CS-446: Information Systems Security

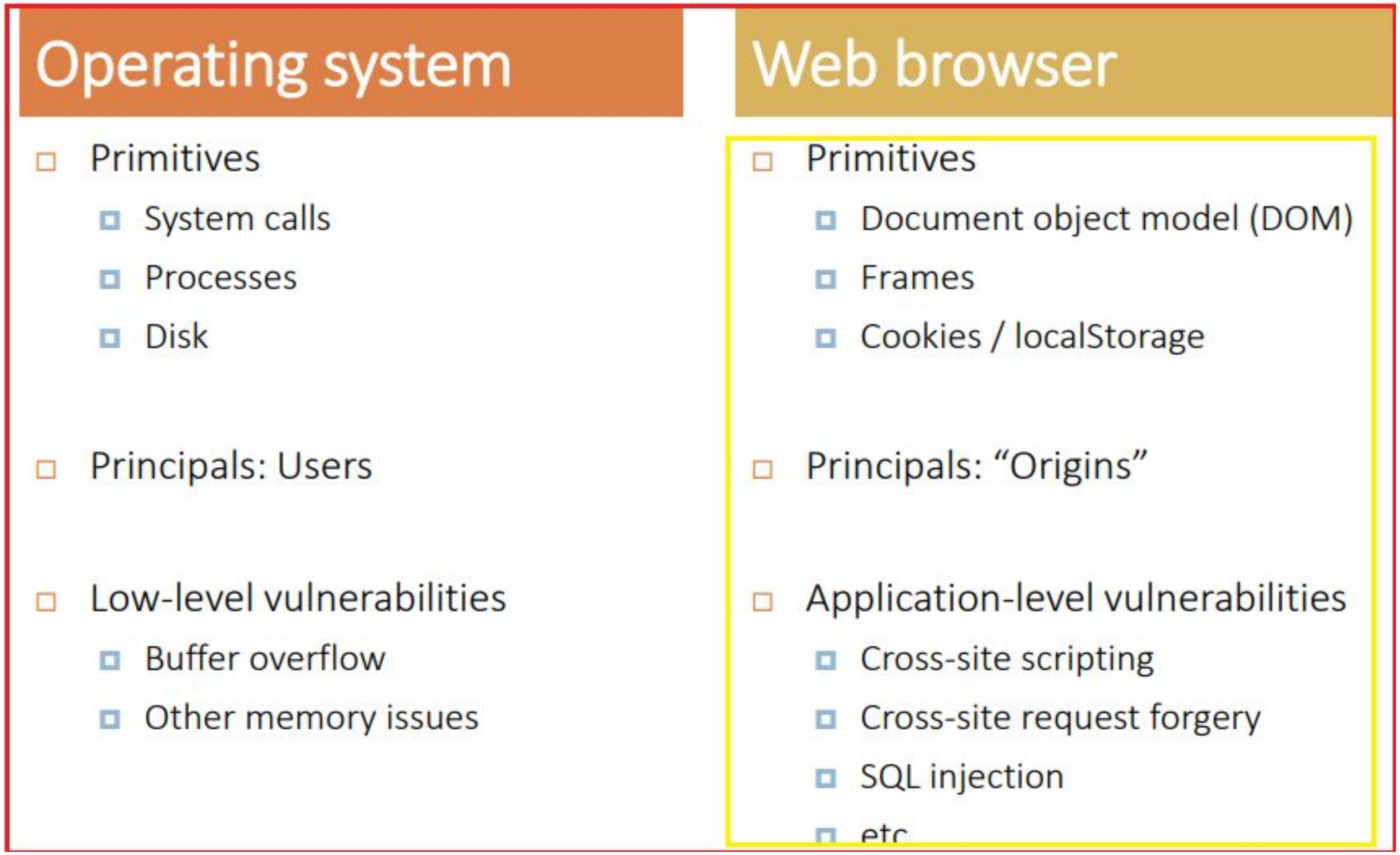
Lecture # 34: Web Session Management

Department of Computer Science

FAST-NUCES

Overview

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



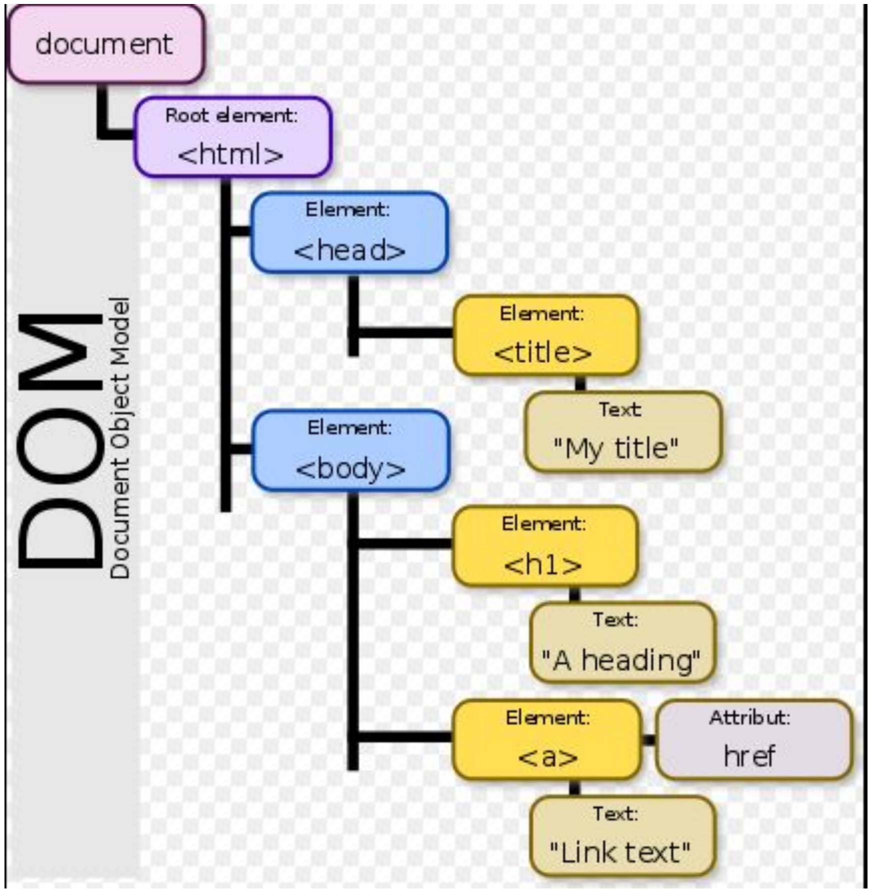
Operating system

- Primitives
  - System calls
  - Processes
  - Disk
- Principals: Users
- Low-level vulnerabilities
  - Buffer overflow
  - Other memory issues

Web browser

- Primitives
  - Document object model (DOM)
  - Frames
  - Cookies / localStorage
- Principals: "Origins"
- Application-level vulnerabilities
  - Cross-site scripting
  - Cross-site request forgery
  - SQL injection
  - etc

Web Session Management (Cookies)


DOM
Document Object Model
document
Root element:
<html>
Element:
<head>
Element:
<title>
Text
"My title"
Element:
<body>
Element:
<h1>
Text:
"A heading"
Element:
<a>
Attribut:
href
Text:
"Link text"

Same Origin Policy (SOP)

Definition - What does *Same Origin Policy (SOP)* mean?

Same origin policy (SOP) is a security mechanism in a client browser that permits webpage scripts to access their associated website's data and methods but restricts its access to scripts and data stored by other websites.

In computing, the **same-origin policy** is an important concept in the web application security model. Under the **policy**, a web browser permits scripts contained in a first web page to access data in a second web page, but only if both web pages have the same **same origin**.

Same-origin Policy

Assume you are logged into Facebook and visit a malicious website in another browser tab. Without the same origin policy JavaScript on that website could do anything to your Facebook account that you are allowed to do. For example read private messages, post status updates, analyse the HTML DOM-tree after you entered your password before submitting the form.

But of course Facebook wants to use JavaScript to enhance the user experience. So it is important that the browser can detect that this JavaScript is trusted to access Facebook resources. That's where the same origin policy comes into play: If the JavaScript is included from a HTML page on facebook.com, it may access facebook.com resources.

Now replace Facebook with your online banking website, and it will be obvious that this is an issue.

