Web security - Browsers

Same Origin Policy

This is the browser's security policy. Under this policy, two pages can interact only if they're from the same origin (scheme, host, port).

- Cookies are sent to the origin that set them
- Javascript can modify pages from the same origin

Personal Sites on cs.berkeley.edu

You can host a personal website at *http://cs.berkeley.edu/~you*.

Security implication: all users' sites are from the same origin. You can steal other sites' cookies or modify their pages. However, you can't run any server-side logic on these websites, so this might not be very significant.

How to Create a Secure Login

Make sure attackers can't use JS to spy on your login keystrokes

One proposal is to use another origin to host the login. The same origin policy will prevent Javascript from interacting with the login page. But if the login mechanism is on another origin, how do we get the cookie back to the application?

How the Same Origin Policy Applies to Cookies

Consider these three domains:

- bar.com
- a.bar.com
- b.bar.com

When the server sets a cookie, it can specify what domains to send it back to. Suppose b.bar.com sets a cookie. It can say:

domain=b.bar.com	only sent to b.bar.com
domain=.bar.com	sent to any subdomain of bar.com
domain=bar.com	only sent to bar.com

Modern browsers special case this mechanism to prevent people from setting cookies for domains like *.com* or *.co.uk*.

Entailments of this Policy

As a server, you can choose who can read the cookie. You can *voluntarily* give access to another subdomain. However, this allows some other server to set cookies that will be sent to you.

Example

Consider Livejournal. They have a site at *livejournal.com*, and users have subdomains.

A user that controls *daw.livejournal.com* can set cookies that will be sent to *livejournal.com* or *fakestevejobs.livejournal.com*.

Session Fixation

HTTP is stateless. In order to link together all your requests, the server sends you a *session cookie* (which is a unguessable random string). Any request with that cookie value set is considered yours. This way, you only have to log in once.

Session fixation refers to an attacker being in control of the session identifier, instead of the server.

1. The attacker makes a request to the server.
2. The server creates a new session and sends the attacker the session identifier *i*.
3. The attacker arranges for the victim to interact with the server using session identifier *i*.
4. The attacker interacts with the server, also using *i*.

Consider what happens, depending what the victim does in step 3.

- The victim enters banking information to complete a purchase. In step 4, the attacker could retrieve the banking credentials.
- The victim logs in to an account. In step 4, the attacker could do anything in the victim's account.

Forcing a Session Identifier on a Victim

Servers often accept the session identifier as a URL parameter. This is to accommodate users who don't accept cookies. The attacker can just send the victim a link like *http://amazon.com/?sessionid=1234...f*.

Often this feature is enabled by default in the web framework, and the developer just hasn't turned it off.

IP Address Validation

A proposal is to associate an IP address with each session, and only allow that address to access the session. This is not practical, because:

- Many users may be behind a single NAT, which reports the same source IP for everyone.
- A user's IP may change throughout a session, for example, due to proxies.

Google

Recently, Google is merging products formerly on separate subdomains into the *google.com* domain. Perhaps in the future, if a single product gets owned, all of Google will be vulnerable.

The Binary Nature of Same Origin Policy

We consider the same origin policy to be “binary,” because there are only two levels of access:

- If two resources are from the same origin, the browser allows all access.
- If they are from different origins, there is almost no interaction allowed.

External Scripts

Suppose the page http://yelp.com/foo.html uses a script from maps.google.com:

```
<script src="http://maps.google.com/api.js">
```

The browser loads this script from maps.google.com and runs it as part of *http://yelp.com*’s origin.

Thus, Yelp must trust Google with complete access.

Cross Site Request Forgery

Here are 5 defenses against CSRF.

1. Referer header

The browser tells the server the URL of the page that sent the request. The server can check if this URL is from its own origin.

2. CSRF tokens

Any request that changes state must be sent with a special value. Variants include:

- Session independent: the value is some random string. The server has to keep a record of all valid token values.
- Cookie double: the same as some cookie’s value. The server can just check to see if the value and the cookie’s value match.

3. XHR with special header

XMLHttpRequest allows developers to set custom headers with a request. It also requires that the requests go to the same origin as the requesting page. The server checks for the presence of the custom header. No other origin can make a request with custom headers.

4. Origin header

Similar to the Referer header, but only reports the origin, rather than the full URL. This is better in terms of privacy.

5. Webkeys

All URLs are just the origin plus some 128-bit hexadecimal string that attackers can't guess. This facilitates mashups, since you can just give away a URL to grant access to your session.

Similarity to Java Mechanisms

Defenses (1), (3), and (4) are analogous to *stack inspection*. The referring page/origin is like the stack frame just outside the request.

Defenses (2) and (5) are analogous to *capabilities*. The token or URL path is unforgeable because it is unguessable.

Origin Header Privacy

It gives less information than Referer, but just the origin alone is still a significant disclosure. One proposal is to send "Origin: other" for cross-origin requests.

This may not be adequate for sever developers. For example, eBay may want to allow requests from PayPal but no other sites.

Defense Implementation Goals

Browser vendors want to make sure they don't break stuff. Even if it only breaks less popular sites, it's still a big deal.

And why bother inventing the Origin header, when we have all these other perfectly good defenses? It's easier, for web developers to use, that's why.

Conclusion

The browser has ambient authority in its cookies. Anyone can cause the browser to send its cookies.