Session Attacks

Mendel Rosenblum

Session State

- Session state is used to control access in web servers

```
app.get(..., function (request, response) {
  if (request.session.login_name
```

- Typically derived from cookies in the request header

```
Cookie:connect.sid=s%3AckNzy0kByJYvsW5mR06ECGs1YXYojCXM.
VvNg0rI3rguSElNZNtdGrMBrDvbW4kvn641bqpcF4ec
Host: localhost:3000
```

- Consider what would happen if an attacker could guess or seal this cookie

Session Hijacking

Session Hijacking

- If an attacker can guess or steal the id associated with your session, he/she can impersonate you.
- Example: predictable session id
  - Server picks session id by incrementing a counter for each new session.
  - Attacker opens connection to server, gets session id.
  - Subtract 1 from session id: can hijack the previous session opened to the server.
- Solution: session ids must be unpredictable.
  - Don't build your own mechanism! Use something provided by your framework.
  - Rails: id = MD5(current time, random nonce)
  - Express Session: Uses module `uid-safe` - cryptographically secure UID (not predictable)
    Roll your own: `app.use(session({genid: function (request) { ….`

Need to use HTTPS to protect cookies

- Even if session id chosen carefully, network attackers can read cookies from unencrypted connections
  - Sessions not using HTTPS inherently vulnerable to network attacks.
- HTTP/HTTPS upgrade problem:
  - Suppose session starts out with HTTP, converts to HTTPS after login
  - Network attacker could have read session id during HTTP portion of session
  - Once logging is complete, attacker can use the id to hijack the logged in session
- Change the session id after any change in privilege or security level

Browser quirk involving cookies

- Cookies sent with all HTTP requests to our web server
  - Even if our app is not the current one being shown!
- Sometime want this: Consider a deep-linking bookmark or the back button
  - Desirable if it automatically picks up the session cookie
- Implication: Other sites/apps running concurrently can generate HTTP requests to our web servers!

Cross-Site Request Forgery (CSRF)

- Attackers can potentially hijack sessions without even knowing session ids:
  - Scenario:
    - Visit your bank's site, start up web app, log in.
    - Then visit the attacker's site (e.g. discussion forum with links, forms, etc.)
    - Attacker's page includes JavaScript that submits form to your bank.
    - When form gets submitted, browser includes bank's web app cookies, including the session id.
    - Bank transfers money to attacker's account.
    - The form can be in an iframe that is invisible, so you never know the attack occurred.
- This is called Cross-Site Request Forgery (CSRF) (Sea-surf)
  - Untrusted site uses trust that was given to user's browser

CSRF Defences

- CSRF was a big issue when frameworks used form submission for input
  - Ruby solution: server can mark forms that came from its pages
    - Every form must contain an additional authentication token as a hidden field
    - Server includes valid token in forms in pages that it generates (hidden form field).
    - Server checks token when form posted, rejects forms without proper token.
- JavaScript frameworks solutions
  - Don't accept POST submission directly from forms
    - Photo App: POST request have bodies of JSON strings
  - HTTP GET should not have side effects
    - Dangerous: Easy to trick the user into clicking on something
  - Have JavaScript include special HTTP request header property with secret
    - Module `csurf` - Adds `XSRF-TOKEN` to request headers

Data Tampering

- Server sends information to browser (cookies, HTML with links & forms)
  - Server can't trust what it gets back: User can view or modify anything provided by server
  - Examples:
    - Session information in cookies
    - CSRF defence (hidden form fields)
- Option #1: Server only uses information as a hint (must validate and correct)
  - Means we have a store all the information on server
- Option #2: Use cryptography to detect any tampering or forging
  - **Message Authentication Codes** (MACs)

Message Authentication Codes (MACs)

- MAC function takes arbitrary-length text, secret key, produces a MAC that provides a unique signature for the text.

  Think: Cryptography secure checksum

- Without knowing the secret key, cannot generate a valid MAC.
- Server includes MAC with data sent to the browser.
- Browser must return both MAC and data.
- Server can check the MAC using its secret key to detect tampering.

  Server checks input from browser and if MAC doesn't match tosses it (e.g. session cookie)

Using MACs in web servers

- MACs are useful if we need:
  - **Authentication** - Know that we (the web server) authored the information
  - **Integrity** - Known that it wasn't tampered with
- Need encryption if we want **confidentiality**
- If we need all three: encrypt then MAC
- Crypto APIs exist for doing these but somewhat of a pain to use