Web Security: Vulnerabilities & Attacks

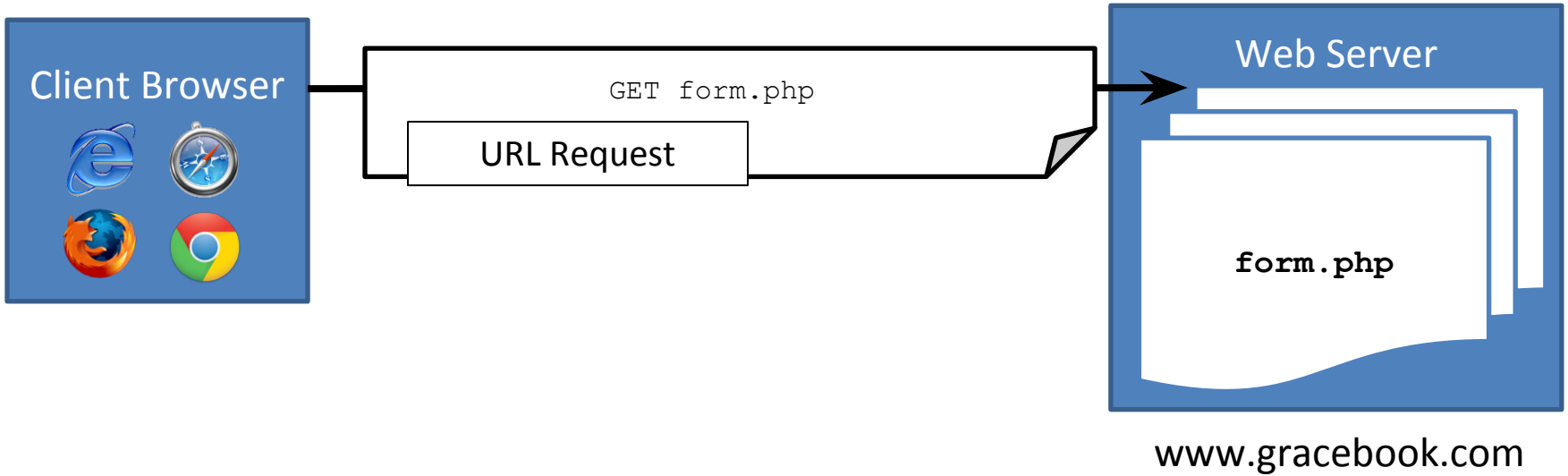
Cross-site Request Forgery

Example Application

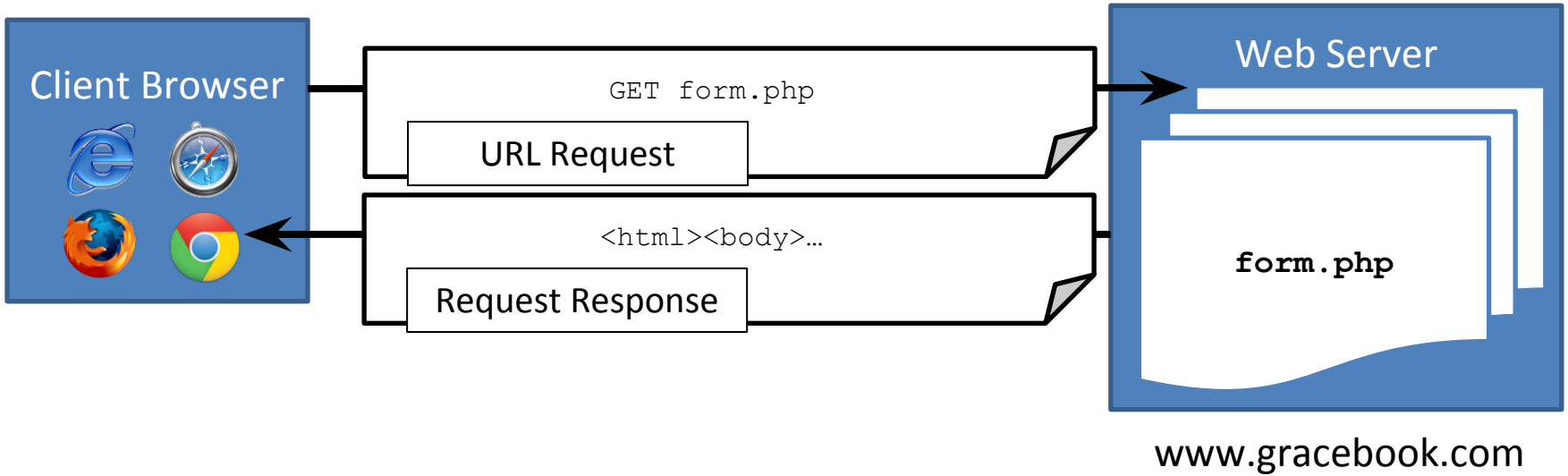
Consider a social networking site, GraceBook, that allows users to ‘share’ happenings from around the web. Users can click the “Share with GraceBook” button which publishes content to GraceBook.

When users press the share button, a `POST` request to http://www.gracebook.com/share.php is made and gracebook.com makes the necessary updates on the server.

