Web Security: Session management and CSRF

CS 161: Computer Security
Prof. Raluca Ada Popa

April 5, 2018

Credit: this deck is a combination of my slides and slide adaptations from previous offerings of this course and from CS 241 of Prof. Dan Boneh
Cookie policy versus same-origin policy
Cookie policy: when browser sends cookie

A cookie with
  domain = example.com, and
  path = /some/path/
will be included on a request to
  http://foo.example.com/some/path/subdirectory/hello.txt
Cookie policy versus same-origin policy

- Consider Javascript on a page loaded from a URL U
- If a cookie is in scope for a URL U, it can be accessed by Javascript loaded on the page with URL U, unless the cookie has the httpOnly flag set.
Examples

**cookie 1**
name = *userid*
value = *u1*
domain = *login.site.com*
path = /
non-secure

**cookie 2**
name = *userid*
value = *u2*
domain = *.site.com*
path = /
non-secure

`http://checkout.site.com/`  `cookie: userid=u2`
`http://login.site.com/`  `cookie: userid=u1, userid=u2`
`http://othersite.com/`  `cookie: none`

JS on each of these URLs can access all cookies that would be sent for that URL if the httpOnly flag is not set.
Indirectly bypassing same-origin policy using cookie policy

- Since the cookie policy and the same-origin policy are different, there are corner cases when one can use cookie policy to bypass same-origin policy
- Ideas how?
Victim user browser

Cookie domains:
- financial.example.com
- blog.example.com

cookie jar for *.example.com

financial.example.com web server

blog.example.com web server

(assume attacker compromised this web server)

Browsers maintain a separate cookie jar per domain group, such as one jar for *.example.com to avoid one domain filling up the jar and affecting another domain. Each browser decides at what granularity to group domains.
Example

Victim user browser

financial.example.com web server

GET

set-cookie:

blog.example.com web server

(assume attacker compromised this web server)

Attacker sets many cookies with domain example.com which overflows the cookie jar for domain *.example.com and overwrites cookies from financial.example.com.
Victim user browser

cookie jar for *.example.com

financial.example.com web server

blog.example.com web server
(assume attacker compromised this web server)

Attacker sets many cookies with domain example.com which overflows the cookie jar for domain *.example.com and overwrites cookies from financial.example.com
Example

Victim user browser

GET

financial.example.com web server

When Alice visits financial.example.com, the browser automatically attaches the attacker’s cookies due to cookie policy (the scope of the cookies is a domain suffix of financial.example.com)

Why is this a problem?

cookie jar for *.example.com
Indirectly bypassing same-origin policy using cookie policy

- Victim thus can login into attackers account at financial.example.com
- This is a problem because the victim might think its their account and might provide sensitive information
- This bypassed same-origin policy (indirectly) because blog.example.com influenced financial.example.com
RFC6265

- For further details on cookies, checkout the standard RFC6265 “HTTP State Management Mechanism”


- Browsers are expected to implement this reference, and any differences are browser specific
Session management
Sessions

- A sequence of requests and responses from one browser to one (or more) sites
  - Session can be long (Gmail - two weeks) or short

- without session mgmt:
  
users would have to constantly re-authenticate

- Session mgmt:
  - Authorize user once;
  - All subsequent requests are tied to user
Pre-history: HTTP auth

One username and password for a group of users

HTTP request: GET /index.html

HTTP response contains:

WWW-Authenticate: Basic realm="Password Required"

Browsers sends hashed password on all subsequent HTTP requests:

Authorization: Basic ZGFddfibzsdfgkjheczI1NXRleHQ=
HTTP auth problems

- Hardly used in commercial sites
  - User cannot log out other than by closing browser
    - What if user has multiple accounts?
    - What if multiple users on same computer?
  - Site cannot customize password dialog
  - Confusing dialog to users
  - Easily spoofed
Session tokens

Browser

GET /index.html

set anonymous session token

GET /books.html
anonymous session token

POST /do-login
Username & password

elevate to a logged-in session token

POST /checkout
logged-in session token

Web Site

check credentials (later)