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)

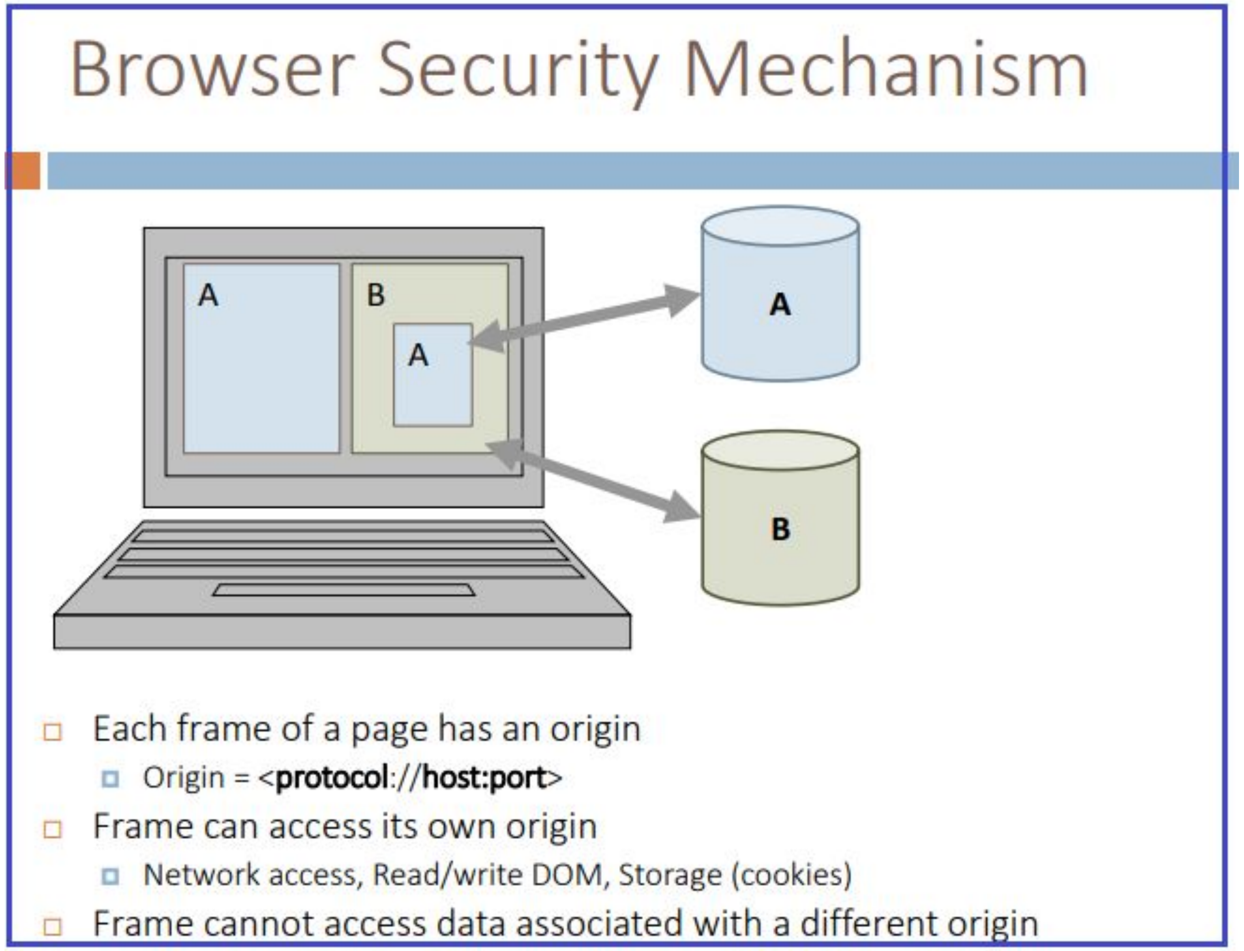
Today: Same Original Policy (SOP) for cookies:

- Generally speaking, based on:

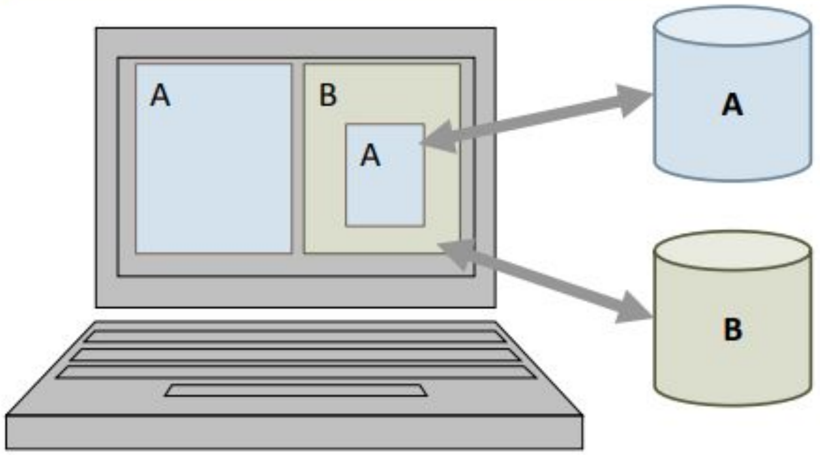
  ([scheme], domain, *path*)

optional

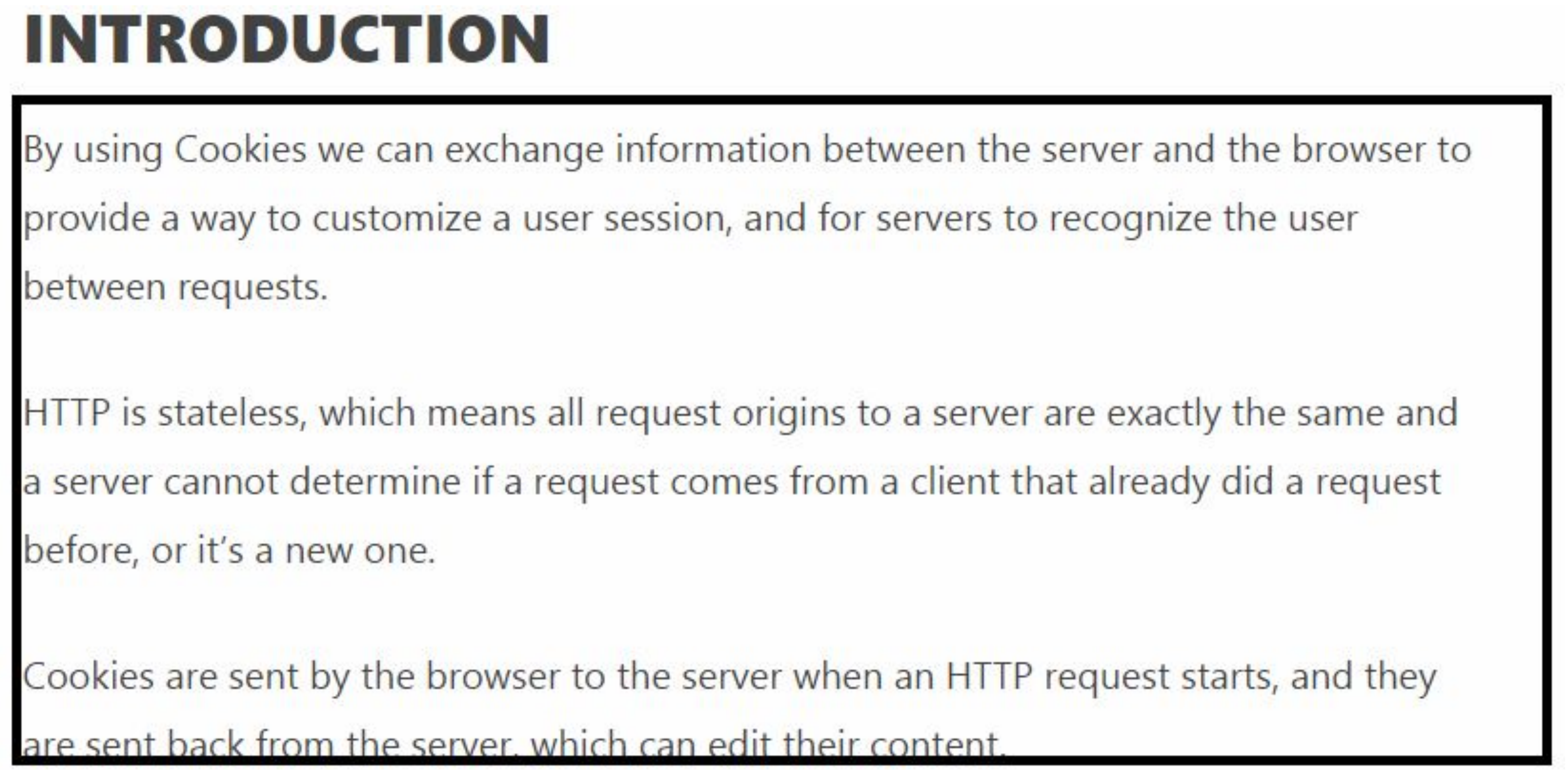
scheme://domain:port/path?params



Browser Security Mechanism



- Each frame of a page has an origin
  - Origin = <**protocol**://**host:port**>
- Frame can access its own origin
  - Network access, Read/write DOM, Storage (cookies)
- Frame cannot access data associated with a different origin