Running Example



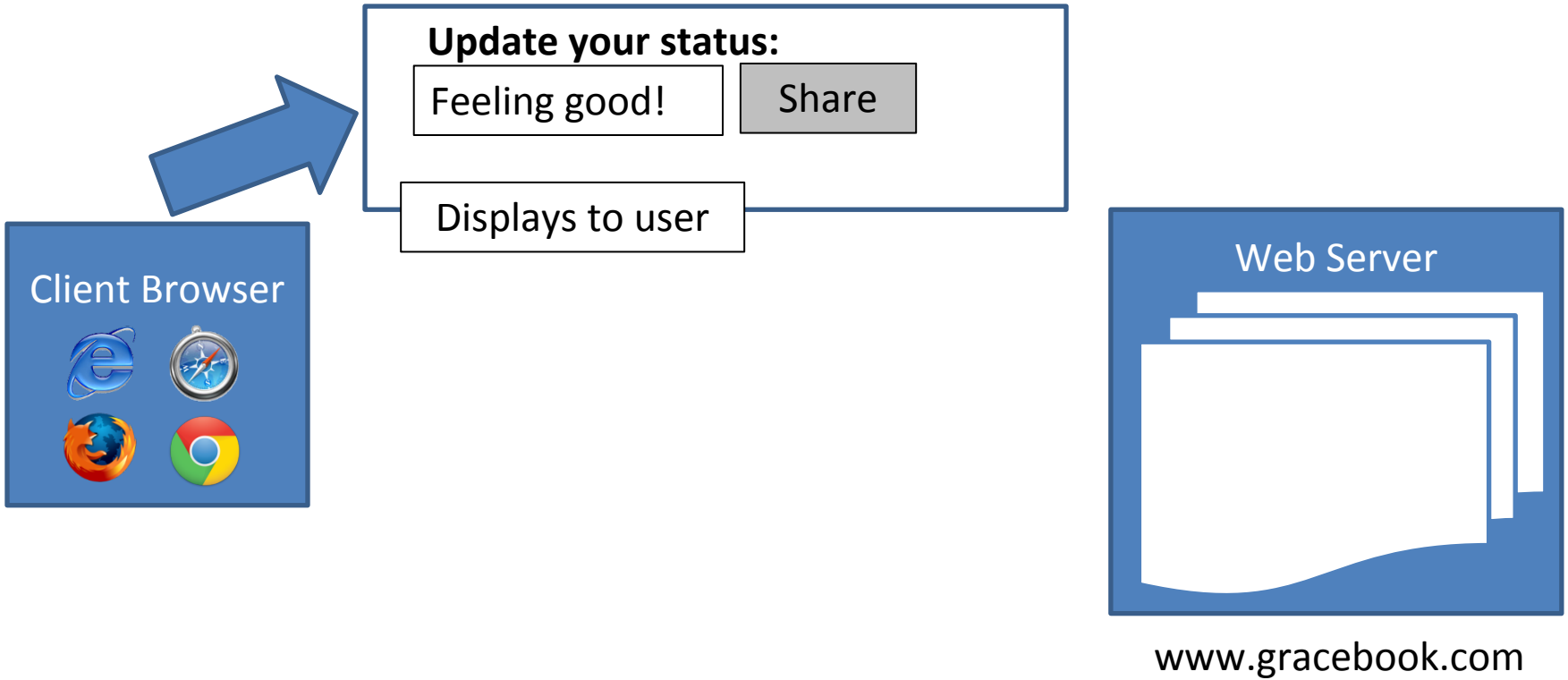
Running Example



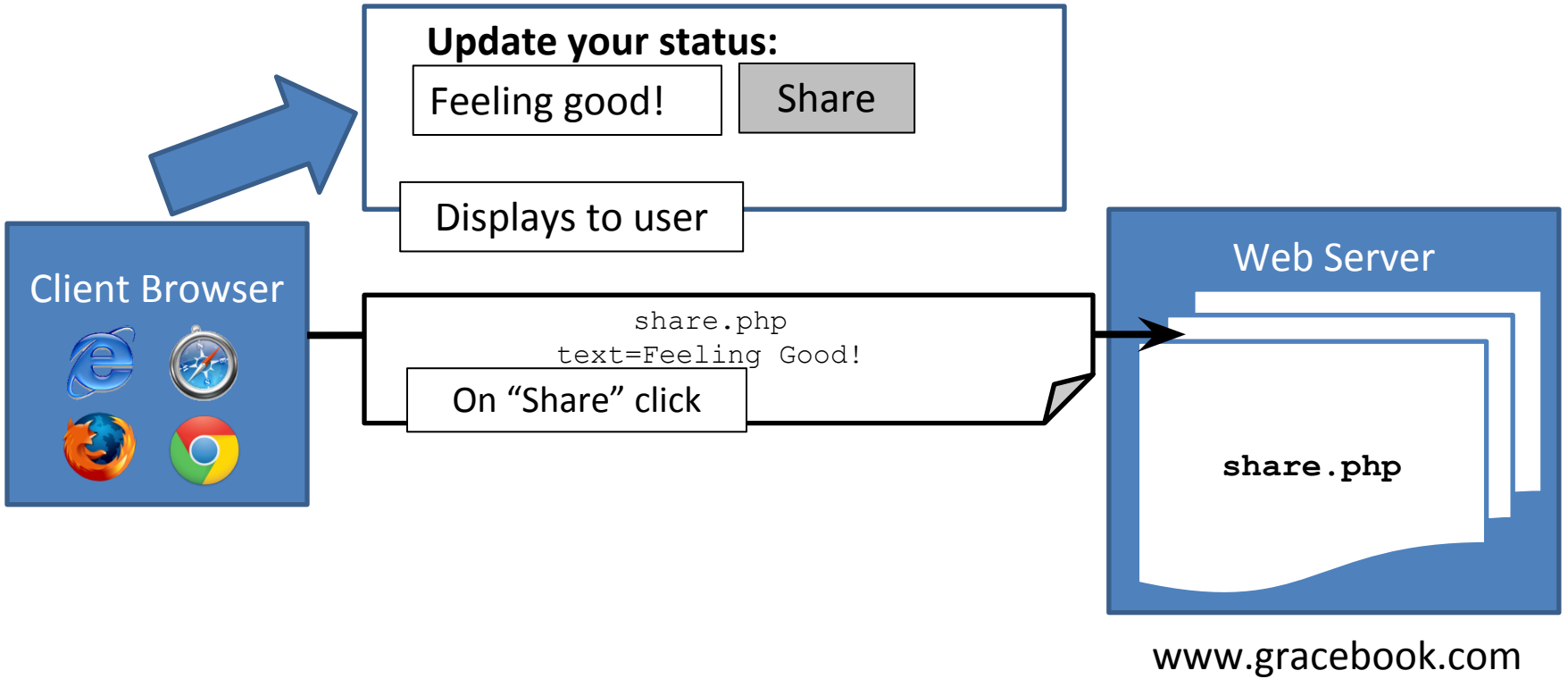
Running Example

```
<html><body>
<div>
Update your status:

<form action="http://www.gracebook.com/share.php" method="post">
<input name="text" value="Feeling good!"></input>
<input type="submit" value="Share"></input>
</form>
</div>
</body></html>
```

