External Content

The **Chrome Apps security model** disallows external content in iframes and the use of inline scripting and eval(). You can override these restrictions, but your external content must be isolated from the app.

Isolated content cannot directly access the app's data or any of the APIs. Use cross-origin XMLHttpRequests and post-messaging to communicate between the event page and sandboxed content and indirectly access the APIs.

API Sample: Want to play with the code? Check out the **sandbox** sample.

Referencing external resources

The **Content Security Policy** used by apps disallows the use of many kinds of remote URLs, so you can't directly reference external images, stylesheets, or fonts from an app page. Instead, you can use use cross-origin XMLHttpRequests to fetch these resources, and then serve them via blob: URLs.

Manifest requirement

To be able to do cross-origin XMLHttpRequests, you'll need to add a permission for the remote URL's host:

```
"permissions": [
   "...",
   "https://supersweetdomainbutnotcspfriendly.com/"
 ]
```

Cross-origin XMLHttpRequest

Fetch the remote URL into the app and serve its contents as a blob: URL:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'https://supersweetdomainbutnotcspfriendly.com/image.png', true);
xhr.responseType = 'blob';
```

```
xhr.onload = function(e) {
  var img = document.createElement('img');
  img.src = window.URL.createObjectURL(this.response);
  document.body.appendChild(img);
};

xhr.send();
```

You may want to **save** these resources locally, so that they are available offline.

Embed external web pages

API Sample: Want to play with the code? Check out the **browser** sample.

The **webview** tag allows you to embed external web content in your app, for example, a web page. It replaces iframes that point to remote URLs, which are disabled inside Chrome Apps. Unlike iframes, the webview tag runs in a separate process. This means that an exploit inside of it will still be isolated and won't be able to gain elevated privileges. Further, since its storage (cookies, etc.) is isolated from the app, there is no way for the web content to access any of the app's data.

Add webview element

Your webview element must include the URL to the source content and specify its dimensions.

```
<webview src="http://news.google.com/" width="640" height="480"></webview>
```

Update properties

To dynamically change the src, width and height properties of a webview tag, you can either set those properties directly on the JavaScript object, or use the setAttribute DOM function.

```
document.querySelector('#mywebview').src =
    'http://blog.chromium.org/';
// or
document.querySelector('#mywebview').setAttribute(
    'src', 'http://blog.chromium.org/');
```

Sandbox local content

Sandboxing allows specified pages to be served in a sandboxed, unique origin. These pages are then exempt from their Content Security Policy. Sandboxed pages can use iframes, inline scripting, and eval(). Check out the manifest field description for **sandbox**.

It's a trade-off though: sandboxed pages can't use the chrome.* APIs. If you need to do things like eval(), go this route to be exempt from CSP, but you won't be able to use the cool new stuff.

Use inline scripts in sandbox

Here's a sample sandboxed page which uses an inline script and eval():

```
<html>
  <body>
    <h1>Woot</h1>
    <script>
      eval('console.log(\'I am an eval-ed inline script.\')');
    </script>
  </body>
</html>
```

Include sandbox in manifest

You need to include the sandbox field in the manifest and list the app pages to be served in a sandbox:

```
"sandbox": {
  "pages": ["sandboxed.html"]
}
```

Opening a sandboxed page in a window

Just like any other app pages, you can create a window that the sandboxed page opens in. Here's a sample that creates two windows, one for the main app window that isn't sandboxed, and one for the sandboxed page:

NOTE: A sandboxed window will not have access to the chrome.app APIs. If a callback is provided to app.window.create it will be run, but will not have the sandboxed window provided to it.

```
chrome.app.runtime.onLaunched.addListener(function() {
  chrome.app.window.create('window.html', {
    'bounds': {
      'width': 400,
      'height': 400,
      'left': 0,
      'top': 0
    }
  });

  chrome.app.window.create('sandboxed.html', {
    'bounds': {
      'width': 400,
      'height': 400,
      'left': 400,
      'top': 0
    }
  });
});
```

Embedding a sandboxed page in an app page

Sandboxed pages can also be embedded within another app page using an iframe:

```
<!DOCTYPE html>
<html>
<head>
</head>
```

```
 <body>
   <p>I am normal app window.</p>

   <iframe src="sandboxed.html" width="300" height="200"></iframe>
 </body>
</html>
```

Sending messages to sandboxed pages

There are two parts to sending a message: you need to post a message from the sender page/window, and listen for messages on the receiving page/window.

Post message

You can use postMessage to communicate between your app and sandboxed content. Here's a sample background script that posts a message to the sandboxed page it opens:

```
var myWin = null;

chrome.app.runtime.onLaunched.addListener(function() {
  chrome.app.window.create('sandboxed.html', {
    'bounds': {
      'width': 400,
      'height': 400
    }
  }, function(win) {
       myWin = win;
       myWin.contentWindow.postMessage('Just wanted to say hey.', '*');
     });
});
```

Generally speaking on the web, you want to specify the exact origin from where the message is sent. Chrome Apps have no access to the unique origin of sandboxed content, so you can only whitelist all

origins as acceptable origins ('*'). On the receiving end, you generally want to check the origin; but since Chrome Apps content is contained, it isn't necessary. To find out more, see **window.postMessage**.

Listen for message and reply

Here's a sample message receiver that gets added to your sandboxed page:

```
var messageHandler = function(event) {
  console.log('Background script says hello.', event.data);

  // Send a reply
  event.source.postMessage(
      {'reply': 'Sandbox received: ' + event.data}, event.origin);
};

window.addEventListener('message', messageHandler);
```

For more details, check out the **sandbox** sample.