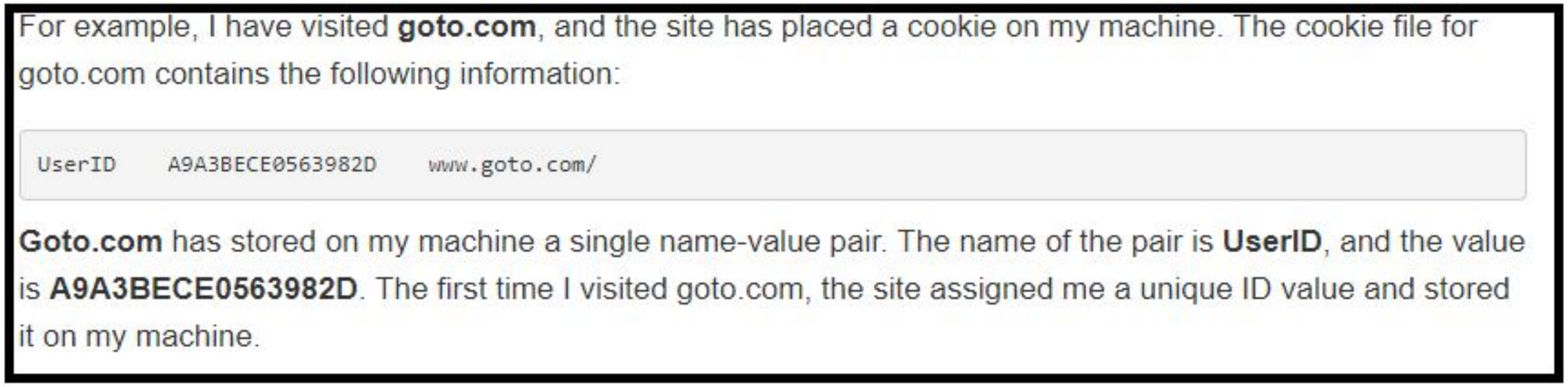


INTRODUCTION

By using Cookies we can exchange information between the server and the browser to provide a way to customize a user session, and for servers to recognize the user between requests.

HTTP is stateless, which means all request origins to a server are exactly the same and a server cannot determine if a request comes from a client that already did a request before, or it’s a new one.

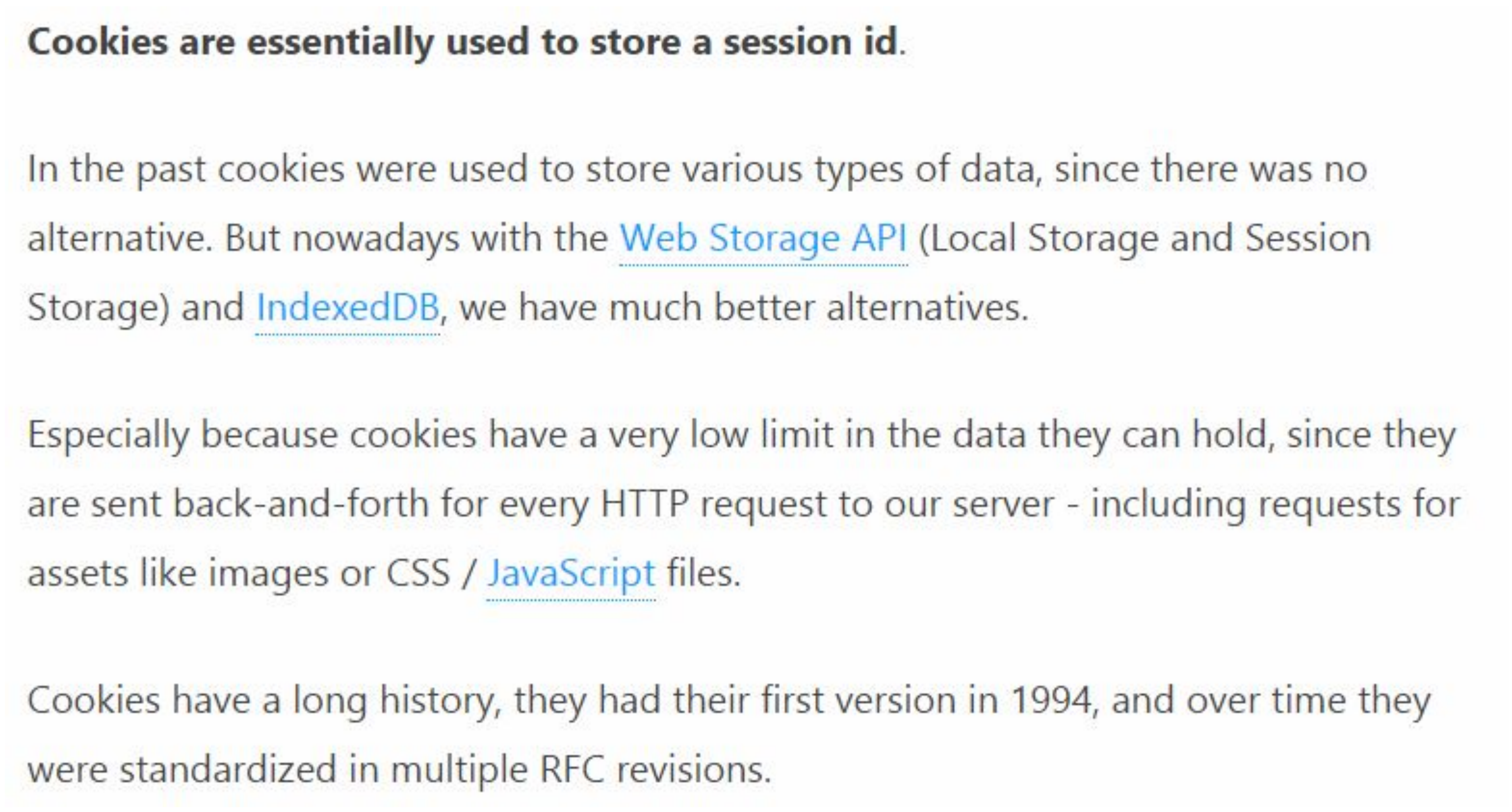
Cookies are sent by the browser to the server when an HTTP request starts, and they are sent back from the server, which can edit their content.



For example, I have visited **goto.com**, and the site has placed a cookie on my machine. The cookie file for goto.com contains the following information:

```
UserID    A9A3BECE0563982D    www.goto.com/
```

Goto.com has stored on my machine a single name-value pair. The name of the pair is **UserID**, and the value is **A9A3BECE0563982D**. The first time I visited goto.com, the site assigned me a unique ID value and stored it on my machine.