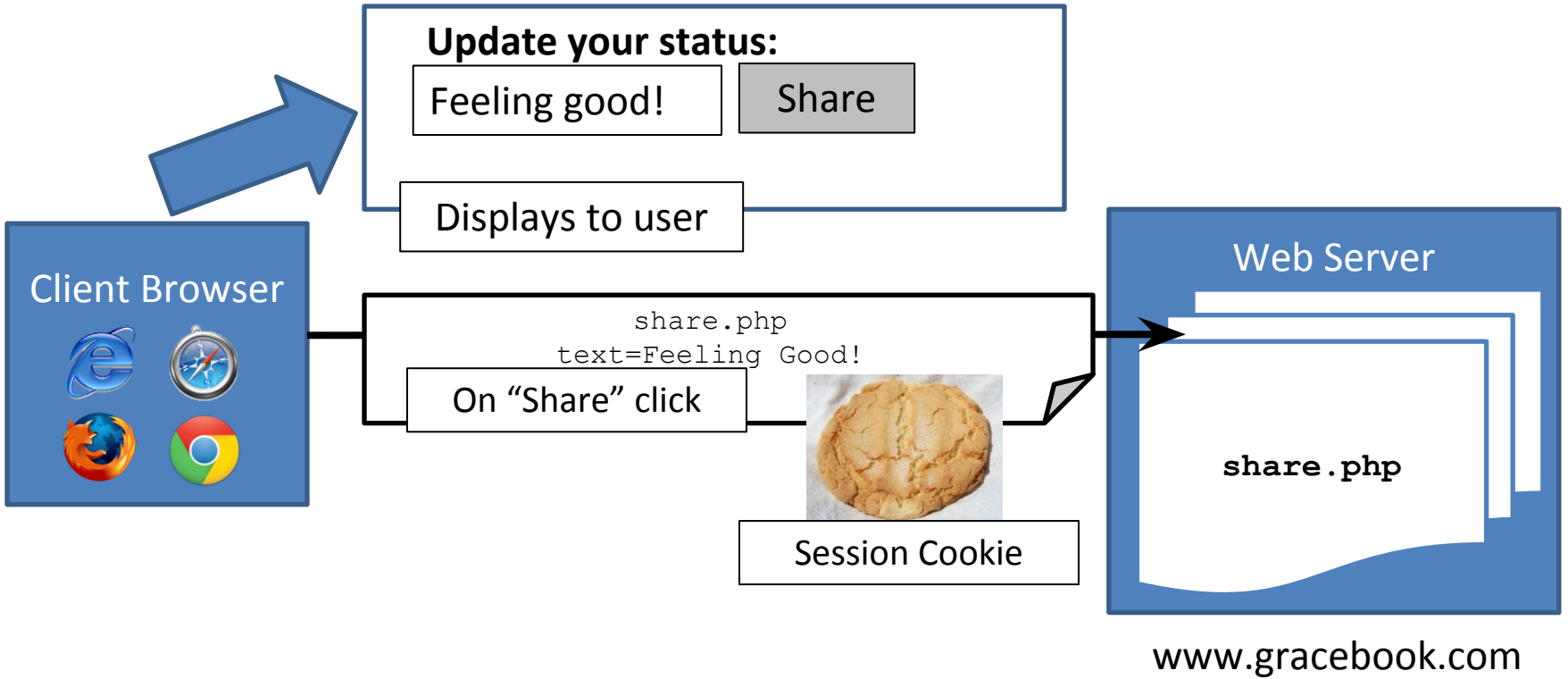
Running Example



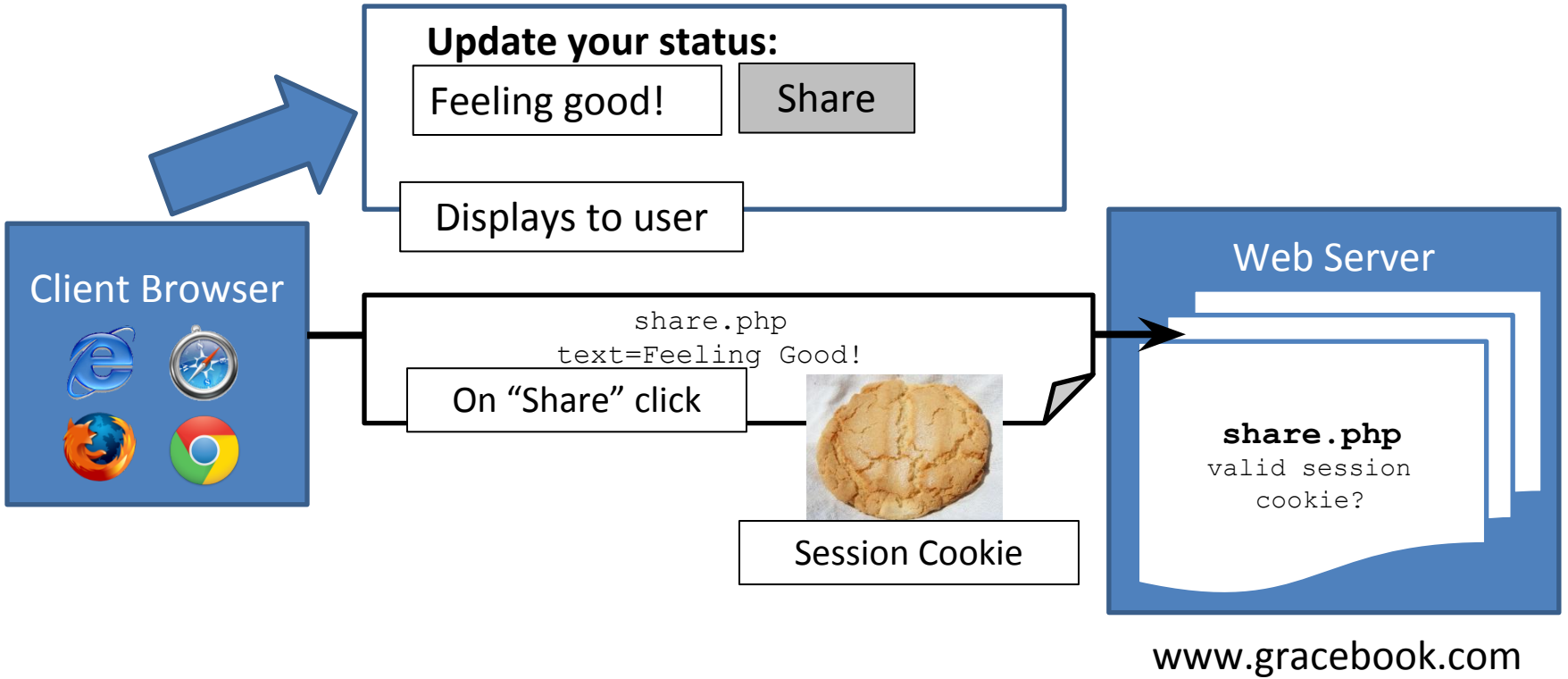
Running Example



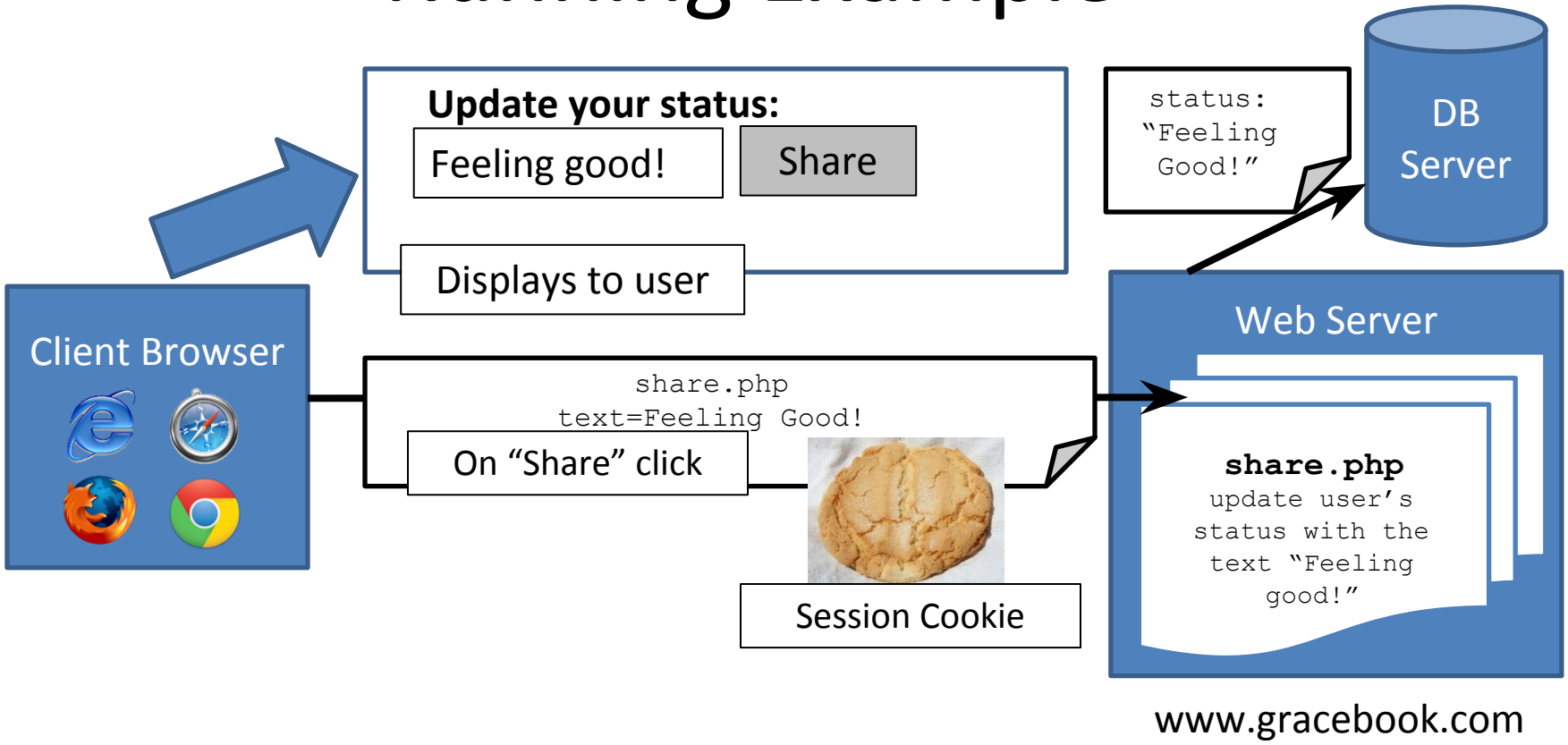
Running Example



Running Example



Running Example



Network Requests

The HTTP `POST` Request looks like this:

```
POST /share.php HTTP/1.1
Host: www.gracebook.com
User-Agent: Mozilla/5.0
Accept: */*
Content-Type: application/x-www-form-urlencoded;
charset=UTF-8
Referer:
   https://www.gracebook.com/form.php
Cookie: auth=beb18dcd75f2c225a9dcd71c73a8d77b5c304fb8

text=Feeling good!
```

CSRF Attack

- The attacker, on `attacker.com`, creates a page containing the following HTML:

```
<form action="http://www.gracebook.com/share.php" method="post" id="
f">
<input type="hidden" name="text" value="SPAM COMMENT"></input>
<script>document.getElementById('f').submit();</script>
```

- What will happen when the user visits the page?

a) The spam comment will be posted to user's share feed on gracebook.com

b) The spam comment will be posted to user's share feed if the user is currently logged in on gracebook.com

c) The spam comment will not be posted to user's share feed on gracebook.com

CSRF Attack

- The attacker, on `attacker.com`, creates a page containing the following HTML:

```
<form action="http://www.gracebook.com/share.php" method="post" id="
f">
<input type="hidden" name="text" value="SPAM COMMENT"></input>
<script>document.getElementById('f').submit();</script>
```

