each can be wrapped in a simple messaging API, just
like what we've written above. The high privilege parent window can act as a
controller and dispatcher, sending messages into specific modules that each have the
fewest privileges possible to do their jobs, listening for results, and ensuring that
each module is well fed with only the information it requires.

Note, however, that you need to be very careful when dealing with framed content
that comes from the same origin as the parent. If a page on `https://example.com/`
frames another page on the same origin with a sandbox that includes both the
`allow-same-origin` and `allow-scripts` flags, then the framed page can reach up into
the parent, and remove the sandbox attribute entirely.
```

Play in your sandbox

Sandboxing is available for you now in a variety of browsers (Firefox 17+, IE10+, and Chrome 19 behind the "Enable experimental Web Platform features" flag at about:flags [latest]). Applying the sandbox attribute to an `iframe` only include allows you to grant certain privileges to the content they display, and those privileges should be relevant to the content in question, correctly. This gives you the opportunity to reduce the risk associated with the inclusion of third-party content, and beyond what is already possible with Content Security Policy.

Moreover, sandboxing is a powerful technique for reducing the risk that a clever attacker will be able to exploit holes in your own code. By separating a monolithic application into a set of sandboxed components, each responsible for a small chunk of self-contained functionality, attackers will be forced to not only compromise specific frames' content but also their controller. That's a much more difficult task, especially since the controller can be greatly reduced in scope. You can spend your security-related effort auditing that code if you ask the browser for help with the rest.

That's not to say that sandboxing is a complete solution to the problem of security on the internet. It offers defense in depth, and unless you have control over your users' clients you can't rely on browser support for all your users' data. Or control your users clients — an enterprise environment, for example — honey). Someday, but for now sandboxing is another layer of protection in your stronger defenses, it's not a complete defense upon which you can solely rely. Still, layers are excellent. I suggest making use of this one.

Further Reading

- "Privilege Separation in HTML5 Applications" is an interesting paper that works through the design of a small framework, and its application to three existing HTML5 apps.
- Sandboxing can be even more flexible when combined with two other new iframe attributes: srcdoc, and seamless. The former allows you to populate a frame with content without the overhead of an HTTP request, and the latter allows style to flow into the framed content. Both have fairly miserable browser support at the moment (Chrome and WebKit nightlies), but will be an interesting combination in the future. You could, for example, sandbox comments on an article via the following code:

```
<iframe sandbox seamless
        srcdoc="p{This is a user's comment!
                It can't execute script!
                Hooray for safety!}</p"></iframe>
```

Last updated: Jun 4, 2010 — Improve article

Except as otherwise noted, the content of this page is licensed under the Creative Commons Attribution 4.0 License, and code samples are licensed under the Apache 2.0 License. For details, see the Google Developers Site Policies.