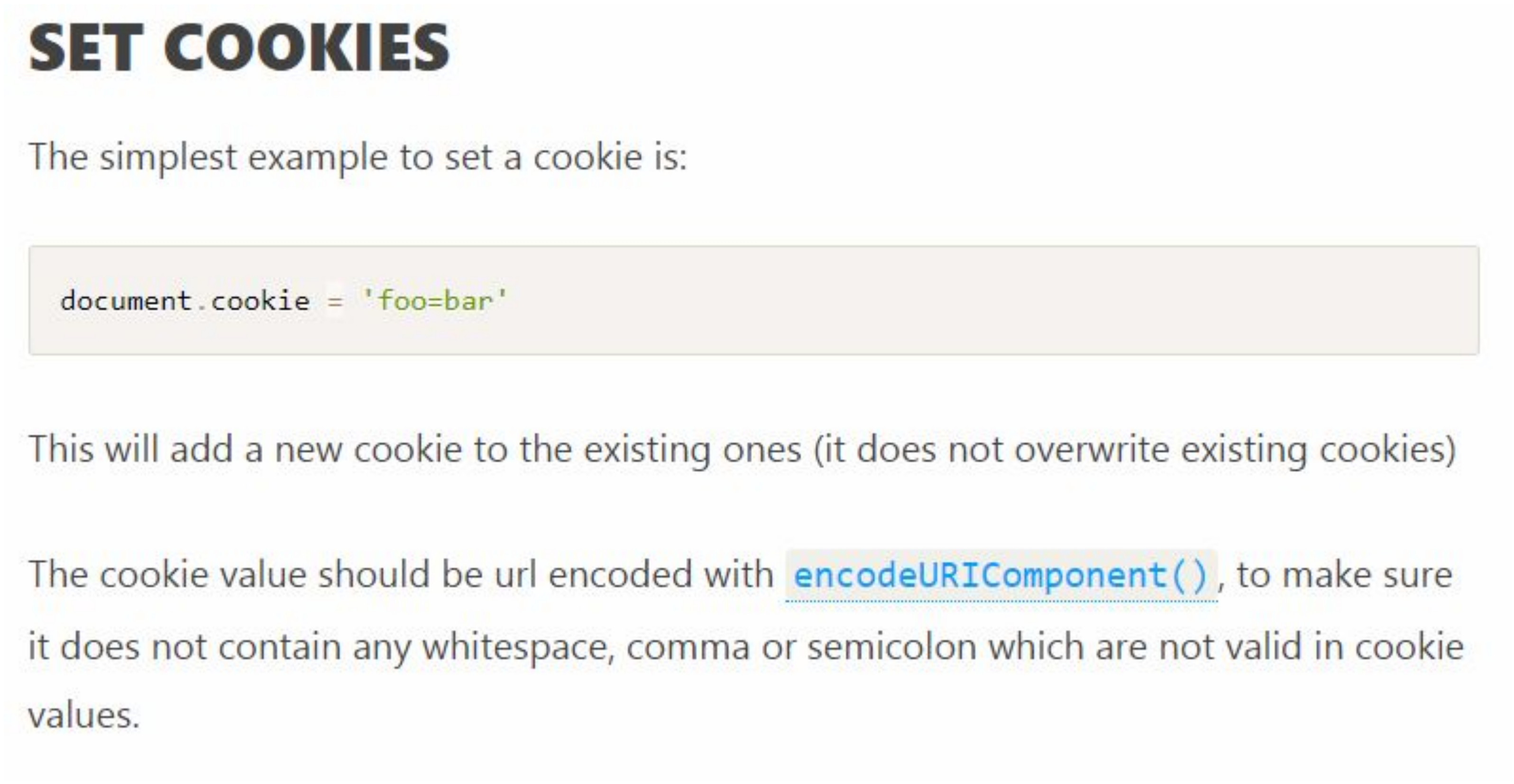


Cookies are essentially used to store a session id.

In the past cookies were used to store various types of data, since there was no alternative. But nowadays with the Web Storage API (Local Storage and Session Storage) and IndexedDB, we have much better alternatives.

Especially because cookies have a very low limit in the data they can hold, since they are sent back-and-forth for every HTTP request to our server - including requests for assets like images or CSS / JavaScript files.

Cookies have a long history, they had their first version in 1994, and over time they were standardized in multiple RFC revisions.



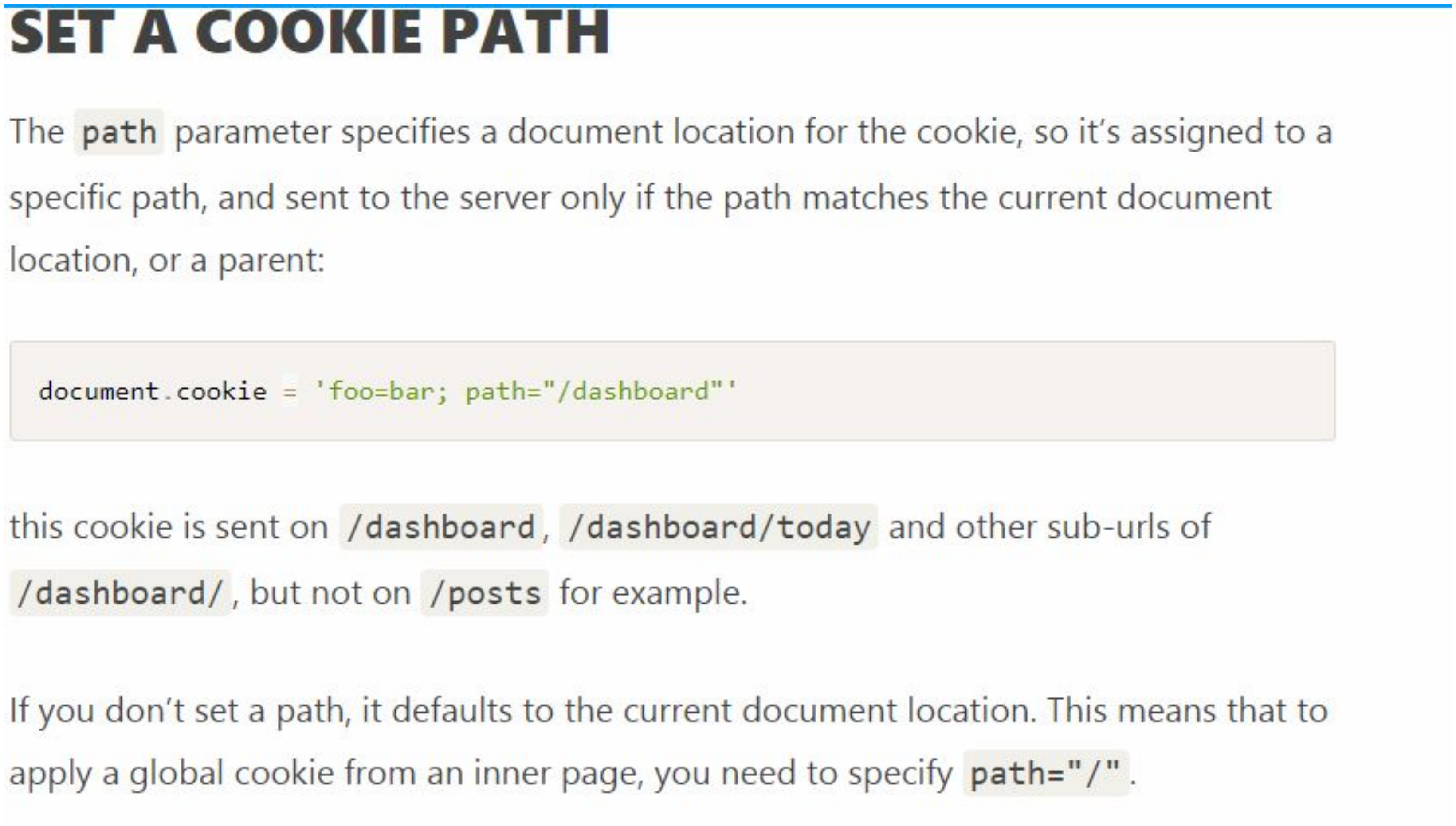
SET COOKIES

The simplest example to set a cookie is:

```
document.cookie = 'foo=bar'
```

This will add a new cookie to the existing ones (it does not overwrite existing cookies)

The cookie value should be url encoded with `encodeURIComponent()`, to make sure it does not contain any whitespace, comma or semicolon which are not valid in cookie values.



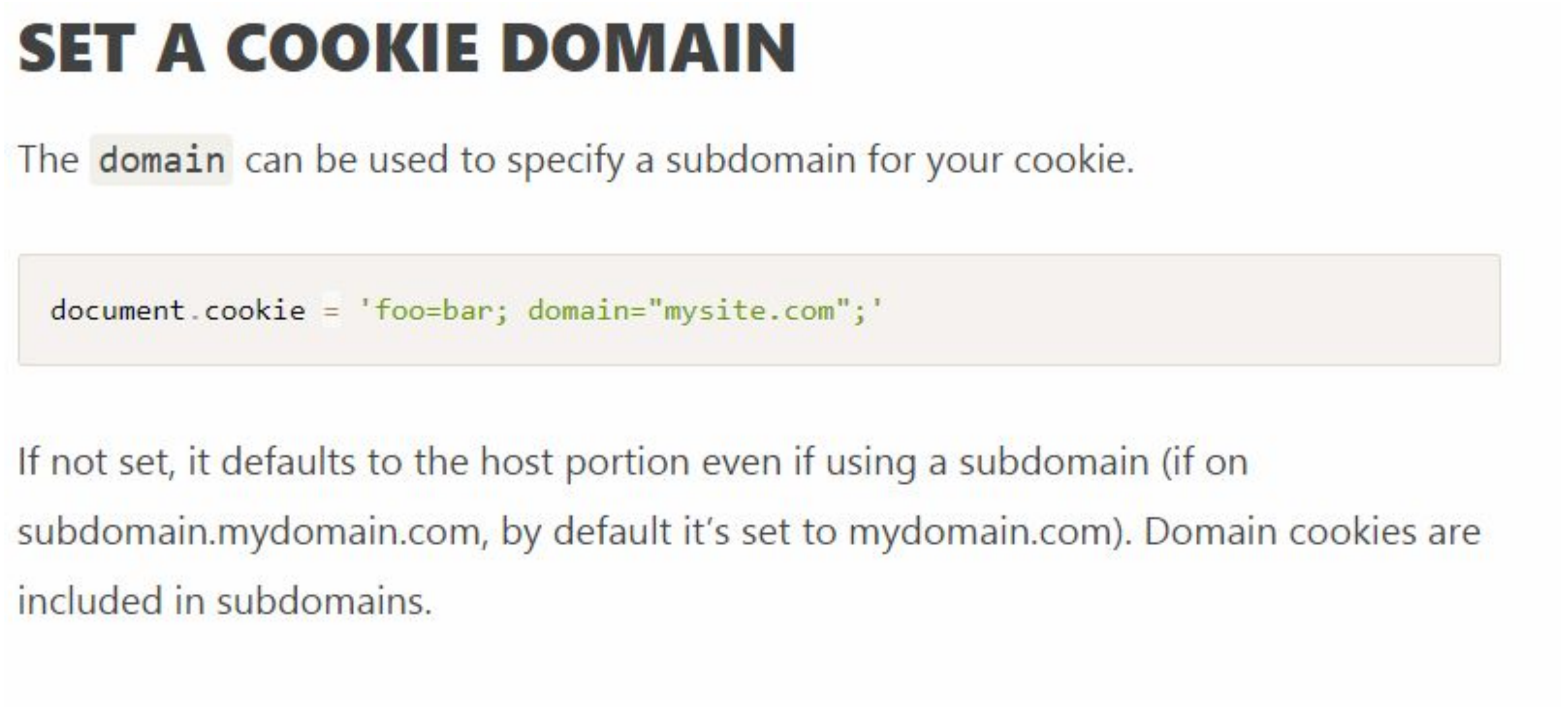
SET A COOKIE PATH

The `path` parameter specifies a document location for the cookie, so it's assigned to a specific path, and sent to the server only if the path matches the current document location, or a parent:

```
document.cookie = 'foo=bar; path="/dashboard"'
```

this cookie is sent on `/dashboard`, `/dashboard/today` and other sub-urls of `/dashboard/`, but not on `/posts` for example.