Validate token
Storing session tokens:  
Lots of options (but none are perfect)

- Browser cookie:
  ```
  Set-Cookie: SessionToken=fduhye63sfdb
  ```

- Embedd in all URL links:
  ```
  https://site.com/checkout ? SessionToken=kh7y3b
  ```

- In a hidden form field:
  ```
  <input type="hidden" name="sessionid" value="kh7y3b">
  ```
Storing session tokens: problems

- Browser cookie:
  browser sends cookie with every request, even when it should not (CSRF)

- Embed in all URL links:
  token leaks via HTTP Referer header
  users might share URLs

- In a hidden form field: short sessions only

Better answer: a combination of all of the above (e.g., browser cookie with CSRF protection using form secret tokens)
Cross Site Request Forgery
What Changed From 2010 to 2013?

The threat landscape for applications security constantly changes. Key factors in this evolution are advances made by attackers. We broadened Failure to Restrict URL Access from the 2010 OWASP Top 10 to be more inclusive: this is because CSRF has been in the OWASP Top 10 for 6 years, and organizations and framework developers have focused on it enough to significantly reduce the number of CSRF vulnerabilities in real world applications.

Insecure Direct Object References moved up in prevalence based on our data set. We believe this is probably due to the increase in use of RESTful APIs and the growth and depth of component based development which has significantly increased the risk of using known vulnerable components.

Site Request Forgery (CSRF) moved down in prevalence based on our data set from 2010 to 2013. To keep pace, we periodically update the OWASP Top 10. In this 2013 release, we made the following changes:

- We merged and broadened Security Misconfiguration into new Using Known Vulnerable Components.
- Site Request Forgery (CSRF) moved down in prevalence based on our data set from 2010 to 2013.
- Broken Authentication and Session Management moved up in prevalence based on our data set. We believe this is probably due to the increase in use of RESTful APIs and the growth and depth of component based development which has significantly increased the risk of using known vulnerable components.

**Top web vulnerabilities**

<table>
<thead>
<tr>
<th>OWASP Top 10 – 2010 (Previous)</th>
<th>OWASP Top 10 – 2013 (New)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 – Injection</td>
<td>A1 – Injection</td>
</tr>
<tr>
<td>A3 – Broken Authentication and Session Management</td>
<td>A2 – Broken Authentication and Session Management</td>
</tr>
<tr>
<td>A2 – Cross-Site Scripting (XSS)</td>
<td>A3 – Cross-Site Scripting (XSS)</td>
</tr>
<tr>
<td>A4 – Insecure Direct Object References</td>
<td>A4 – Insecure Direct Object References</td>
</tr>
<tr>
<td>A6 – Security Misconfiguration</td>
<td>A5 – Security Misconfiguration</td>
</tr>
<tr>
<td>A7 – Insecure Cryptographic Storage – Merged with A9</td>
<td>A6 – Sensitive Data Exposure</td>
</tr>
<tr>
<td>A8 – Failure to Restrict (IP) access – Broadened into</td>
<td>A7 – Missing Function Level Access Control</td>
</tr>
<tr>
<td><strong>A5 – Cross-Site Request Forgery (CSRF)</strong></td>
<td><strong>A8 – Cross-Site Request Forgery (CSRF)</strong></td>
</tr>
<tr>
<td>&lt;buried in A6: Security Misconfiguration&gt;</td>
<td>A9 – Using Known Vulnerable Components</td>
</tr>
</tbody>
</table>
HTML Forms

Allow a user to provide some data which gets sent with an HTTP POST request to a server

<form action="bank.com/action.php">
  First name:  <input type="text" name="firstname">
  Last name: <input type="text" name="lastname">
  <input type="submit" value="Submit"></form>

When filling in Alice and Smith, and clicking submit, the browser issues HTTP POST request bank.com/action.php?firstname=Alice&lastname=Smith

As always, the browser attaches relevant cookies
Consider cookie storing session token

