CS155

Web Security: *Session Management*

Acknowledgments: Lecture slides are from the Computer Security course thought by Dan Boneh and John Mitchell at Stanford University. When slides are obtained from other sources, a a reference will be noted on the bottom of that slide. A full list of references is provided on the last slide.

Same origin policy: review

Review: Same Origin Policy (SOP) for DOM:

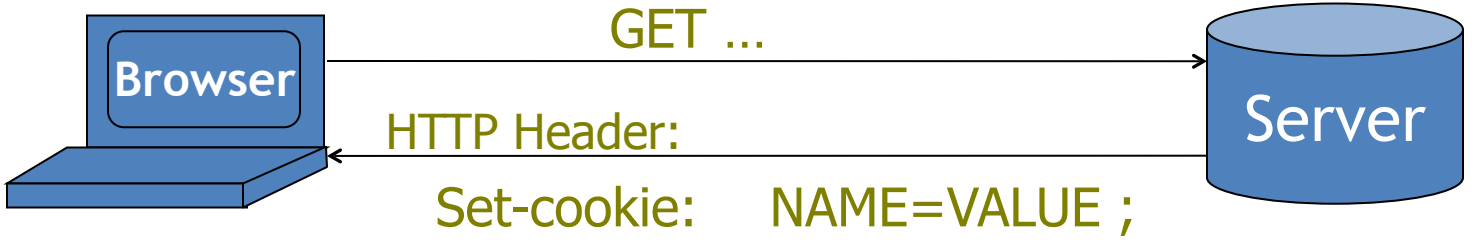
- Origin A can access origin B's DOM if match on **(scheme, domain, port)**

This lecture: Same Original Policy (SOP) for cookies:

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

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 ← weak XSS defense

SameSite = [lax | strict]

weak CSRF defense

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
login.site.com
.site.com

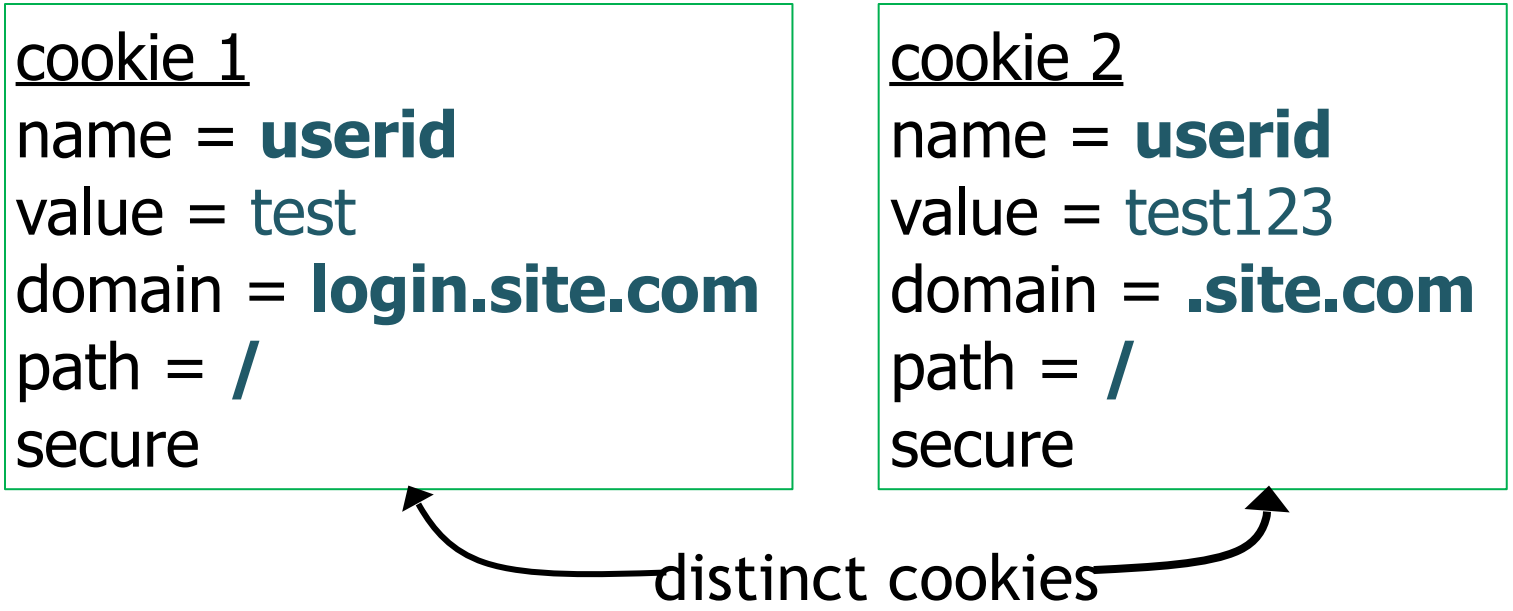
disallowed domains
other.site.com
othersite.com
.com