If you don't set a path, it defaults to the current document location. This means that to apply a global cookie from an inner page, you need to specify `path="/"`.



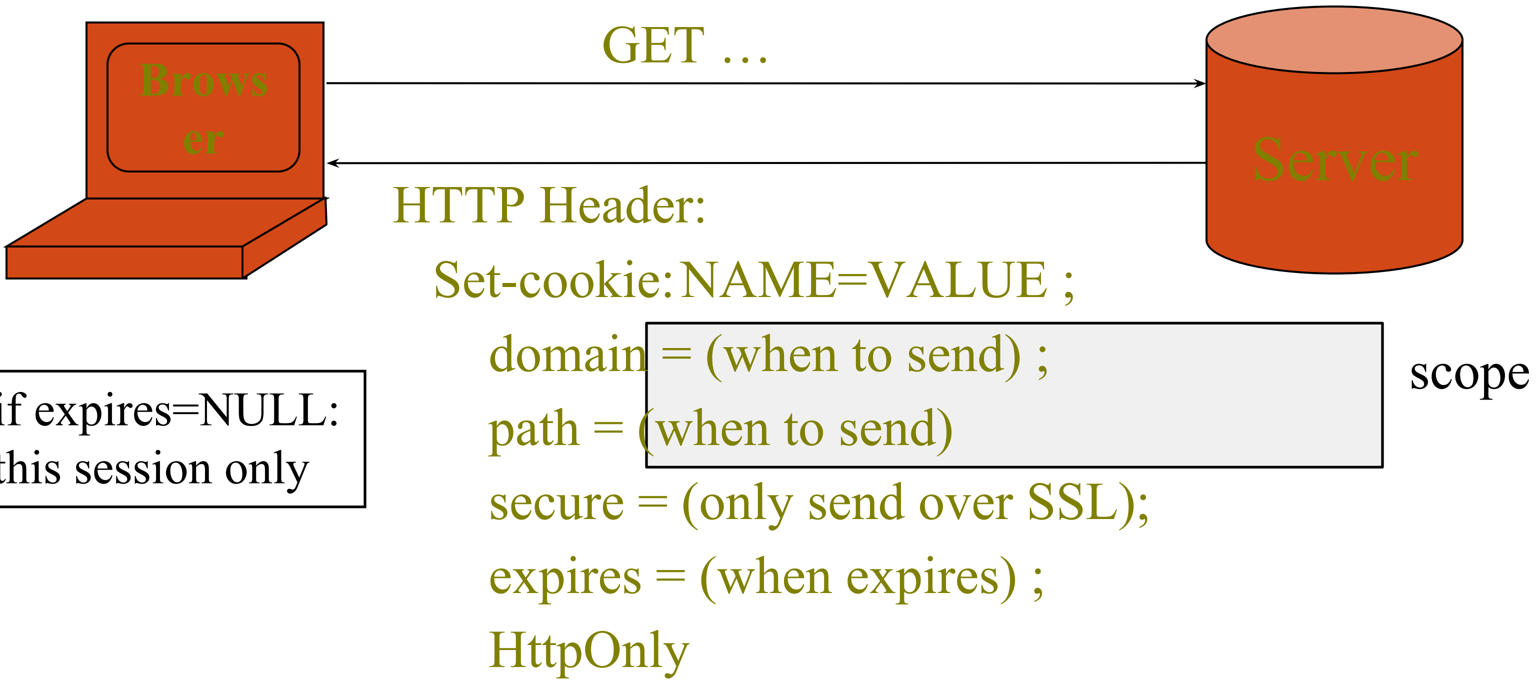
SET A COOKIE DOMAIN

The `domain` can be used to specify a subdomain for your cookie.

```
document.cookie = 'foo=bar; domain="mysite.com";'
```

If not set, it defaults to the host portion even if using a subdomain (if on subdomain.mydomain.com, by default it's set to mydomain.com). Domain cookies are included in subdomains.

Setting/deleting cookies by server



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

if expires=NULL:
this session only

- 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)

<u>cookie 1</u>
name = **userid**
value = test
domain = **login.site.com**
path = /
secure

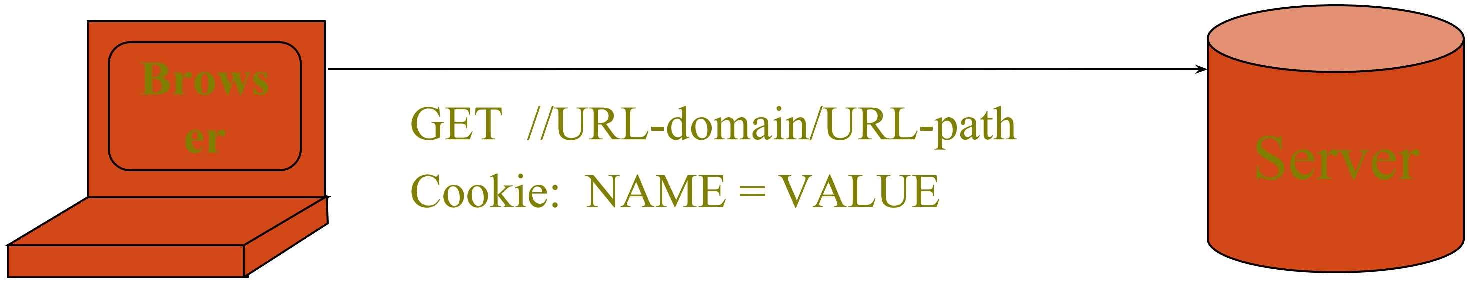
<u>cookie 2</u>
name = **userid**
value = test123
domain = **.site.com**
path = /
secure

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
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 (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)

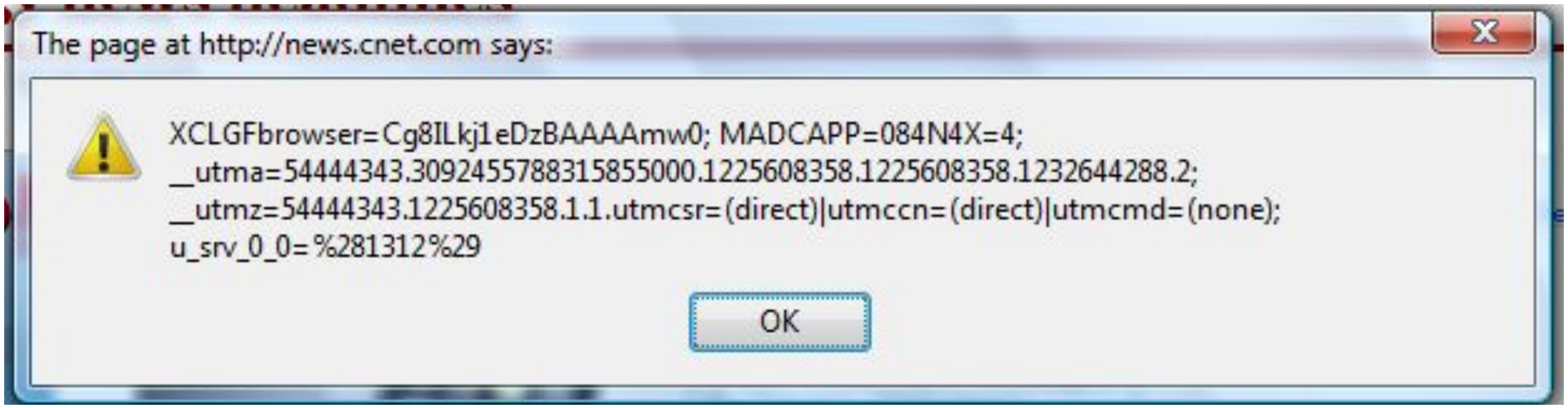
- Deleting a cookie:

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

document.cookie often used to customize page in Javascript

Javascript URL

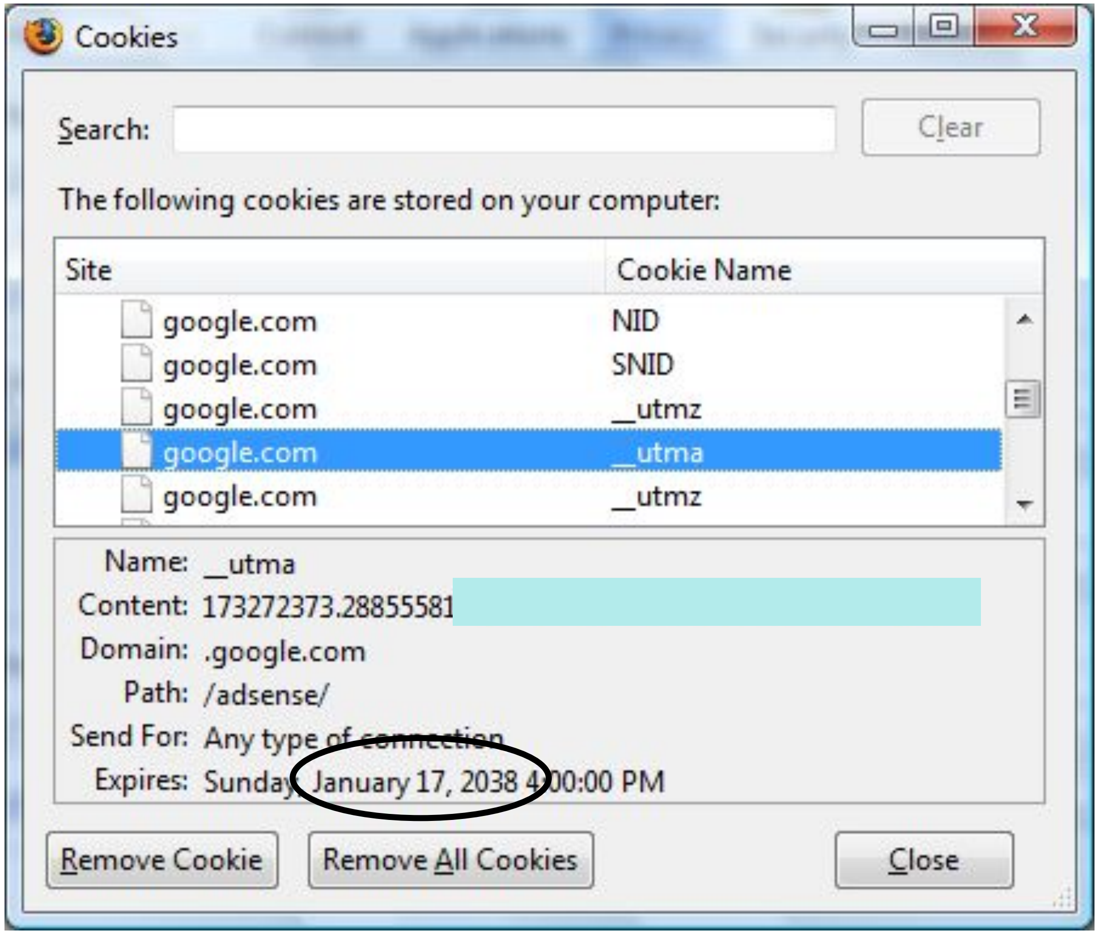
javascript: alert(**document.cookie**)



```
XCLGFbrowser=Cg8ILkj1eDzBAAAAmw0; MADCAPP=084N4X=4;
__utma=54444343.3092455788315855000.1225608358.1225608358.1232644288.2;
__utmz=54444343.1225608358.1.1.utmcsr=(direct)|utmccn=(direct)|utmcmd=(none);
u_srv_0_0=%281312%29
```

Displays all cookies for current document

Viewing/deleting cookies in Browser UI



The following cookies are stored on your computer:

Site	Cookie Name
google.com	NID
google.com	SNID
google.com	__utmz
google.com	__utma
google.com	__utmz

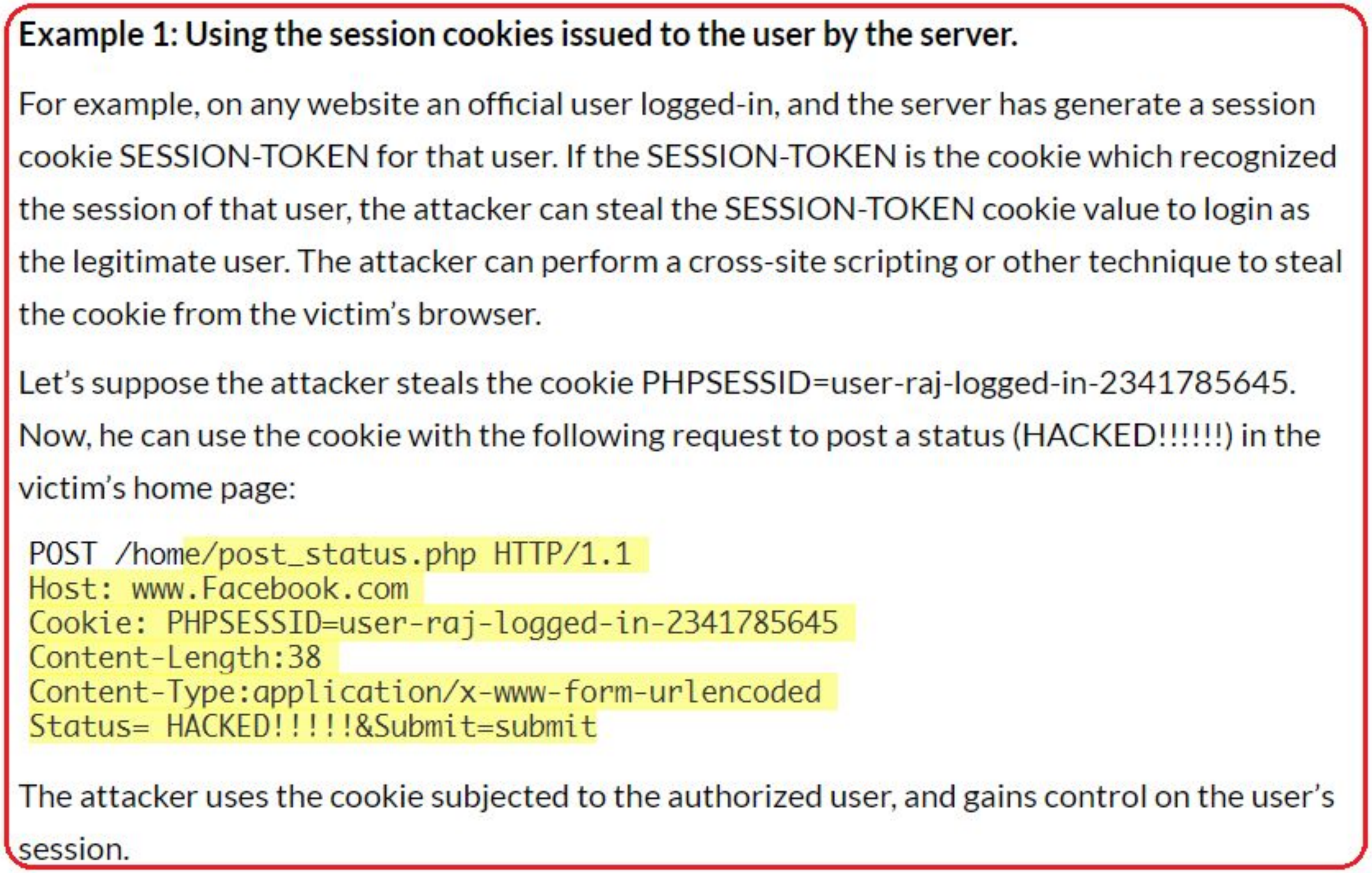
Name: __utma
Content: 173272373.28855581
Domain: .google.com
Path: /adsense/
Send For: Any type of connection
Expires: Sunday, January 17, 2038 4:00:00 PM

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: Using the session cookies issued to the user by the server.

For example, on any website an official user logged-in, and the server has generate a session cookie SESSION-TOKEN for that user. If the SESSION-TOKEN is the cookie which recognized the session of that user, the attacker can steal the SESSION-TOKEN cookie value to login as the legitimate user. The attacker can perform a cross-site scripting or other technique to steal the cookie from the victim's browser.