- What will happen when the user visits the page?

a) The spam comment will be posted to user's share feed on gracebook.com

b) The spam comment will be posted to user's share feed if the user is currently logged in on gracebook.com

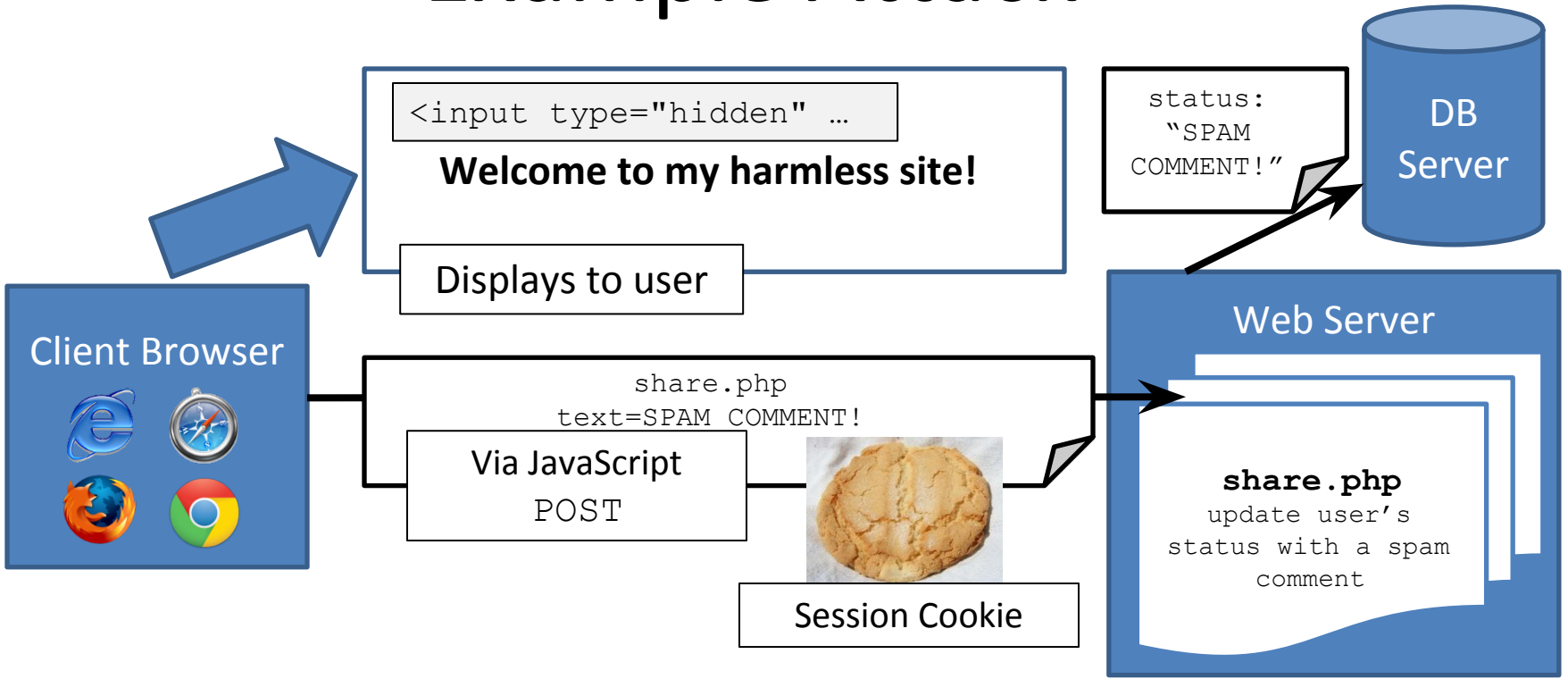
c) The spam comment will not be posted to user's share feed on gracebook.com

CSRF Attack

- JavaScript code can automatically submit the form in the background to post spam to the user's GraceBook feed.
- Similarly, a `GET` based CSRF is also possible. Making `GET` requests is easier: just an `img` tag suffices.

```
<img src="http://www.gracebook.com/share.php?text=SPAM%20COMMENT" />
```

Example Attack



CSRF Defense

- Origin headers
  - Introduction of a new header, similar to Referer.
  - Unlike Referer, only shows scheme, host, and port (no path data or query string)
- Nonce-based
  - Use a nonce to ensure that only `form.php` can get to `share.php`.

CSRF via POST requests

Consider the Referer value from the `POST` request outlined earlier. In the case of the CSRF attacks, will it be different?

a. `Yes`
b. `No`

CSRF via POST requests

Consider the Referer value from the `POST` request outlined earlier. In the case of the CSRF attacks, will it be different?

a. `Yes`
b. `No`

Origin Header

- Instead of sending whole referring URL, which might leak private information, only send the referring scheme, host, and port.

```
POST /share.php HTTP/1.1
Host: www.gracebook.com
User-Agent: Mozilla/5.0
Accept: */*
Content-Type: application/x-www-form-urlencoded;
charset=UTF-8
Origin: http://www.gracebook.com/
Cookie: auth=beb18dcd75f2c225a9dcd71c73a8d77b5c304fb8

text=hi
```

Origin Header

- Instead of sending whole referring URL, which might leak private information, only send the referring scheme, host, and port.

```
POST /share.php HTTP/1.1
Host: www.gracebook.com
User-Agent: Mozilla/5.0
Accept: */*
Content-Type: application/x-www-form-urlencoded;
charset=UTF-8
Origin: http://www.gracebook.com/
Cookie: auth=beb18dcd75f2c225a9dcd71c73a8d77b5c304fb8

text=hi
```

No path string or query data

Nonce based protection

- Recall the expected flow of the application:
  - The message to be shared is first shown to the user on `form.php` (the `GET` request)
  - When user assents, a `POST` request to `share.php` makes the actual post
- The server creates a nonce, includes it in a hidden field in `form.php` and checks it in `share.php`.

Nonce based protection

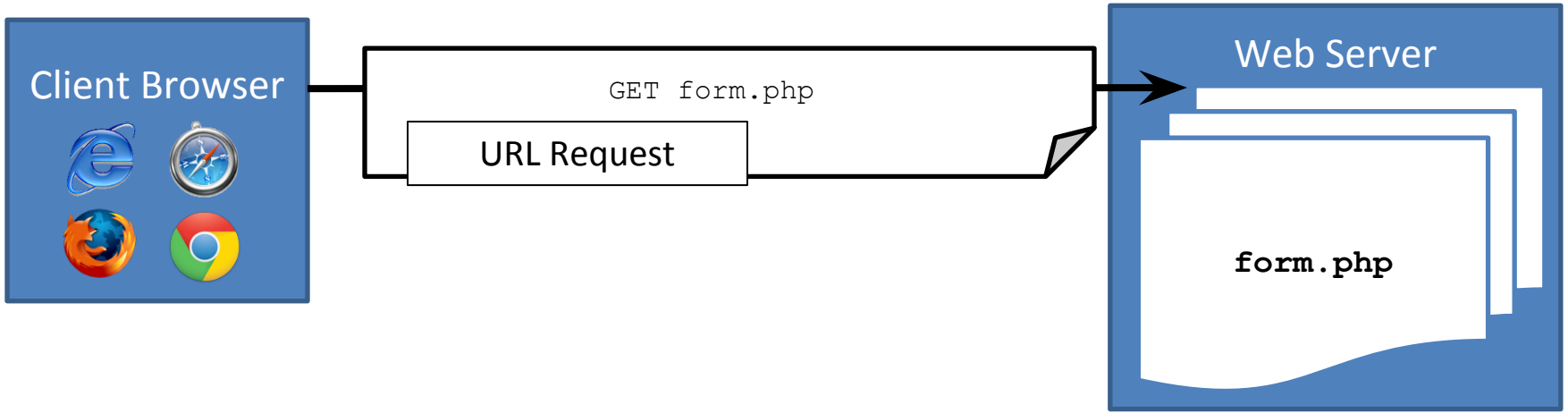
The form with nonce

```
<form action="share.php" method="post">
<input type="hidden" name="csrfnonce" value="av834favcb623">
<input type="textarea" name="text" value="Feeling good!">
```

