October 08: Introduction to Web Security

Scribe: Rohan Padhye

October 8, 2015

Web security is an important topic because web applications are particularly hard to secure, and are one of the most vulnerable/buggy computer systems in existence today.

1 Why is it so complex?

There are several issues particular to web applications that make reasoning about their security complex:

- Web applications run client-side code (JavaScript) that is downloaded from a remote server.
- One web application may include code (i.e. scripts) and data (e.g. images) downloaded from multiple remote servers in the same page.
- Multiple web applications may be simulataneously active within the same browser session in the form of windows, tabes or frames.
- The Document Object Model (DOM) is a complex resource containing a mix of data and executable scripts.
- JavaScript code running in a web page may download additional resources by embedding images, other scripts or making remote HTTP calls using `XmlHttpRequest` (a.k.a. AJAX). These HTTP requests are made in the background without the user's explicit knowledge or action. Further, the browser treats most such requests in the same way as if a user entered the URL in the address bar, in that it attaches to the request all cookies that it has stored and that match the cookie origin policy.
- Another source of background communication are Web Sockets that allow persistent client-server communication when a web page.
- URL parsing poses another source of risk as browsers may interpret some URLs as code (e.g. the `javascript:` protocol).
- Web applications may have access to lot of sensitive information apart from data from the application itself; for example some applications can access a user's location (if permission has been granted before).
- Browser plugins further complicate the matter since depending on the permissions granted to them they may be able to read/write data from arbitrary web applications as well as the user's filesystem.

2 Client-side Security

A web browser's **threat model** is that some sites may be completely compromised (i.e. their server-side content is controlled by a maliciou adversary) and the user has inadvertently opened such a web page alongside a legitimiate web app or has some content from such a malicious site embedded in a web page downloaded from a legitimiate web app that is rendering sensitive content.

The **goal** of the web browser's security layer is to prevent malicious site(s) from reading/writing data from/to other web applications.

Consider the following example browser session:

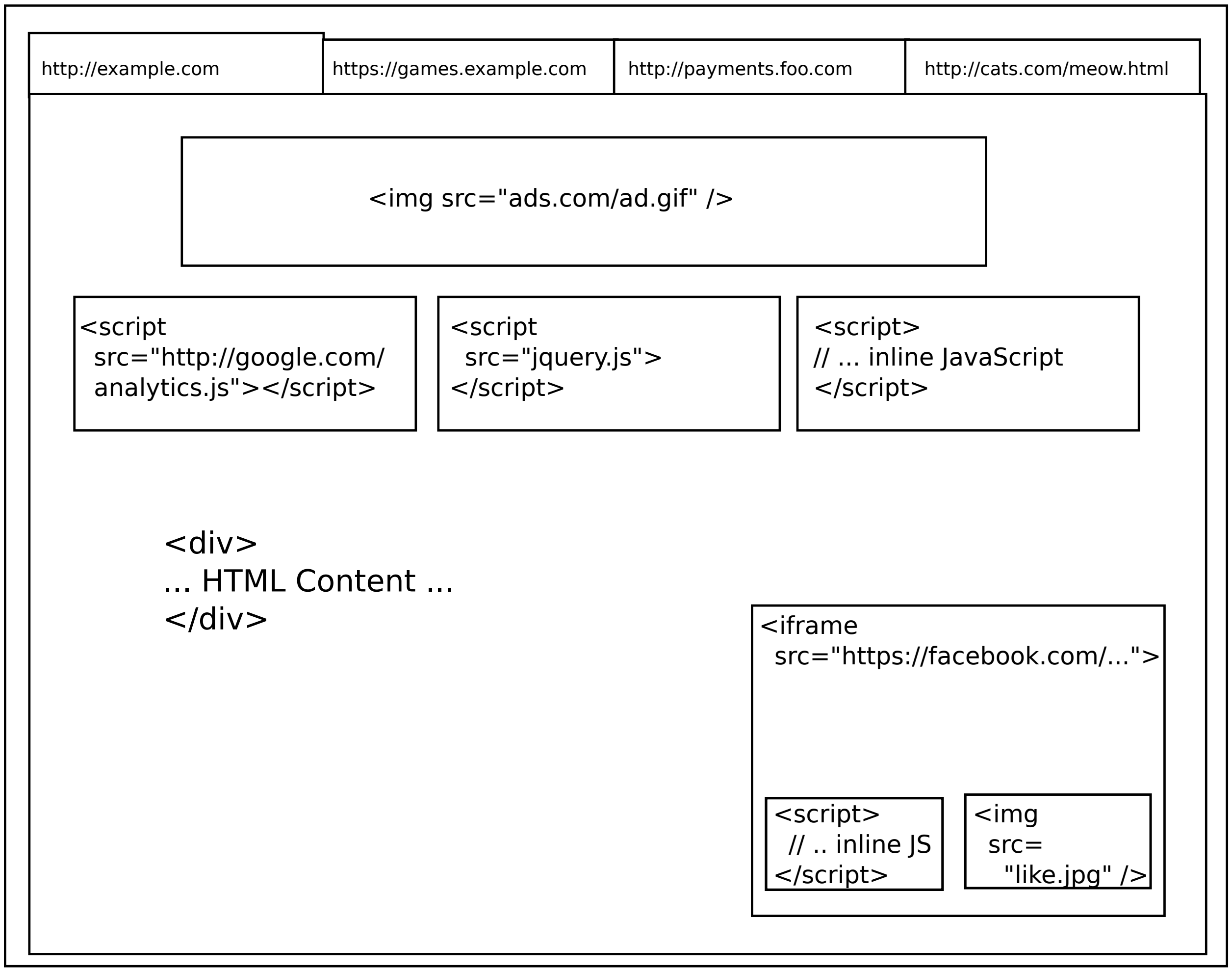
http://example.com
https://games.example.com
http://payments.foo.com
http://cats.com/meow.html
<img src="ads.com/ad.gif" />
<script src="http://google.com/analytics.js"></script>
<script src="jquery.js"></script>
<script> // ... inline JavaScript </script>
<div> ... HTML Content ... </div>
<iframe src="https://facebook.com/...">
<script> // .. inline JS </script>
<img src= "like.jpg" />