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

Problematic for sites like .stanford.edu (and some hosting centers)

path: can be set to anything

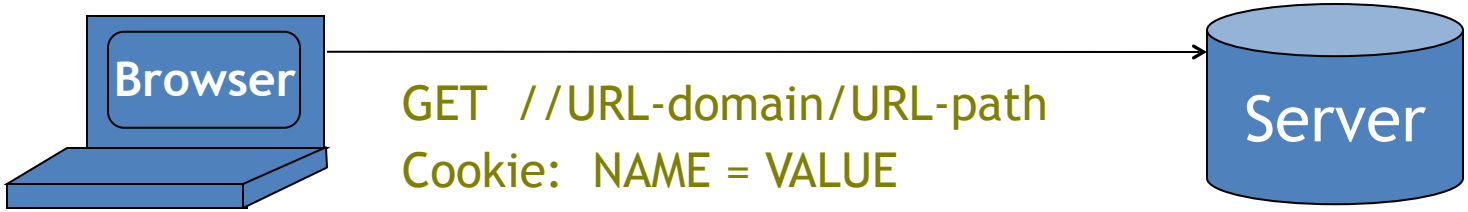
Cookies are identified by (name,domain,path)



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	cookie 2
name = **userid**	name = **userid**
value = u1	value = u2
domain = **login.site.com**	domain = **.site.com**
path = **/**	path = **/**
secure	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

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)

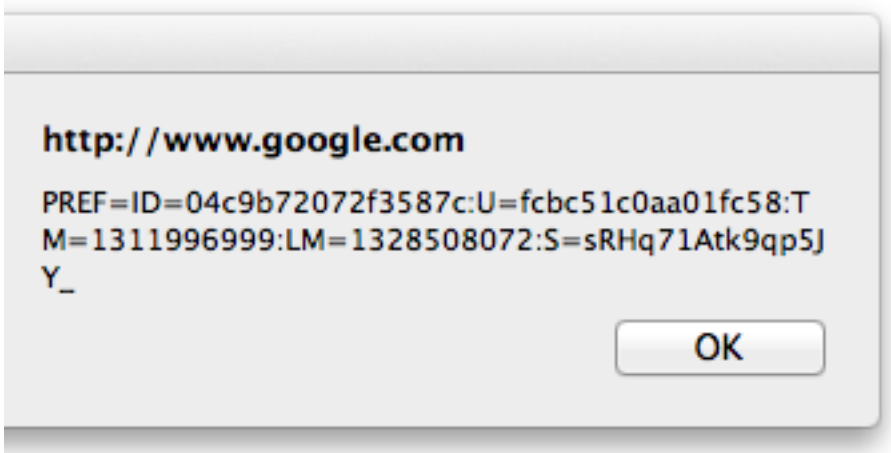
Deleting a cookie:

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

HttpOnly cookies: not included in document.cookie

Javascript URL

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



Displays all cookies for current document

Cookie protocol problems

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-2026 18:35:11 GMT; **Secure; HttpOnly**
set-cookie: **SAPISID**=wj1gYKLFy-RmWybP/ANtKMtPIHNambvdI4; Domain=.google.com;Path=/
Expires=Wed, 09-Mar-2026 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 HTTPS cookies !

- HTTPS cookie value cannot be trusted

Interaction with the DOM SOP

Cookie SOP path separation:

x.com/A does not see cookies of x.com/B

Not a security measure: x.com/A has access to DOM of x.com/B

```
<iframe src="x.com/B"></iframe>
alert(frames[0].document.cookie);
```

Path separation is done for efficiency not security:

x.com/A is only sent the cookies it needs

Cookies have no integrity

User can change and delete cookie values

- Edit cookie database (FF: cookies.sqlite)
- Modify Cookie header (FF: 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">

Not so silly ... (old)

- D3.COM Pty Ltd: ShopFactory 5.8
- @Retail Corporation: @Retail
- Adgrafix: Check It Out
- Baron Consulting Group: WebSite Tool
- ComCity Corporation: SalesCart
- Crested Butte Software: EasyCart
- Dansie.net: Dansie Shopping Cart
- Intelligent Vending Systems: Intellivend
- Make-a-Store: Make-a-Store OrderPage
- McMurtrey/Whitaker & Associates: Cart32 3.0
- pknutsen@nethut.no: CartMan 1.04
- Rich Media Technologies: JustAddCommerce 5.0
- SmartCart: SmartCart
- Web Express: Shoptron 1.2