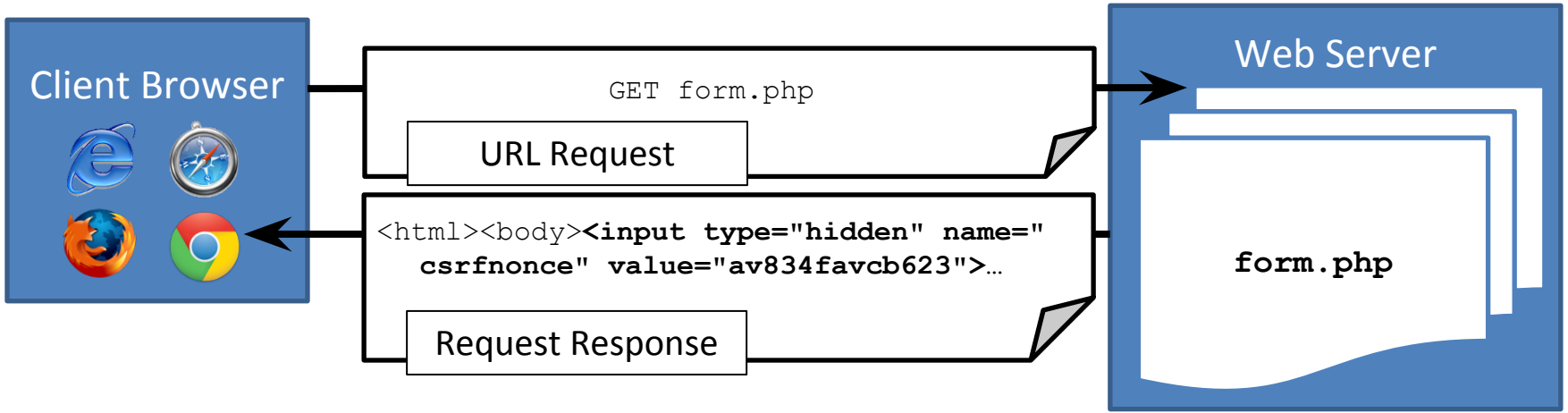
```
POST /share.php HTTP/1.1
Host: www.gracebook.com
User-Agent: Mozilla/5.0
Accept: */*
Content-Type: application/x-www-form-urlencoded;
charset=UTF-8
Origin: http://www.gracebook.com/
Cookie: auth=beb18dcd75f2c225a9dcd71c73a8d77b5c304fb8

Text=Feeling good!&csrfnonce=av834favcb623
```