Let's suppose the attacker steals the cookie PHPSESSID=user-raj-logged-in-2341785645. Now, he can use the cookie with the following request to post a status (HACKED!!!!!!) in the victim's home page:

```
POST /home/post_status.php HTTP/1.1
Host: www.Facebook.com
Cookie: PHPSESSID=user-raj-logged-in-2341785645
Content-Length:38
Content-Type:application/x-www-form-urlencoded
Status= HACKED!!!!!&Submit=submit
```

The attacker uses the cookie subjected to the authorized user, and gains control on the user's session.

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"

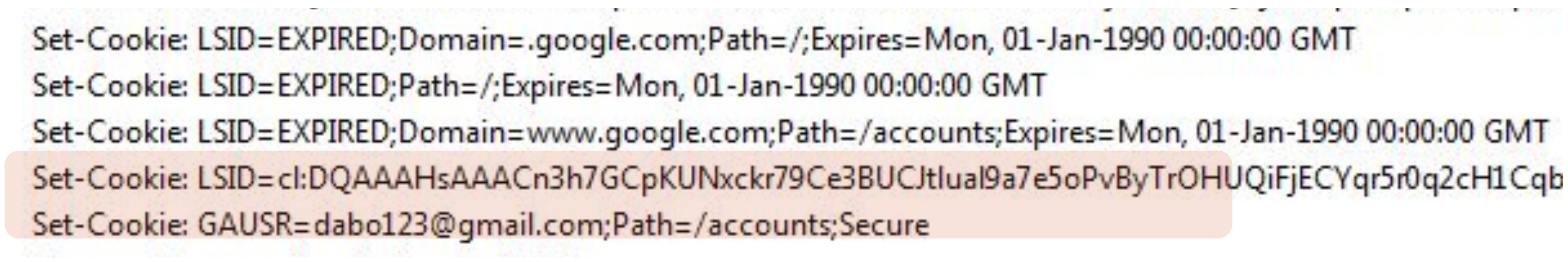
- Alice visits **cs155.site.com** to submit homework.

  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

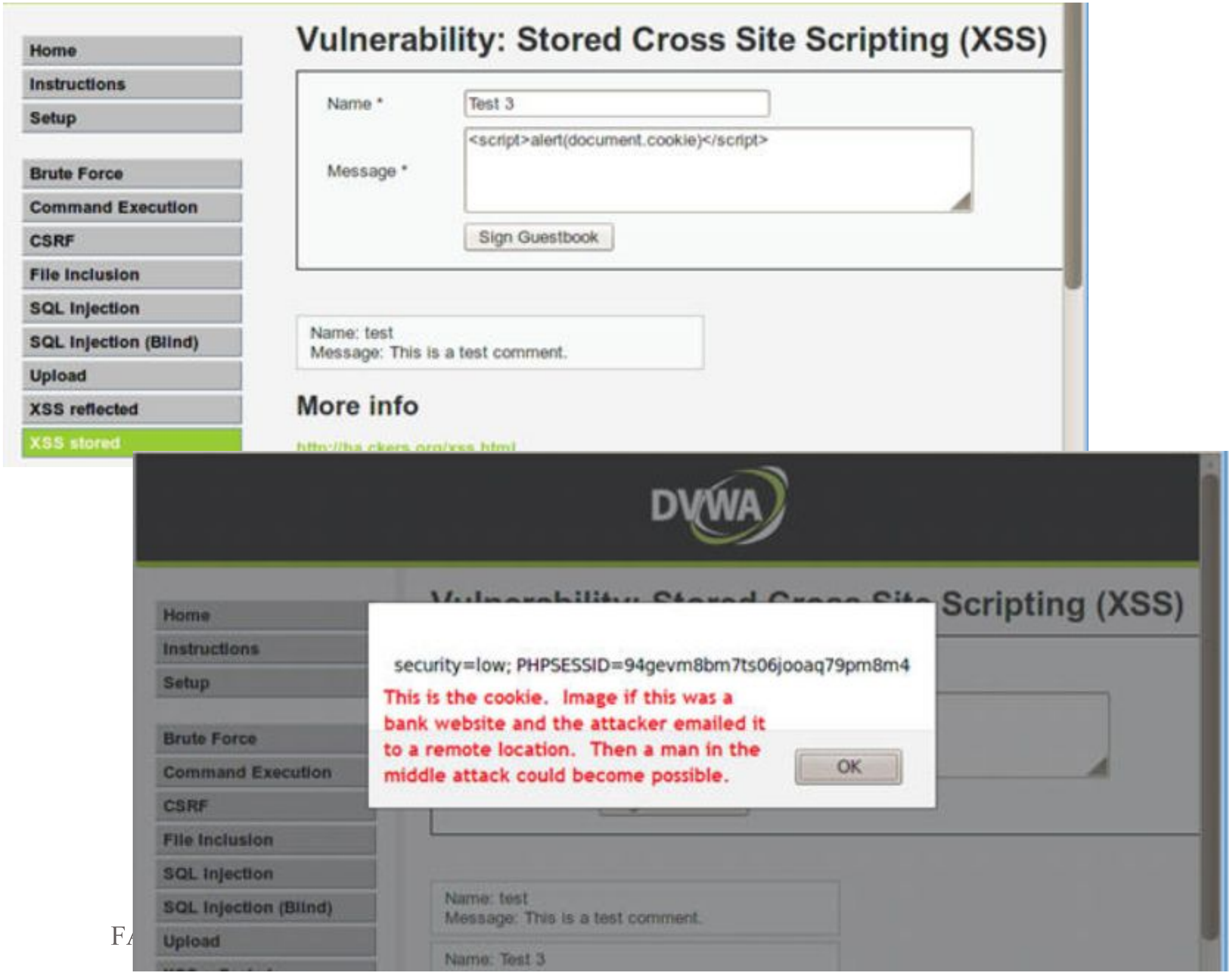
  and overwrite secure cookie

- Problem: network attacker can re-write HTTPS cookies !

  ⇒ HTTPS cookie value cannot be trusted

Accessing Cookies via DOM

- Same domain scoping rules as for sending cookies to the server
- document.cookie returns a string with all cookies available for the document
  - Often used in JavaScript to customize page
- Javascript can set and delete cookies via DOM
  - document.cookie = "name=value; expires=…; "
  - document.cookie = "name=; expires= Thu, 01-Jan-70"



Vulnerability: Stored Cross Site Scripting (XSS)

Name * Test 3

Message * `<script>alert(document.cookie)</script>`

Name: test
Message: This is a test comment.

More info

http://ha.ckers.org/xss.html

```
security=low; PHPSESSID=94gevm8bm7ts06jooaq79pm8m4
```

This is the cookie. Image if this was a bank website and the attacker emailed it to a remote location. Then a man in the middle attack could become possible.

Name: test
Message: This is a test comment.

Name: Test 3

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 ($)