Figure 1: Sample browser session with multiple tabs open and a web page with multiple embedded resources from different origins.

In this example, what communication is allowed between various resources in the browser?

1. Should code from `jquery.js` be able to access DOM elements on the web page?
   - Yes, it is a DOM manipulation library and intentionally embedded from the same origin.

2. Should code from `analytics.js` be able access DOM elements on the web page?
    - Maybe not, but embedding code in a `<script>` tag implies that it runs as if it were code from the origin of the web page, so it surely can.
3. Should the code running in the Facebook `<iframe>` be able to read/modify DOM elements on the containing web page?
    - No, the Facebook frame has been embedded for external functionality and it should not access the containing page's DOM, and this is enforced by the same-origin-policy.
4. Should the code running in the main web page be able to read/modify contents in the Facebook `<iframe>`?
    - No, because the Facebook `<iframe>` has content that may be private to the user in case they are logged in and this should not be information that the containing web page should be able to read, and this is enforced by the same-origin-policy.

3 Same Origin Policy

Definition 1 *An origin of a resource downloaded from some URL is the tuple containing the URL's protocol (e.g. http, https, ftp, file), the host (e.g. example.com, payments.foo.com) and the port (e.g. 80, 443). Two resources are from the same origin if and only if all these parts of their URLs exactly match.*

This definition of *origin* is used by browsers to enforce the same-origin-policy, which (informally) is as follows:

Definition 2 *Code that belongs to one origin may not access resources (data or functionality) that belongs to a different origin.*

For example:

- Resources from `http://example.com` and `https://example.com` cannot communicate, because their origins have different protocols.
- Resources from `https://example.com/a.html` and `https://example.com/b.html` can communicate despite coming from different paths because they have the same origin.
- Resources from `https://example.com:81` and `https://example.com` cannot communicate, because the latter has an origin with the implied port of 443.

3.1 How is a resource's origin determined?

- Each window/tab/frame gets the origin from its location URL.
- Scripts included on a web page, either inline or through embedded of external files using the `<script>` tag, are considered to have an origin of the containing web page, regardless of the actual source of the script file.

