Securing Web Applications

Charlie Reis, Google

Writing Web Apps is Hard

- Tricky to make things work cross-browser
- AJAX adds client-side state, complexity
- **Securing against attacks is even harder**

Web Apps are Easy Targets

- Attackers can:
  - Send direct attacks to your server
  - Attack via another user's web browser

Outline

1. Client Side Logic and State

2. Cross Site Scripting

3. Cross Site Request Forgery

4. JavaScript Hijacking

1. Client-Side Logic & State

- **Do not trust the client, ever**
- Don’t trust the user
- Don’t trust the hardware (e.g., clock)
- Definitely don’t trust the client’s browser

Clients are Still Useful

- AJAX puts more content, logic, & state at client
- **Implications**:
  - Check any data from the user on the server
  - Know that any client-side check may be skipped
  - Be careful when exposing AJAX APIs on server

Outline

1. Client Side Logic and State

2. Cross Site Scripting

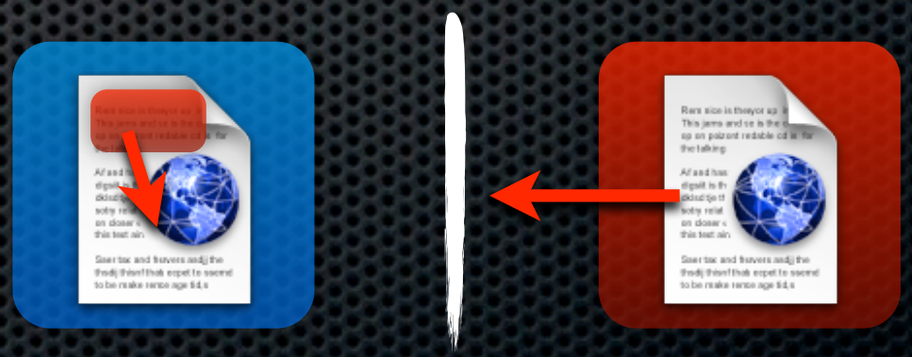
3. Cross Site Request Forgery

4. JavaScript Hijacking

2. Cross-Site Scripting (XSS)

- One of the most common web app vulnerabilities
- Web client security boils down to **Same Origin Policy**

- XSS gets around it
  - Attacker runs scripts inside your web page



XSS Consequences

- With XSS, attackers can:
  - **Steal cookies** (then take over user account)
  - **Phish users** (e.g., fake login dialog)
  - Control almost **all** content/code in entire origin

Types of XSS

- **Reflected XSS**
  - Some input echoed on the page (e.g., search query)
  - Victim must follow an evil link

http://app.com/query?<script>...</script>

- **Stored XSS**
  - Some input stored in a database (e.g., blog comment)
  - All later visitors become victims
- **DOM-Based XSS**
  - Client-side JS code echoes input into page

Defense: Data Sanitization

- Don’t ever trust user input
  - Escape all **output data** to prevent scripts
- Important: escaping must be **context sensitive**

foo<script>...</script>

- Don’t invent your own sanitizer. It will be wrong.
  - Use template systems or frameworks if possible

See: ClearSilver, Google CTemplate, GWT, etc

Outline

1. Client Side Logic and State

2. Cross Site Scripting

3. Cross Site Request Forgery

4. JavaScript Hijacking

3. Cross-Site Request Forgery

- Who's logged in? The user, or his browser?
- Pages in background can make requests to your site
  - Browser sends user's cookies
- Evil pages can act in user's name
  - Transfer funds, delete data, change state

Defense: Prevent Forging

- Make state-changing requests hard to guess
  - Avoid predictable URLs if possible
- **Tokens**: verify request comes from a page you trust
  - Put a hard-to-forge token in your page
  - Verify its presence in the request

Outline

1. Client Side Logic and State
2. Cross Site Scripting
3. Cross Site Request Forgery
4. **JavaScript Hijacking**

4. JavaScript Hijacking

- Many JS files have sensitive data (e.g., JSON)
- **Any web site can embed such files:**

```
<script src="mail.com/contacts">
```

- Can't *read* their contents, but can *run* them
- That's enough for attackers to get the data (e.g., Gmail contacts)

Defense: Don’t let data run

- Put something at the top to prevent execution
  - e.g., **while(1);** or unparseable characters
  - Remove before parsing the JSON
- Don’t use predictable URLs

Outline

1. Client Side Logic and State

2. Cross Site Scripting

3. Cross Site Request Forgery

4. JavaScript Hijacking

Ways Browsers Can Help

- **HTTP Only Cookies**: harder to get via XSS
- **Strict Transport Security**: only allow HTTPS
- **Reflective XSS Filters** (IE8, Chrome)

Summary

- Be wary of your users, think like an attacker
- Use security-minded frameworks whenever possible
  - Encapsulate the hard stuff

References & Tools

- **Foundations of Security: What Every Web Programmer Needs to Know**
  (Daswani, Kern, Kesavan)
- **Browser Security Handbook**
  http://code.google.com/p/browsersec/wiki/Main
- **RatProxy**
  http://code.google.com/p/ratproxy/