Server code compares nonce

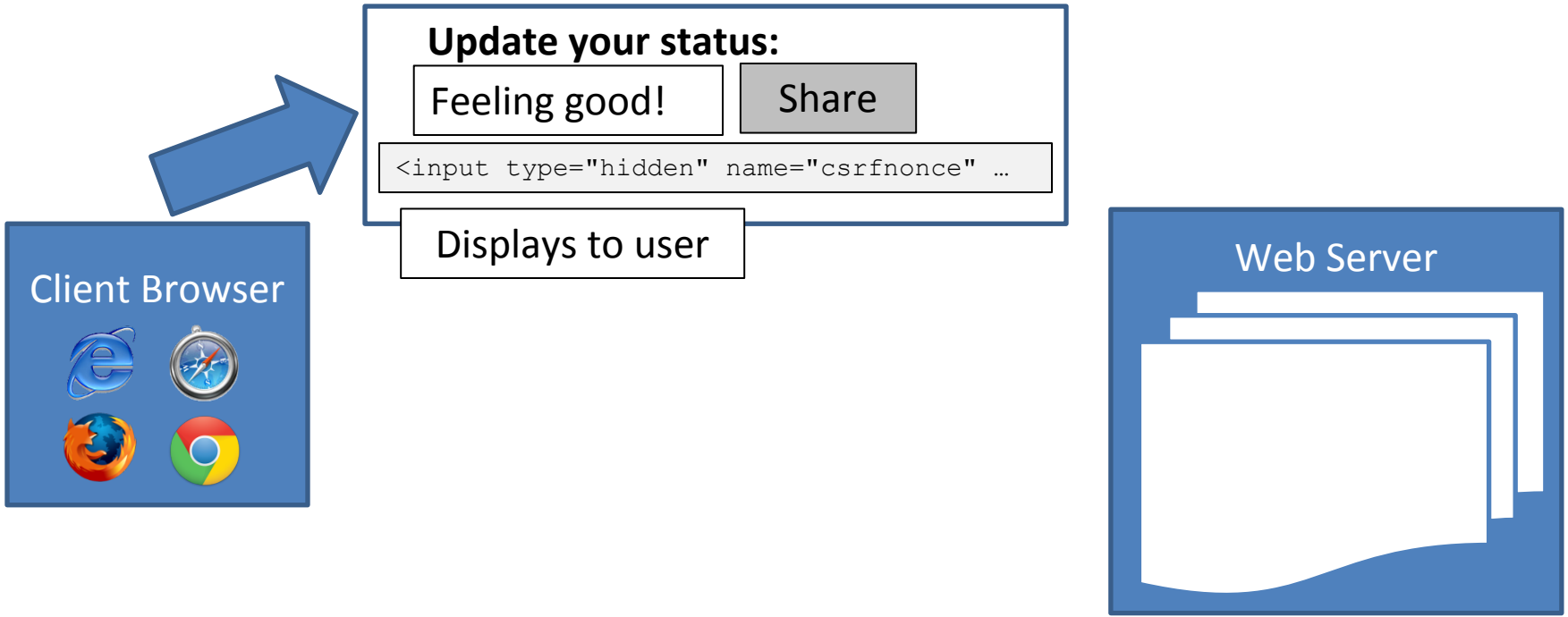
Legitimate Case



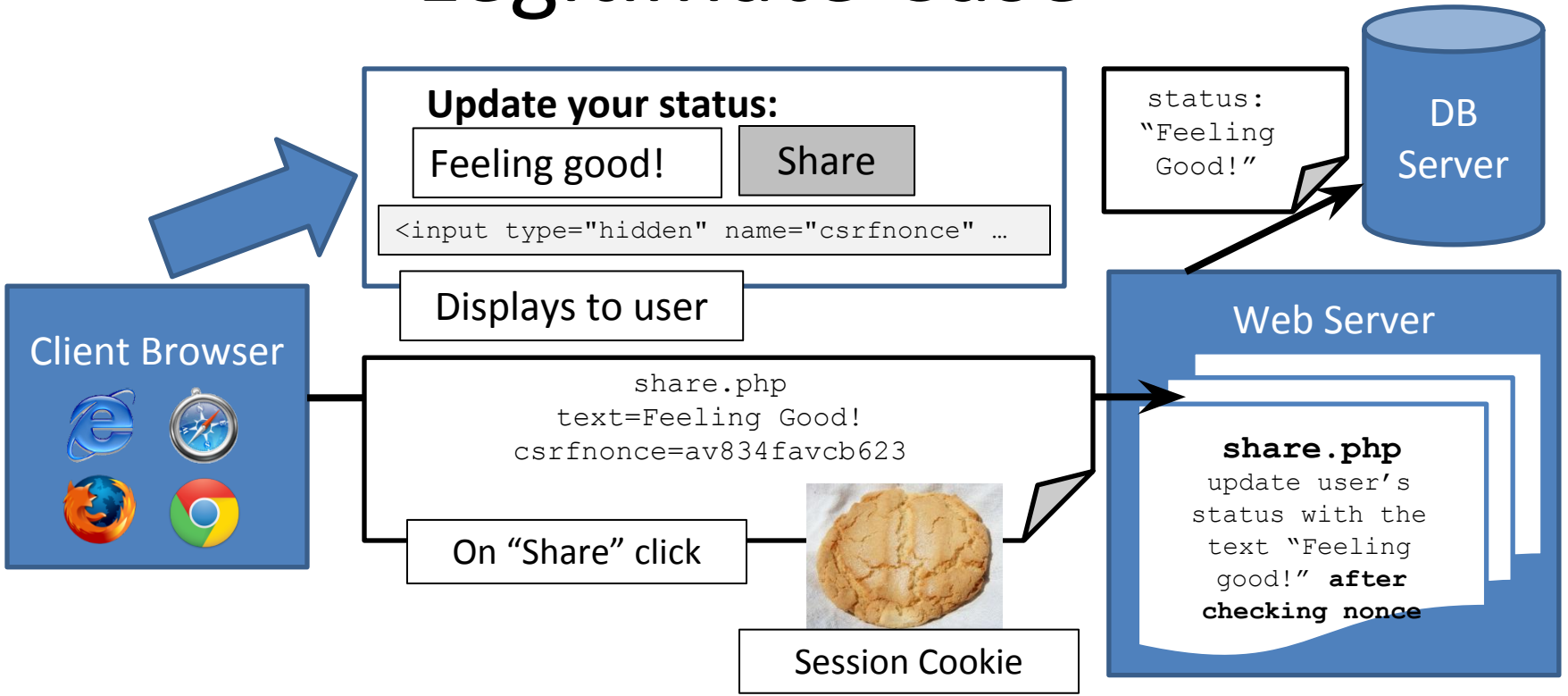
Legitimate Case



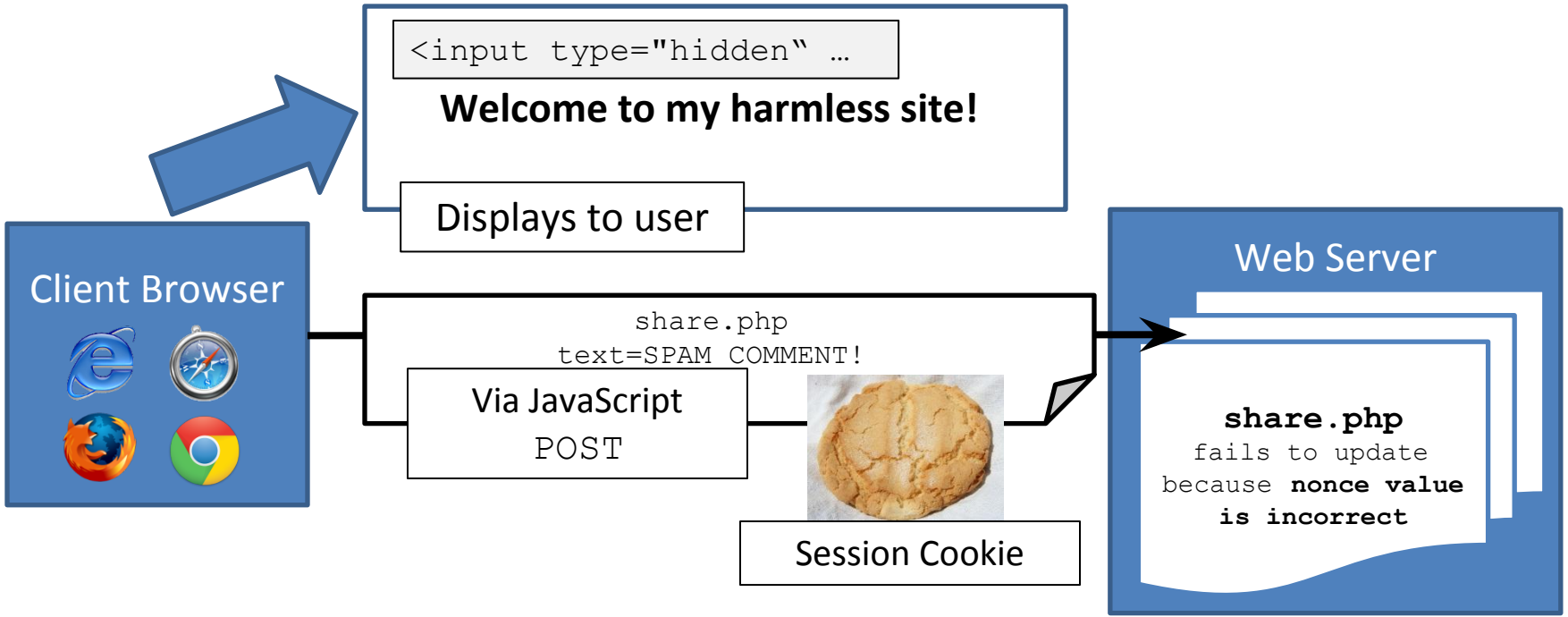
Legitimate Case



Legitimate Case



Attack Case


<input type="hidden" …
Welcome to my harmless site!
Displays to user
Client Browser
share.php
text=SPAM COMMENT!
Via JavaScript
POST
Session Cookie
Web Server
share.php
fails to update
because nonce value
is incorrect

Recap

- CSRF: Cross Site Request Forgery
- An attack which forces an end user to execute unwanted actions on a web application in which he/she is currently authenticated.
- Caused because browser automatically includes authorization credentials such as cookies.
- Fixed using Origin headers and nonces
  - Origin headers not supported in older browsers.

Web Session Management

Slides credit: Dan Boneh

Same origin policy: "high level"

Same Origin Policy (SOP) for DOM:

- Origin A can access origin B's DOM if match on

  (scheme, domain, port)

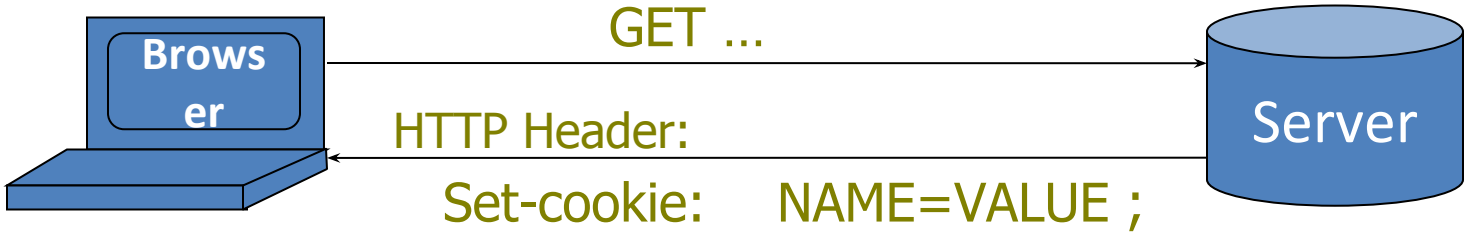
Same Original Policy (SOP) for cookies:

- Based on: **([scheme], domain, *path*)**

optional

scheme://domain:port/path?params

Setting/deleting cookies by server



if expires=NULL:
this session only

if expires=past date:
browser deletes cookie

domain = (when to send) ;
path = (when to send)

scope

secure = (only send over SSL);
expires = (when expires) ;
HttpOnly

Default scope is domain and path of setting URL

Scope setting rules (write SOP)

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

example:

host = “login.site.com”