- Passive content has no origin and is not allowed to access any other resources (e.g. the image rendering routine in a browser should be isolated so that it cannot be tricked into accessing data from *any* other resource).
- `XmlHttpRequest` can only make requests to resources with the same origin as their script. This rule has certain exceptions, as some remote end-points can specifically allow cross-origin requests (that only come from certain other origins or perhaps only those that do not send cookies with them) by the use of HTTP headers. Hence, browsers might make a `HEAD` request to remote resources to determine this before allowing/denying a script's AJAX call.

3.2 Cross-Origin Communication

Communication between resources downloaded from different origins may be possible if they both explicitly allow it, in one of the following two ways:

1. `window.postMessage` - this is a standard API that allows a script to send a message to another window.
   - For example, `iFrame.postMessage("user=bob","https://login.foo.com")`.
   - The browser will send this message if it determines that the origin of the target window object, in this case `iFrame`, exactly matches the destination in the message, in this case `https://login.foo.com`.
   - The window that receives the message can choose to handle it by registering an event handler for events of type `message`, which when received have a property of `origin` which can be validated to ensure that they are coming from a source that the application expects to send messages.
   - Care must be taken by the code that is receiving the message to check for an exact match with the source origin.
2. `document.domain` - a web page can request the browser to change its own origin to one with a hostname that is a superdomain of the current domain (e.g. `login.foo.com` can change its own domain to `foo.com`) so that resources from different sub-domains can talk to each other.
   - Resources that change their domain can only access data from other resources that have also changed their domain. In the above example, the web app can't simply access data from a window with origin `foo.com`, unless that window also changes its own domain (perhaps to itself, but this operation signals that is is ready to yeild access to other resources which have undergone domain changes).

In some sense, the same-origin-policy is similar to privilege separation used for inter-process communication or sandboxed environments in operating systems.

4 Cookies

Cookies are set by web applications either as part of the HTTP response or by JavaScript code using `document.cookie`. They are also sent along with every HTTP request if the URL of the request matches the cookie-origin policy, which is different from the Same-Origin policy.

Definition 3 *The origin of a cookie is a pair of a domain (e.g.* `example.com`*) and path (e.g.* `/sites/xyz`*).*

Definition 4 *A cookie having some domain and path as its origin is sent along with every HTTP request to a URL whose domain is a sub-domain of the cookie's domain and whose path is a child of the cookie's path.*

For example, a cookie with origin `(example.com, /sites/xyz)` is also sent to a HTTP request for `http://games.example.com/sites/xyz/abc`. Note that the protocol and port do not play a part in this check.

4.1 Cookie-policy vs. Same-origin-policy

The inconsistency in the cookie policy and same-origin policy result in several unexpected data flows. For example, even though cookies are not sent with HTTP requests to URLs with the same host and different paths, the scripts that are executed from such paths have the same origin and hence can access each other's cookies using the JavaScript `document.cookie` API. Conversely, the same origin policy forbids communication between two resources that have different hosts but cookies are sent along with HTTP requests to URLs whose hosts are super-domains of the cookie's domain (e.g. cookie from `example.com` is accessible from `games.example.com`).

4.2 Server controls on cookies

Web servers can control how the cookies they set are moved around by the use of special attributes. For example:

- The `httpOnly` attribute instructs the browser to only pass around the cookie with HTTP requests, and not allow JavaScript to access it via `document.cookie`.
- The `secure` attribute instructs the browser to only passs around the cookie with HTTPS requests, and not plain unencrypted HTTP requests so as to prevent eavesdroppers from reading sensitive information.

5 Other Resources

5.1 DOM Nodes

Elements in the DOM inherit the origin of the window/tab/frame they belong to. A special case is resources like external images which are embedded on a web page; some browsers do not allow JavaScript running the web page's origin to access the contents of external images since they might contain sensitive data.

5.2 Windows/Frames

Browser windows/frames belong to the origin of their location URL. A web server can prevent a web page to be embedded inside an `<iframe>` through the use of special HTTP Headers. This may be done to prevent containing web pages from triggering some sort of screen capture or tricking the user into pressing a button inside the iFrame by clever placement of DOM nodes.