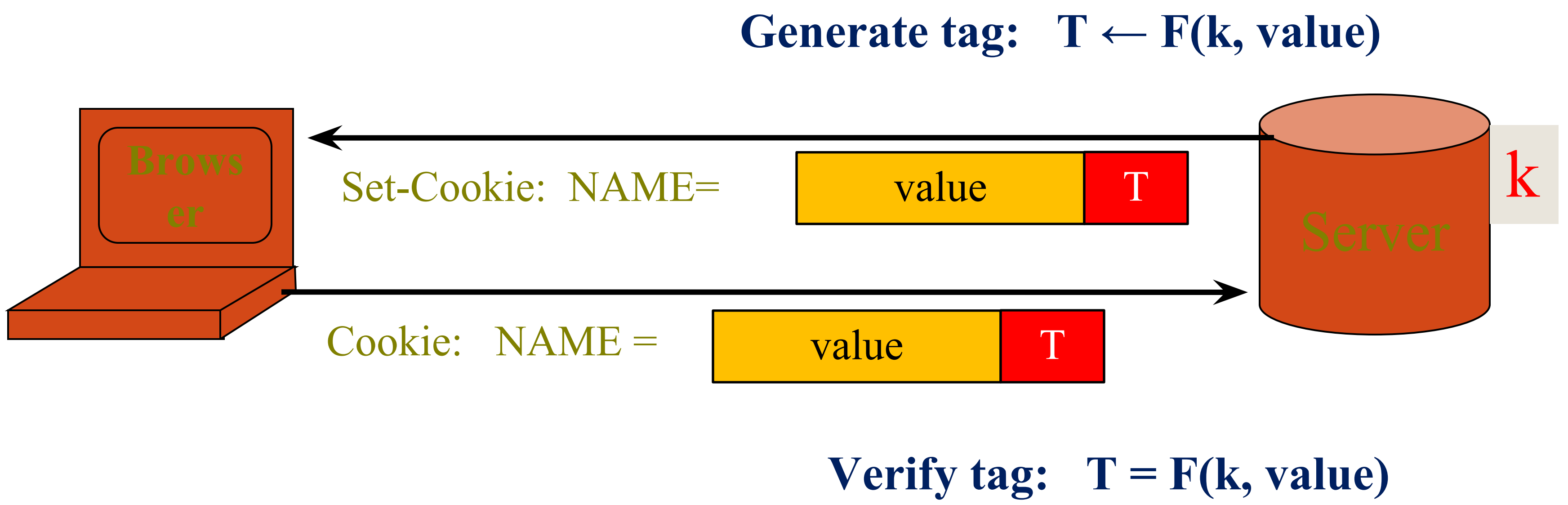
Similar to problem with hidden fields

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

Solution: Cryptographic Checksums

Goal: Data Integrity

Requires secret key ***k*** unknown to browser



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

Session Management

Sessions

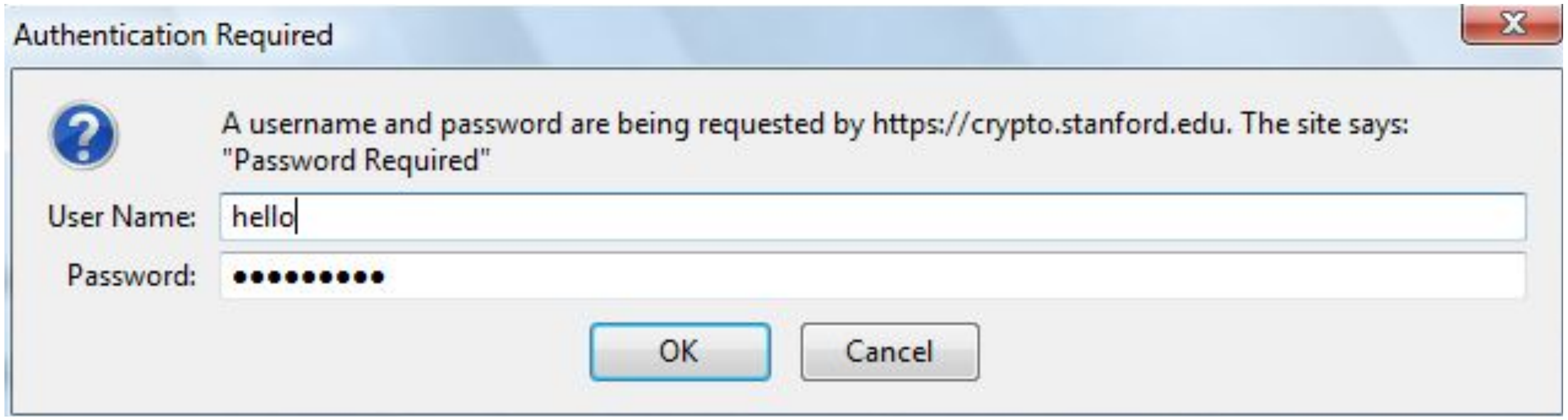
- A sequence of requests and responses from one browser to one (or more) sites
  - Session can be long (Gmail) or short
  - without session mgmt: users would have to constantly re-authenticate
- Session mgmt:
  - Authorize user once;
  - All subsequent requests are bound to user

Pre-history: HTTP auth

HTTP request: GET /index.html

HTTP response contains:

WWW-Authenticate: Basic realm="Password Required“


Authentication Required

A username and password are being requested by https://crypto.stanford.edu. The site says: "Password Required"

User Name: hello

Password: ••••••••••

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)

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)

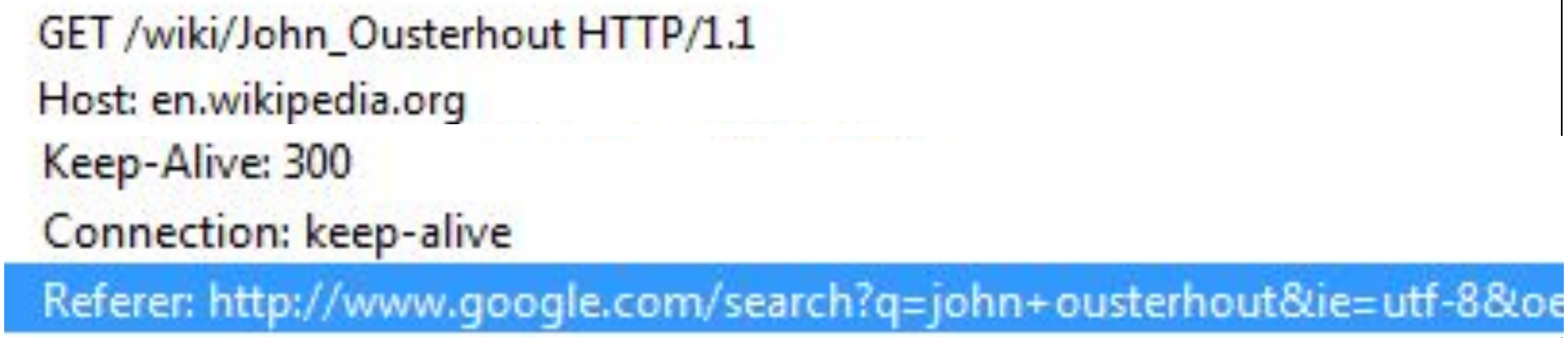
- Embed in all URL links:

  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

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, $\mathbf{MAC_k}$(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)

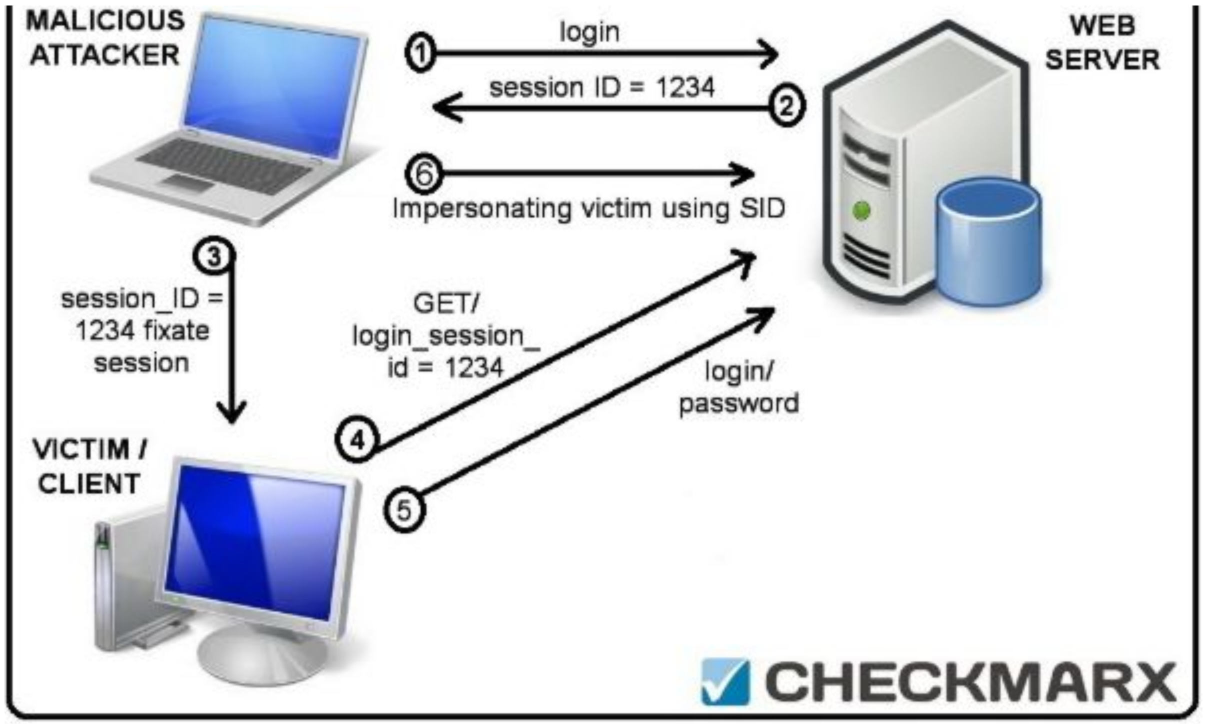
2. Cookie theft

- 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.


MALICIOUS ATTACKER
WEB SERVER
① login
session ID = 1234 ②
⑥ Impersonating victim using SID
③ session_ID = 1234 fixate session
GET/ login_session_ id = 1234 ④
login/ password ⑤
VICTIM / CLIENT
CHECKMARX

1. The malicious attacker connects to the web server.
2. The web server generates a SID (1234) and issues it to the attacker.
3. The attacker then crafts a malicious URL containing the SID and uses various techniques (i.e – phishing) to trick the victim into clicking the URL.
4. The victim clicks on the URL. The server, seeing that an SID already exists, uses it in response to the request.
5. The user logs into the website with his username and password.
6. The attacker now has an authenticated session and can interact with the vulnerable web server on the victim's behalf.

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)