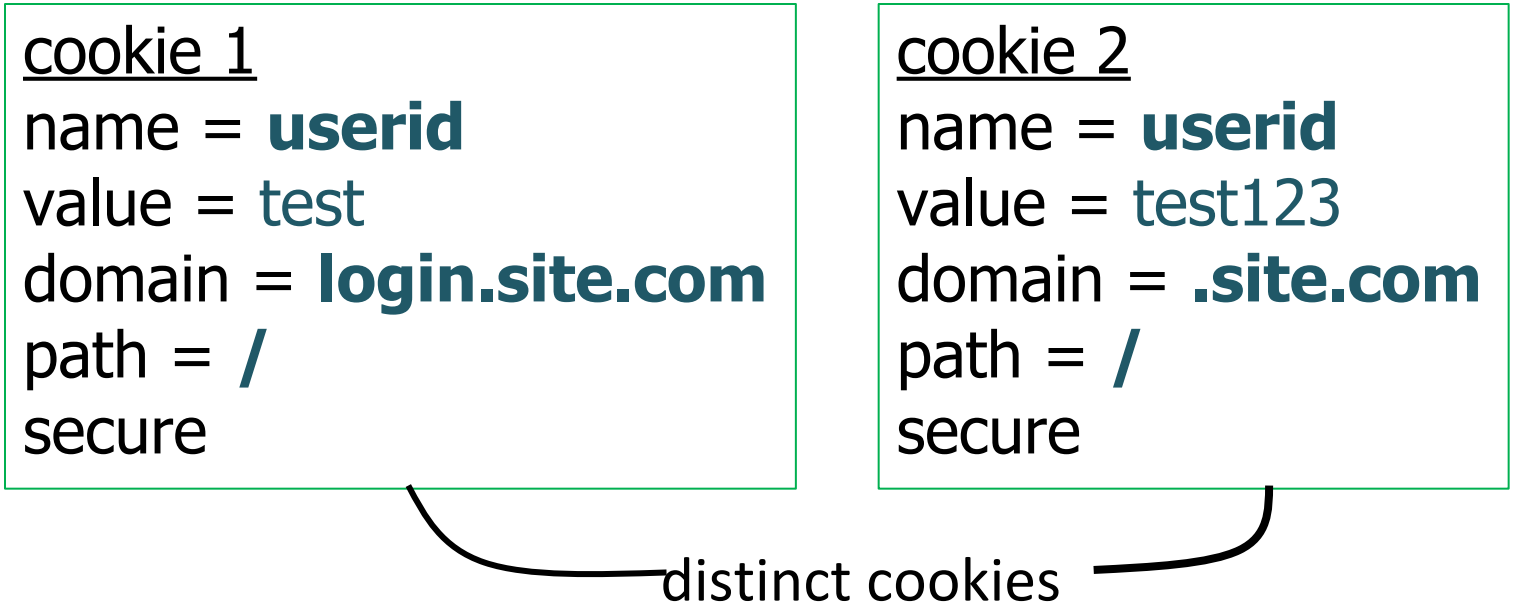
allowed domains	disallowed domains
login.site.com	**other.site.com**
.site.com	**othersite.com**
	.com

⇒ 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

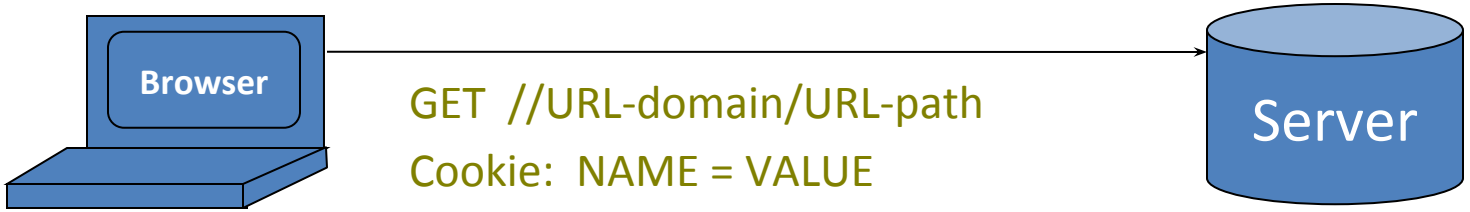
Cookies are identified by (name,domain,path)



Both cookies stored in browser's cookie jar;

both are in scope of **login.site.com**

Reading cookies on server (read SOP)



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”]

Goal: server only sees cookies in its scope

Examples

both set by **login.site.com**

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:

document.cookie = “name=value; expires=…; ”

Reading a cookie: alert(document.cookie)

prints string containing all cookies available for document (based on [protocol], domain, path)

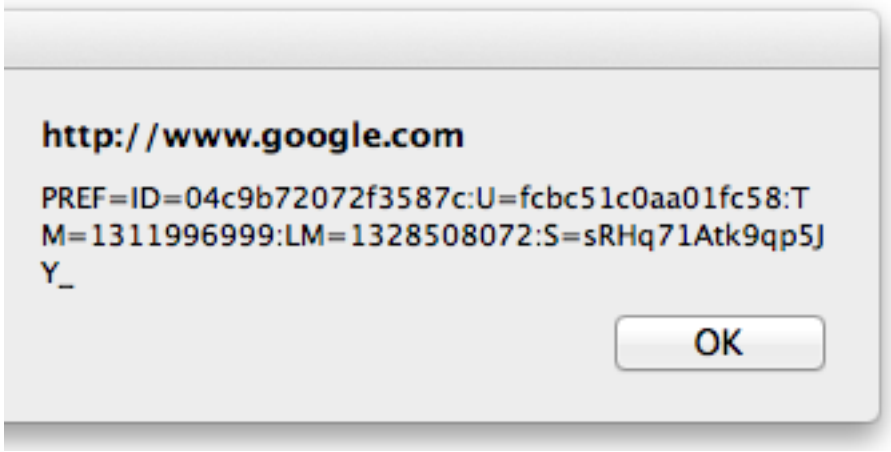
Deleting a cookie:

document.cookie = “name=; expires= Thu, 01-Jan-70”

document.cookie often used to customize page in Javascript

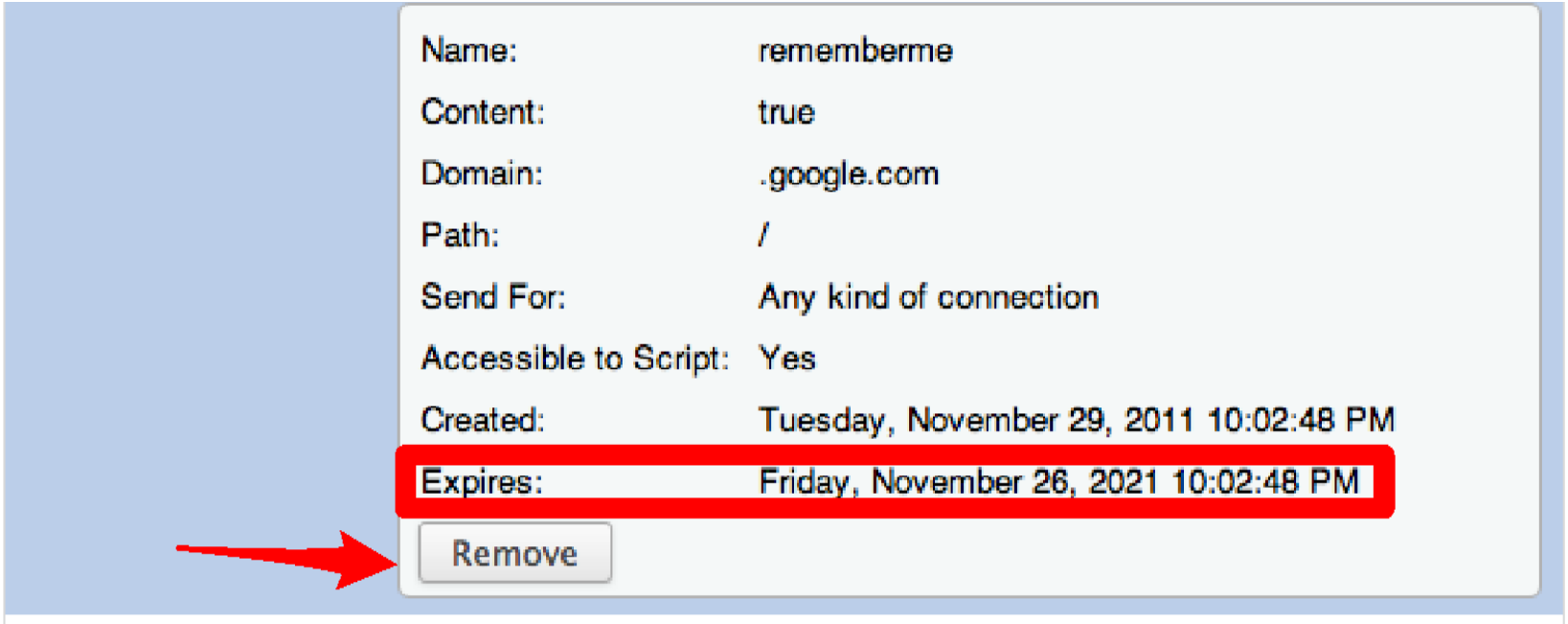
Javascript URL

javascript: alert(**document.cookie**)



