Web Security:
Session management

CS 161: Computer Security

Prof. Raluca Ada Popa

April 4, 2019

Some content adapted from materials by David Wagner or Dan Boneh
Announcements

• Midterm 2: Apr 9, 8pm - 10pm
• Covers up to the material this week
• Review session: April 4th from 6-8pm in Soda 306
• I’m offering extra office hours today, 5-6pm, Soda 729
Cookies

- A way of maintaining state

Browser maintains cookie jar with all cookies it receives

```
Browser
```

GET ...

```
http response contains
```

```
Server
```
Setting/deleting cookies by server

The first time a browser connects to a particular web server, it has no cookies for that web server.

When the web server responds, it includes a **Set-Cookie** header that defines a cookie.

Each cookie is just a name-value pair (with some extra metadata).
View a cookie

In a web console (firefox, tool->web developer->web console),
type

    document.cookie

to see the cookie for that site
Example Gmail cookies

The output of document.cookie in the web console for my Gmail (redacted to remove my real SIDs):

"GMAIL_AT=mslgjadgdga3qwfqad34fwerfxacRSA;
CONSENT=YES+CH.en-GB+V9;
SID=askfjw448qufierhixcnihfnxqkhfaflkhnzk33;
APISID=4oq58tkjexqac;
SAPISID=345qxqa;
1P_JAR=2019-04-04-06;
SIDCC=lgact3etmfxa4q3gcgemam"

Each name=value is one cookie. document.cookie lists all cookies in scope for document.
Cookie scope

When the browser connects to the same server later, it automatically attaches the cookies in scope: header containing the name and value, which the server can use to connect related requests.

Domain and path inform the browser about which sites to send this cookie to.
HTTP Header:

```text
Set-cookie: NAME=VALUE ;
domain = (when to send) ;
path = (when to send) ;
secure = (only send over HTTPS);
```

- **Secure**: sent over https only
  - https provides secure communication using TLS (privacy and integrity)
Cookie scope

GET ...

HTTP Header:
Set-cookie: NAME=VALUE ;
  domain = (when to send) ;
  path = (when to send)
  secure = (only send over SSL);
  expires = (when expires) ;
  HttpOnly

- Expires is expiration date
  - Delete cookie by setting “expires” to date in past
- HttpOnly: cookie cannot be accessed by Javascript, but only sent by browser
Cookie scope

• Scope of cookie might not be the same as the URL-host name of the web server setting it

The cookie policy has two parts:
1. What scopes a URL-host name web server is allowed to set on a cookie
2. When the browser sends a cookie to a URL
What scope a server may set for a cookie

The browser checks if the web server may set the cookie, and if not, it will not accept the cookie.

domain: any domain-suffix of URL-hostname, except TLD

e.g. `login.site.com` can set cookies for all of `.site.com` but not for another site or TLD

<table>
<thead>
<tr>
<th>allowed domains</th>
<th>disallowed domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>login.site.com</td>
<td>user.site.com</td>
</tr>
<tr>
<td>.site.com</td>
<td>othersite.com</td>
</tr>
</tbody>
</table>

⇒ `login.site.com` can set cookies for all of `.site.com` but not for another site or TLD

Problematic for sites like `.berkeley.edu`

path: can be set to anything
We discussed the semantics of HTTP cookies in Chapter 3, but that discussion left out one important detail: the security rules that must be implemented to protect cookies belonging to one site from being tampered with by unrelated pages. This topic is particularly interesting because the approach taken here predates the same-origin policy and interacts with it in a number of unexpected ways.

Cookies are meant to be scoped to domains, and they can't be limited easily to just a single hostname value. The domain parameter provided with an cookie may simply match the current hostname (such as foo.example.com), but this will not prevent the cookie from being sent to any eventual subdomains, such as bar.foo.example.com. A qualified right-hand fragment of the hostname, such as example.com, can be specified to request a broader scope, however.

Amusingly, the original RFCs imply that Netscape engineers wanted to allow exact host-scoped cookies, but they did not follow their own advice. The syntax devised for this purpose was not recognized by the descendants of Netscape Navigator (or by any other implementation for that matter). To a limited extent, setting host-scoped cookies is possible in some browsers by completely omitting the domain parameter, but this method will have no effect in Internet Explorer.

Table 9-3 illustrates cookie-setting behavior in some distinctive cases.

The only other true cookie-scoping parameter is the path prefix: Any cookie can be set with a specified path value. This instructs the browser to send the cookie back only on requests to matching directories; a cookie scoped to domain of example.com and path of /some/path/ will be included on a request to http://foo.example.com/some/path/subdirectory/hello_world.txt.

This mechanism can be deceptive. URL paths are not taken into account during same-origin policy checks and, therefore, do not form a useful security boundary. Regardless of how cookies work, JavaScript code can simply hop between any URLs on a single host at will and inject malicious payloads into...
Content Isolation Logic

149

Security Policy for Cookies

We discussed the semantics of HTTP cookies in Chapter 3, but that discussion left out one important detail: the security rules that must be implemented to protect cookies belonging to one site from being tampered with by unrelated pages. This topic is particularly interesting because the approach taken here predates the same-origin policy and interacts with it in a number of unexpected ways.