- Server assigns a session token to each user after they logged in, places it in the cookie
- The server keeps a table of username to current session token, so when it sees the session token it knows which user
Session using cookies

Browser

POST/login.cgi

Server

Set-cookie: session token

GET/POST...
Cookie: session token

response
Basic picture

1. Establish session
2. Visit server
3. Receive malicious page
4. Send forged request (w/ cookie)

What can go bad? URL contains transaction action

User Victim
- Cookie for bank.com with session token

Server Victim bank.com

Attack Server
Cross Site Request Forgery (CSRF)

Example:
- User logs in to bank.com
  - Session cookie remains in browser state
- User visits malicious site containing:
  ```html
  <form name=F action=http://bank.com/BillPay.php>
  <input name=recipient value=badguy> ...
  <script> document.F.submit(); </script>
  ```
- Browser sends user auth cookie with request
  - Transaction will be fulfilled

Problem:
- cookie auth is insufficient when side effects occur
Form post with cookie

www.attacker.com

GET /blog HTTP/1.1

Victim Browser

www.bank.com
Form post with cookie

GET /blog HTTP/1.1

POST /transfer HTTP/1.1
Referer: http://www.attacker.com/blog
Recipient=attacker&amount=$100
Cookie: SessionID=523FA4cd2E

HTTP/1.1 200 OK
Transfer complete!
Squigler demo
An attacker could
- add videos to a user’s "Favorites,"
- add himself to a user’s "Friend" or "Family" list,
- send arbitrary messages on the user’s behalf,
- flagged videos as inappropriate,
- automatically shared a video with a user’s contacts, subscribed a user to a "channel" (a set of videos published by one person or group), and
- added videos to a user’s "QuickList" (a list of videos a user intends to watch at a later point).
Popular websites fall victim to CSRF exploits
Defenses ideas?
CSRF Defenses

- CSRF token

- Referer Validation

- Others (e.g., custom HTTP Header) we won’t go into
CSRF token

1. goodsite.com server wants to protect itself, so it includes a secret token into the webpage (e.g., in forms as a hidden field)
2. Requests to goodsite.com include the secret
3. goodsite.com server checks that the token embedded in the webpage is the expected one; reject request if not

Can the token be?

- 123456
- Dateofbirth

CSRF token must be hard to guess by the attacker
How token is used

- The server stores state that binds the user's CSRF token to the user's session id
- Embeds CSRF token in every form
- On every request the server validates that the supplied CSRF token is associated with the user's session id
- Disadvantage is that the server needs to maintain a large state table to validate the tokens.
Other CRSF protection: Referer Validation

- When the browser issues an HTTP request, it includes a referer header that indicates which URL initiated the request.
- This information in the Referer header could be used to distinguish between same site request and cross site request.
Referer Validation

Facebook Login

For your security, never enter your Facebook password on sites not located on Facebook.com.

Email: 
Password: 

Remember me
Login or Sign up for Facebook

Forgot your password?
Referer Validation Defense

- HTTP Referer header
  - Referer: http://www.facebook.com/
  - Referer: http://www.attacker.com/evil.html
  - Referer:
    - Strict policy disallows (secure, less usable)
    - Lenient policy allows (less secure, more usable)
Privacy Issues with Referer header

- The referer contains sensitive information that impinges on the privacy.
- The referer header reveals contents of the search query that lead to visit a website.
- Some organizations are concerned that confidential information about their corporate intranet might leak to external websites via Referer header.
Referer Privacy Problems

- Referer may leak privacy-sensitive information

- Common sources of blocking:
  - Network stripping by the organization
  - Network stripping by local machine
  - Stripped by browser for HTTPS -> HTTP transitions
  - User preference in browser
Cookies add state to HTTP
- Cookies are used for session management
- They are attached by the browser automatically to HTTP requests

CSRF attacks execute request on benign site because cookie is sent automatically

Defenses for CSRF:
- embed unpredicatable token and check it later
- check referer header