Displays all cookies for current document

Viewing/deleting cookies in Browser UI



Cookie protocol problems

Server is blind:

- Does not see cookie attributes (e.g. secure, HttpOnly)
- Does not see which domain set the cookie

Server only sees: **Cookie: NAME=VALUE**

Example 1: login server problems

1. Alice logs in at **login.site.com**

   login.site.com sets session-id cookie for **.site.com**

2. Alice visits **evil.site.com**

   overwrites **.site.com** session-id cookie with session-id of user "badguy"

3. Alice visits **course.site.com** to submit homework.

   course.site.com thinks it is talking to "badguy"

Problem: **course.site.com** expects session-id from **login.site.com**; cannot tell that session-id cookie was overwritten

Example 2: "secure" cookies are not secure

Alice logs in at **https**://accounts.google.com

set-cookie: **SSID**=A7_ESAgDpKYk5TGnf; Domain=.google.com; Path=/ ;
Expires=Wed, 09-Mar-2022 18:35:11 GMT; **Secure; HttpOnly**
set-cookie: **SAPISID**=wj1gYKLFy-RmWybP/ANtKMtPIHNambvdI4; Domain=.google.com;Path=/ ;
Expires=Wed, 09-Mar-2022 18:35:11 GMT; **Secure**

Alice visits **http**://www.google.com (cleartext)

- Network attacker can inject into response
  Set-Cookie: SSID=badguy; secure
  and overwrite secure cookie

Problem: network attacker can re-write cookies over HTTP

Cookies have no integrity

User can change and delete cookie values

- Edit cookie database (cookies.sqlite)
- Modify Cookie header (TamperData extension)

Silly example: shopping cart software

Set-cookie: shopping-cart-total = 150 ($)

User edits cookie file (cookie poisoning):

Cookie: shopping-cart-total = 15 ($)

Similar problem with hidden fields

<INPUT TYPE="hidden" NAME=price VALUE="150">

Sessions

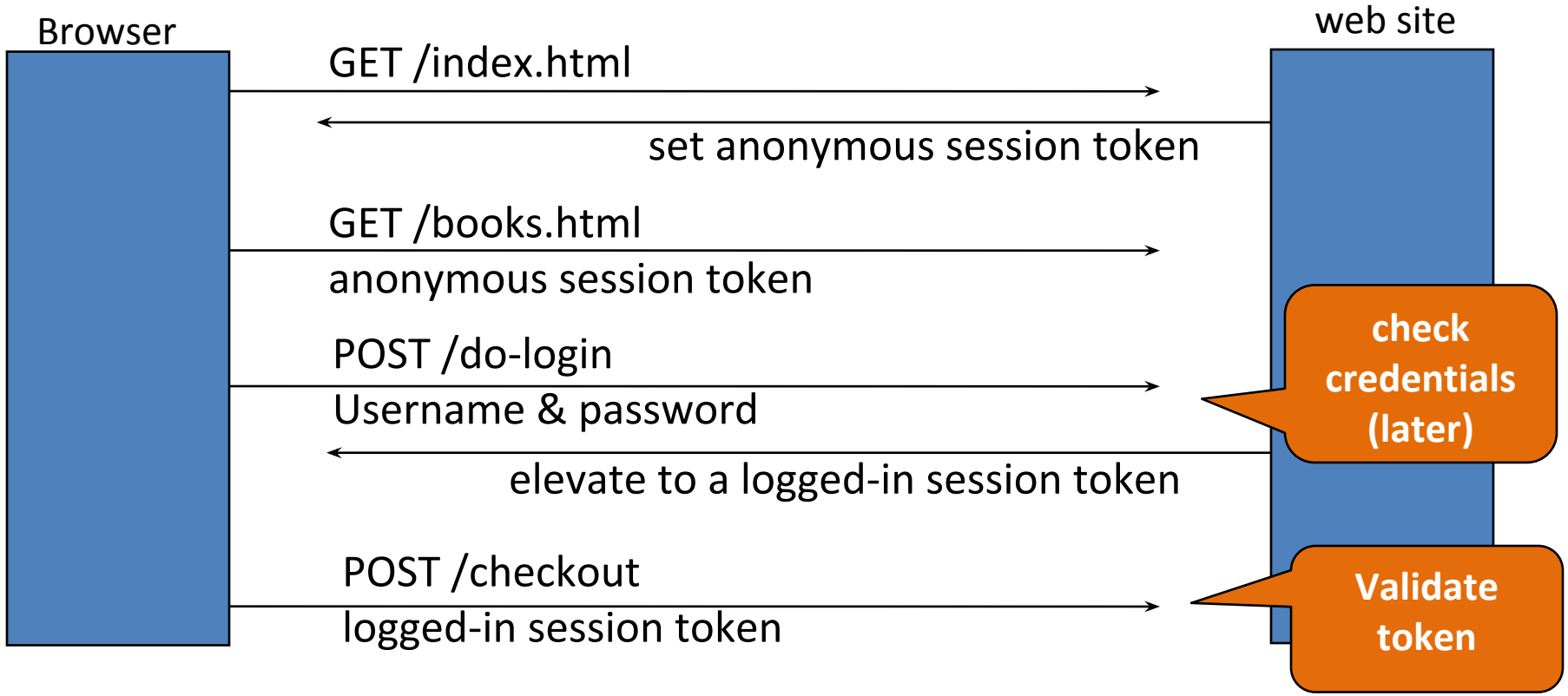
A sequence of requests and responses from one browser to one (or more) sites

- Session can be long (e.g. Gmail) or short
- without session mgmt:
  users would have to constantly re-authenticate

Session mgmt: authorize user once;

- All subsequent requests are tied to user

Session tokens



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">

Window.name DOM property

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
(or if user posts URL in a public blog)

In a hidden form field: does not work for long-lived sessions

---

Best answer: a combination of all of the above.

The HTTP referer header

```
GET /wiki/John_Ousterhout HTTP/1.1
Host: en.wikipedia.org
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.google.com/search?q=john+ousterhout&ie=utf-8&oe
```

Referer leaks URL session token to 3rd parties

The Logout Process

Web sites must provide a logout function:

- Functionality: let user to login as different user
- Security: prevent others from abusing account

What happens during logout:

1. Delete SessionToken from client
2. Mark session token as expired on server

Problem: many web sites do (1) but not (2) !!

$\Rightarrow$ Especially risky for sites who fall back to HTTP after login

Session hijacking

Attacker waits for user to login

then attacker steals user's Session Token
and "hijacks" session

$\Rightarrow$ attacker can issue arbitrary requests on behalf of user

Example: **FireSheep**.

Firefox extension that hijacks Facebook session tokens over WiFi.

Solution: use HTTPS

Session token theft

Example 1: login over HTTPS, but subsequent HTTP

- Enables cookie theft at wireless Café (e.g. Firesheep)
- Other reasons why session token sent in the clear:
  - HTTPS/HTTP mixed content pages at site

Example 2: Cross Site Scripting (XSS) exploits

Amplified by poor logout procedures:

- Logout must invalidate token on server

Session Fixation

Assume the session ID is set by a URL parameter.

http://gracebook.com/index.html?sessionid=1337

The attacker can trick the user into acting on behalf of the attacker.

Session fixation attacks

Suppose attacker can set the user's session token:

- For URL tokens, trick user into clicking on URL
- For cookie tokens, set using XSS exploits

Attack: (say, using URL tokens)

1. Attacker gets anonymous session token for site.com
2. Sends URL to user with attacker's session token
3. User clicks on URL and logs into site.com
   - this elevates attacker's token to logged-in token
4. Attacker uses elevated token to hijack user's session.

Session fixation: lesson

When elevating user from anonymous to logged-in:

always issue a new session token

After login, token changes to value unknown to attacker.

$\Rightarrow$ Attacker's token is not elevated.