Cookies are meant to be scoped to domains, and they can't be limited easily to just a single hostname value. The domain parameter provided with a cookie may simply match the current hostname (such as foo.example.com), but this will not prevent the cookie from being sent to any eventual subdomains, such as bar.foo.example.com. A qualified right-hand fragment of the hostname, such as example.com, can be specified to request a broader scope, however.

Amusingly, the original RFCs imply that Netscape engineers wanted to allow exact host-scoped cookies, but they did not follow their own advice. The syntax devised for this purpose was not recognized by the descendants of Netscape Navigator (or by any other implementation for that matter). To a limited extent, setting host-scoped cookies is possible in some browsers by completely omitting the domain parameter, but this method will have no effect in Internet Explorer.

Table 9-3 illustrates cookie-setting behavior in some distinctive cases.

The only other true cookie-scoping parameter is the path prefix: Any cookie can be set with a specified path value. This instructs the browser to send the cookie back only on requests to matching directories; a cookie scoped to domain of example.com and path of /some/path/ will be included on a request to http://foo.example.com/some/path/subdirectory/hello_world.txt.

This mechanism can be deceptive. URL paths are not taken into account during same-origin policy checks and, therefore, do not form a useful security boundary. Regardless of how cookies work, JavaScript code can simply hop between any URLs on a single host at will and inject malicious payloads into

Table 9-3: A Sample of Cookie-Setting Behaviors

<table>
<thead>
<tr>
<th>domain</th>
<th>Whether it will be set, and if so, where it will be sent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>(value omitted)</td>
<td>foo.example.com (exact)</td>
</tr>
<tr>
<td>bar.foo.example.com</td>
<td>Cookie not set: domain more specific than origin</td>
</tr>
<tr>
<td>foo.example.com</td>
<td>*.foo.example.com</td>
</tr>
<tr>
<td>baz.example.com</td>
<td>Cookie not set: domain mismatch</td>
</tr>
<tr>
<td>example.com</td>
<td>*.example.com</td>
</tr>
<tr>
<td>ample.com</td>
<td>Cookie not set: domain mismatch</td>
</tr>
<tr>
<td>.com</td>
<td>Cookie not set: domain too broad, security risk</td>
</tr>
</tbody>
</table>
When browser sends cookie

GET //URL-domain/URL-path
Cookie: NAME = VALUE

Goal: server only sees cookies in its scope

Browser sends all cookies in URL scope:
• cookie-domain is domain-suffix of URL-domain, and
• cookie-path is prefix of URL-path, and
• [protocol=HTTPS if cookie is “secure”]
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
Examples: Which cookie will be sent?

**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
```
Examples

**cookie 1**
- name = userid
- value = u1
- domain = login.site.com
- path = /
- 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=u2

https://login.site.com/
cookie: userid=u1; userid=u2
(arbitrary order)
Client side read/write:  document.cookie

• Setting a cookie in Javascript:
  ```javascript
  document.cookie = "name=value; expires=...; "
  ```

• Reading a cookie:
  ```javascript
  alert(document.cookie)
  ```
  prints string containing all cookies available for document (based on [protocol], domain, path)

• Deleting a cookie:
  ```javascript
  document.cookie = "name=; expires= Thu, 01-Jan-70"
  ```

document.cookie often used to customize page in Javascript
Viewing/deleting cookies in Browser UI

Firefox: Tools -> page info -> security -> view cookies
Cookie policy versus same-origin policy
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 hold group domains.
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

financial.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
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?
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 (banks)
  – without session mgmt:
    users would have to constantly re-authenticate

• Session management:
  – 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

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

- Browser cookie:
  Set-Cookie: SessionToken=fduhye63sfdb

- Embed 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 the above (e.g., browser cookie with CSRF protection using form secret tokens)
Random fact about … Pieter Abbeel

“I enjoyed acting in this Verizon commercial (aired nationally over 500 times :). ”

[https://vimeo.com/259366281]
Cross Site Request Forgery
HTML Forms

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

```html
<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 the cookie stores the 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

Server

POST/login.cgi

Set-cookie: session token

GET/POST...

Cookie: session token

response
Basic picture

1. establish session
2. visit server
3. receive malicious page with session token
4. send forged request (w/ cookie)

What can go bad? URL contains transaction action
Cross Site Request Forgery (CSRF)

- 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>
  </form>
  ```

- 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 -> Victim Browser
GET /blog HTTP/1.1

www.bank.com
Form post with cookie

User credentials

Cookie: SessionID=523FA4cd2E
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).
Facebook Hit by Cross-Site Request Forgery Attack

By Sean Michael Kerner | August 20, 2009

September 30, 2008

Popular websites fall victim to CSRF exploits
Defenses
ideas?
CSRF Defenses

• CSRF token

<\texttt{input type=hidden value=23a3af01b}> Referer: http://www.facebook.com/home.php

• Referer Validation

Referer: http://www.facebook.com/home.php

• Others (e.g., custom HTTP Header) we won’t go into
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 the 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: [empty] 💎
    • 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

   http://intranet.corp.apple.com/
   projects/iphone/competitors.html

• 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
Summary: sessions and CSRF

- 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