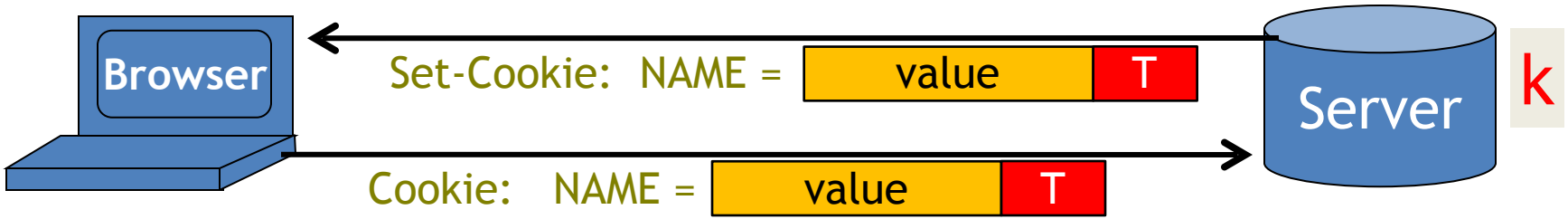
Source: http://xforce.iss.net/xforce/xfdb/4621

Solution: cryptographic checksums

Goal: data integrity

Requires server-side secret key k unknown to browser

Generate tag: T ⟵ MACsign(k, SID ll name ll value)



Verify tag: MACverify(k, SID ll name ll value, T)

Binding to session-id (SID) makes it harder to replay old cookies

Example: ASP.NET

System.Web.Configuration.MachineKey

- Secret web server key intended for cookie protection

Creating an encrypted cookie with integrity:

```
HttpCookie   cookie = new HttpCookie(name, val);
HttpCookie   encodedCookie =
                HttpSecureCookie.Encode (cookie);
```

Decrypting and validating an encrypted cookie:

```
HttpSecureCookie.Decode (cookie);
```

Session Management

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;

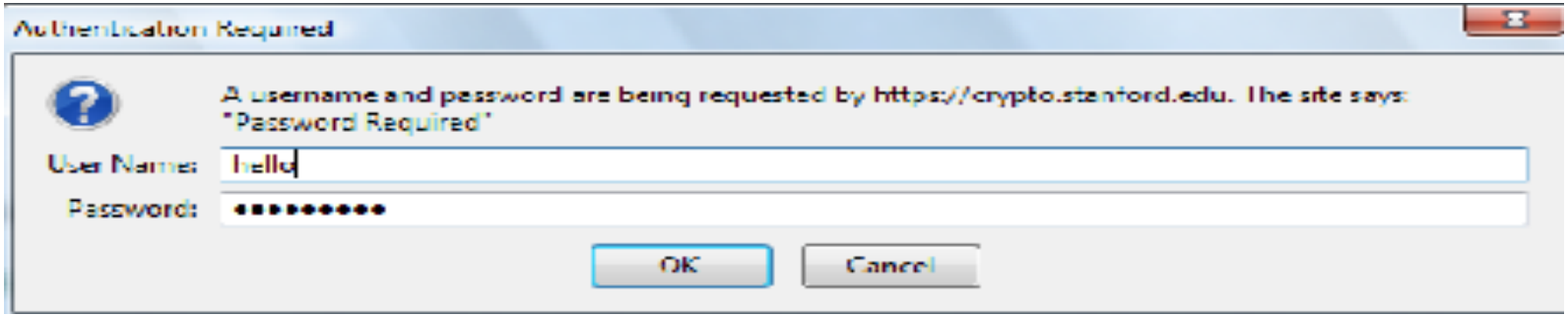
- All subsequent requests are tied to user

Pre-history: HTTP auth

HTTP request: GET /index.html

HTTP response contains:

WWW-Authenticate: Basic realm="Password Required"



Browsers sends hashed password on all subsequent HTTP requests:

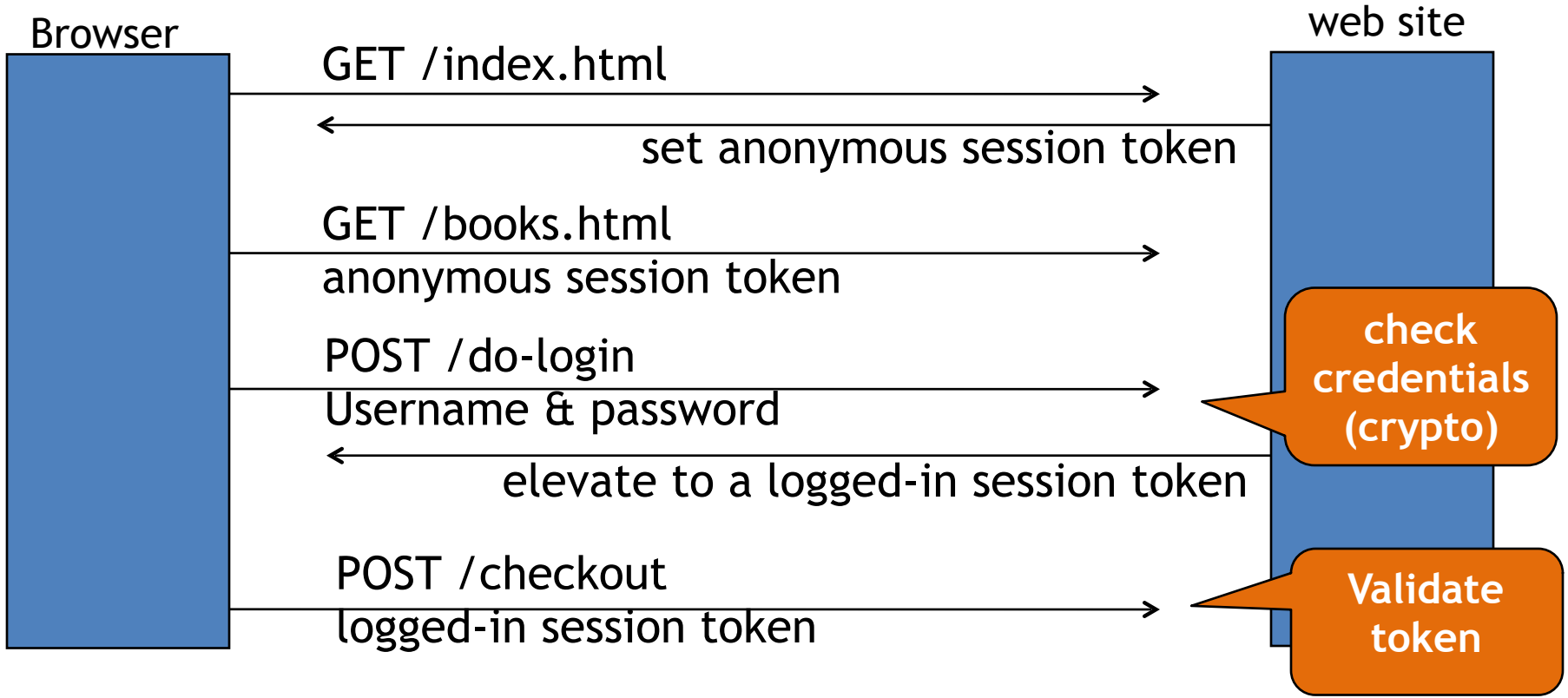
Authorization: Basic ZGFddfibzsdfgkjheczI1NXRleHQ=

HTTP auth problems

Hardly used in commercial sites:

- User cannot log out other than by closing browser
  - What if user has multiple accounts? multiple users on same machine?
- Site cannot customize password dialog
- Confusing dialog to users
- Easily spoofed

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">

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

Referer supression:

- not sent when HTTPS site refers to an HTTP site
- in HTML5: `<a rel="noreferrer" href=www.example.com>`

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) !!

⇒ Especially risky for sites who fall back to HTTP after login

Session hijacking

Session hijacking

Attacker waits for user to login

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

⇒ attacker can issue arbitrary requests on behalf of user

Example: **FireSheep** [2010]

Firefox extension that hijacks Facebook
session tokens over WiFi. Solution: HTTPS after login

Beware: Predictable tokens

Example 1: counter

⇒ user logs in, gets counter value,

can view sessions of other users

Example 2: weak MAC. token = { userid, MAC_k(userid) }

- Weak MAC exposes **k** from few cookies.

Apache Tomcat: generateSessionId()

- Returns random session ID [server retrieves client state based on sess-id]

Session tokens must be unpredictable to attacker

To generate: use underlying framework (e.g. ASP, Tomcat, Rails)

Rails: token = MD5(current time, random nonce)

Beware: Session token theft

Example 1: login over HTTPS, but subsequent HTTP

- Enables cookie theft at wireless Café (e.g. Firesheep)
- Other ways network attacker can steal token:
  - Site has mixed HTTPS/HTTP pages ⇒ token sent over HTTP
  - Man-in-the-middle attacks on SSL

Example 2: Cross Site Scripting (XSS) exploits

Amplified by poor logout procedures:

- Logout must invalidate token on server

Mitigating SessionToken theft by binding SessionToken to client's computer

A common idea: embed machine specific data in SID

Client IP addr: makes it harder to use token at another machine

- But honest client may change IP addr during session
  - client will be logged out for no reason.

SSL session id: same problem as IP address (and even worse)

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

⇒ Attacker's token is not elevated.

Summary

- Always assume cookie data retrieved from client is adversarial
- Session tokens are split across multiple client state mechanisms:
  - Cookies, hidden form fields, URL parameters
  - Cookies by themselves are insecure (CSRF, cookie overwrite)
  - Session tokens must be unpredictable and resist theft by network attacker
- Ensure logout invalidates session on server

THE END