CS261 Lecture 15: Web Security on Browsers

Scribe: Charles Reiss

19 October 2009

1 Cookies

- kept by client, sent to server on every request;
- most common implementation of single sign-on over HTTP;
  - older techniques — URL rewriting, HTTP Basic authentication;
- typical pattern:
  - Server sends cookie `session_id=`*`random session ID`*;
  - Client sends back `session_id=...` on every request to the server
  - server maps session ID → logged user name → sensitive personal information
- restrictions on cookies
  - Only sent to originating domain
  - Time limit — session (until browser closed) or persistent
- *not* capabilities
  - more like ambient authority; the cookie jar is the bag of rights indexed on browser and URL;
  - URL rewriting is capabilities in the web context;
  - confused deputy attacks using cookies: CSRF;

1.1 Cross-Site Request Forgery (CSRF)

Confused deputy attack on web browser.
Example:

- Preconditions:
  - User is logged into GMail (a session ID cookie exists)
  - GMail has a guessable URL with side-effects (e.g. `gmail.com/delete-cur-msg`)
- Attacker.com points a form to `gmail.com/delete-cur-msg`
- User clicks on it, and browser sends POST request to GMail *with the session cookie*.
- Click actually not needed — JS can submit form, use images for GET requests, etc.

CSRF defenses:

- Secret token (usually in URL)
  - $\approx$ URL as capabilities
  - might extend to allow sharing
  - confidentiality issues with using URL
    - stored in browser history
    - browser sends as referer, avoid with:
      - laundering redirect
      - fragment identifier (`http://foo/bar#`*`fragment`*) and JavaScript (fragment is not ordinarily sent to server)
      - HTTPS (maybe)
  - granularity issue: leaking one URL might leak all access to users' account
- Strict referer checking
- Origin header (proposed in paper)

1.2 Cookies in the Cryptographer's Threat Model

Non-HTTPS cookies are broken for the cryptographer's threat model: interception implies full access. HTTPS can avoid these problems, but many think using HTTPS is too expensive. Possible middle-ground: crypto implemented in JavaScript.

2 The Same Origin Policy

The security policy web browsers implement for JavaScript.

- **Goal**: Isolate website from client's computer and other websites.
- origin = principal = `protocol://domain-name`
- JavaScript from one origin has *no access* to entities from other origins and *full access* to entities from same origin
  - even when embedding another website (e.g. `<iframe>`)

Exceptions to the same origin policy:

- `<script src=`*`URL`*`>` runs *URL* with the `<script>` tag's origin's privileges
- `<iframe>`'s URLs can be changed to/from other origins

3 Clickjacking

Demo: Mouse-clicking game causes user to give Flash permission to use the user's webcam and microphone.
Root cause: Web applications can draw outside their own input region

Clickjacking defenses:

- Websites opt-in to "don't put me in a frame" (IE 8)
  - Might be able to detect when in a frame portably
- Strict drawing region restrictions (but has backwards compatibility problems)

4 Keyjacking

Example attack (on old Firefox):

- Wait until user is about to press enter
- Ask to install extension; browser displays "Do you want to install . . . ?" defaulting to 'yes.'

Current Firefox's defense: wait 3 seconds before allowing yes option.

5 Frame navigation attacks

Frame navigation policies:

- version 1 — Any page/window can navigate any other
  - Example attack — navigate during/just before password entry
- version 2 — Any frame can navigate any frame within the same page
  - Example attack — malicious mashup (e.g. iGoogle) gadget phishes the password for another gadget
- version 3 (current) — Parent frames can navigate descendants