Web Security 1: Same-Origin and Cookie Policy

Dayeol Lee and Eldon Schoop

1 Overview of HTTP and Web Application Structure

Common web applications are comprised of documents loaded over the HyperText Transfer Protocol (HTTP). These documents can contain page structure (HTML: HyperText Markup Language), styling information (Cascading Style Sheets: CSS), and code (JS: JavaScript) which can modify the document, respond to input, and make web requests.

1.1 Hypertext Transfer Protocol

HTTP is a protocol to transfer data between client and server on the web. HTTP consists of request and response. HTTP request contains URL, which is a string combination of protocol, hostname, port, path, query, and fragment. If a client browser sends a request to a web server, the server sends a response back with web page data.

The two main types ("Methods") of requests are `GET`, which is typically stateless, and `POST`, which has side-effects and contains a *body* for sending information to the web server. `GET` requests are used to download web pages, and `POST` requests are commonly used to transmit form submission data to a web server.

HTTP Requests contain the Method (GET/POST), Path, HTTP Version, and headers (e.g., User-Agent, Referrer, Language). POST requests additionally contain a Data field.

HTTP Responses contain the HTTP Version, Status Code (e.g., 200 OK, 404 Not Found, 500 Internal Server Error), headers, and data (which could be a web page).

1.2 The Document Object Model

Web page data is mainly composed of thre languages: HTML, CSS, and Javascript. HTML defines the whole structure of the web document, CSS defines the presentation style of the document, and Javascript manipulates the contents and style of the web page. JavaScript is particularly powerful because it can *change* the DOM, e.g., changing images, style, hiding elements, and even changing the cursor. These elements together form the structure and style of the web page, constituting a tree structure called the Document Object Model (DOM).

To interpret the documents and create the DOM, web browsers *parse* the HTML and CSS, and execute the JS in its *JS Engine*. The browser uses the outputs of the parser to create the DOM and modifies it with via evaluated JS. The *painter* renders a bitmap image from these inputs, creating the web page view.

Web pages may contain *Frames*, a web page embedded within another. This enables content to be aggregated from multiple sources on the client side, with the caveat that frames may only draw within their own rectangle, and may not change, or be changed by, the parent web page.

2 Client-side Security

Multiple tabs, multiple scripts on each page. What should be able to communicate with what?

Client-side security for web applications is particularly challenging because web development exists in the "wild west"–standards exist, but are often interpreted differently between websites, browsers, platforms, devices, and developers. Users may also have many web pages open in different windows or tabs, each page with multiple scripts running. So what should be allowed to communicate with what?

2.1 Same-Origin Policy

Simply put, the Same-Origin Policy is intended to ensure each site in the browser is isolated from all others, and that multiple pages from the same site are not isolated.

2.1.1 What is an origin?

An *Origin* is the granularity of protection for the Same Origin Policy. It consists of a protocol, hostname, and port: https://example.com:8888/examples/index.html

Browsers determine if a site's origin is the same as another via *string matching*! If the origin string does not perfectly match, then the origin is not the same. Note the *path* is not included in the origin string. In the same-origin policy, one origin cannot read or modify pages from different origins, and javascript on one page cannot read or modify pages from different origins. Note embedded elements such as images are copied from the external origin and rendered locally, so they have the same origin as the rendering page. IFrames preserve the origin of the URL they are served from.

2.1.2 Examples

The following examples show which origins are allowed to communicate (taken from slides).

Originating Document	Accessed Document	Allowed?
http://example.com/**a/**	http://example.com/**b/**	Yes
http://example.com/	http://**www**.example.com/	No
http://example.com/	**https**://example.com/b/	No
http://example.com**:81**/	http://example.com**:82**/	No
http://example.com/	http://example.com**:81**/	No

2.2 Cross-Origin Communication

Different origins are allowed to communicate via *cross-origin communication*, either through overriding *both pages' origins* manually through JavaScript, or through the HTML5 postMessage API. The postMessage API allows a page to receive messages from other origins, and relies on the developer to decide how to handle it based off of the origin from the other page.

3 HTTP Cookie

Since HTTP requests and responses do not define any states, the HTTP server and client use cookies to maintain state. When a client (i.e., browser) connects to a server for the first time, it has no states nor cookies. The server responses back with `Set-Cookie` header containing a cookie, which will be stored by the client. If the client sends following requests with the cookie, the server can maintain or alter the internal state for the client based on that cookie value. An example usage of cookie is the session ID. Once the a user loged in to a web server, the server has to assign a unique session ID to the user, send it with `Set-Cookie` header, and maintain the session as long as the user sends the valid session ID in the cookie. Cookies can be accessed by Javascript via `document.cookie`.

3.1 Cookie Policy

The cookies should be sent only to the web servers within the cookies scope because the cookie could include some sensitive information (e.g., session ID). The server can inform the browser about cookie scope by adding additional key-value pairs to `Set-Cookie` header as follows:

- `domain` indicates a subdomain of servers that are allowed to see the cookie. The browser will use the exact domain of URL as the cookies domain if not set.
- `path` indicates the prefix of URLs' path that are supposed to get the cookie. Note that `path` was intended for performance, not security. Web pages having the same origin still can access cookie via `document.cookie` even though the paths are mismatched.
- If `secure` presents, the browser should send the cookie only over HTTPS connections.
- If `expires` presents, the browser should store the cookie with expiration date, and use the cookie only until that date.
- If `HttpOnly` presents, the browser should not allow script to read the cookie via `document.cookie`.

When the client browser receives `Set-Cookie` header with `domain`, it checks if the `domain` complies with URL domain so that any server cannot control cookies of other sites. Any domain-suffix can be the cookies domain, except the top level domain-suffix such as `.com`, `.net`, `.org`, and so on. If the cookie-domain is invalid, the browser just does not set the cookie. The browser uses the cookies domain and other scopes to decide whether the cookie should be sent to the server. Any cookies that have matching scope are sent along with the request.

3.2 Cookie Policy vs. Same-Origin Policy

Cookie policy should cooperate with same-origin policy such that the browser does not leak any data to the other origins, like `document.cookie`. The browser can simply prohibit any access to `document.cookie` from different-origin site even though it is within the cookie's domain. For example, `apple.foo.com` cannot access `document.cookie` of `banana.foo.com` despite the domain of the cookie was `*.foo.com`.

4 Session

Session is a sequence of HTTP transactions from one client. A server can use a session to maintain authentication state of a user. For example, a server can regard a session as authorized if a user tied to that session once logged in. A session can last either long (e.g., Gmail account) or short (e.g., banking) period of time.

To keep track requests in a session, the server uses *session token*, which is a unique string. The browser stores the token and uses it to maintain the session with the server.

Several options exist to store session tokens. Every option has its own downside.

- **Browser Cookie**: we can use the cookie to store and send session token. However, this option is vulnerable to CSRF attack as we will see in next lectures.
- **URL query**: we can embed the token in the URL query. However, this may leak session token to opponents by `Referer` header, or just by a human error, like copying and pasting the URL link.
- **Hidden Form Field**: the browser can add a hidden form field containing the token so that the token can be sent upon the form submission. However, this cannot be used for long sessions because the form field will disappear after submission. Also, Javascript can access and leak the token disregarding the same-origin policy.