see [PF] 4.1, 4.2, 4.3

Lecture 9: Web Security

9.1 Background

9.2 Vulnerabilities in the “secure” channel

9.3 Mislead the user (address bar)

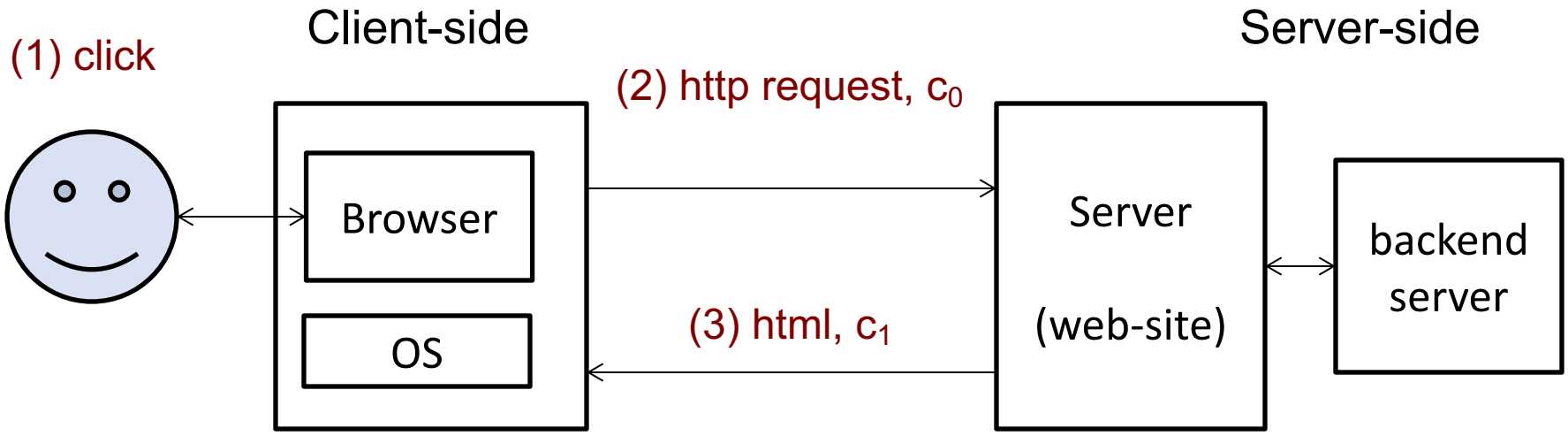
9.4 Cookies and same origin policy

9.5 Cross-site scripting (XSS)

9.6 Cross-site request forgery

9.1 Background

Overview



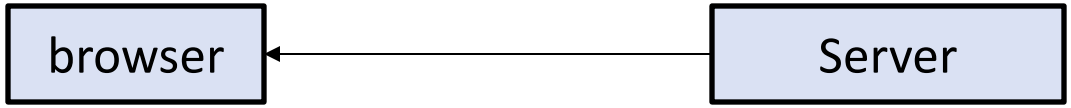
(4) render (including running scripts in the html file)

1. User clicks on a "link". e.g mysoc.nus.edu.sg/
2. A http request is sent to the server. (with cookie c_0 if any)
3. Server constructs and sends a "html" file to the browser, possibly with a cookie c_1
   (the cookie can be viewed as some information the server wants the browser to keep, which to be passed to the server during the next visit. The cookie can be some secrets).
4. The browser renders the html file. The html file describes the layout to be rendered and presented to the user. The html file also instruct the browser to construct and store cookies. (which could be differ from c_1)

In Firefox, right-click, choose "view page source" to view the html file that is sent from the server to the browser.
(or Tool->Wed Developer->Toggle tool)
In Chrome, View->Developer->View Source (or View->Developer->Developer tool)

Closer look into an example.

1) Browser visits Google page. A html file H is sent to the browser. The browser renders H. (Try viewing the raw form of H. In chrome, view->developer->view source. Note that there are many occurrences of the keyword "<script>" which marks the beginning of a script.)



2) User enters the keywords "CS2017 NUS"
3) The browser, by running H, constructs a query. Eg:

https://www.google.com.sg/webhp?sourceid=chrome-instant&ion=1&espv=2&ie=UTF-8#q=CS2107+NUS

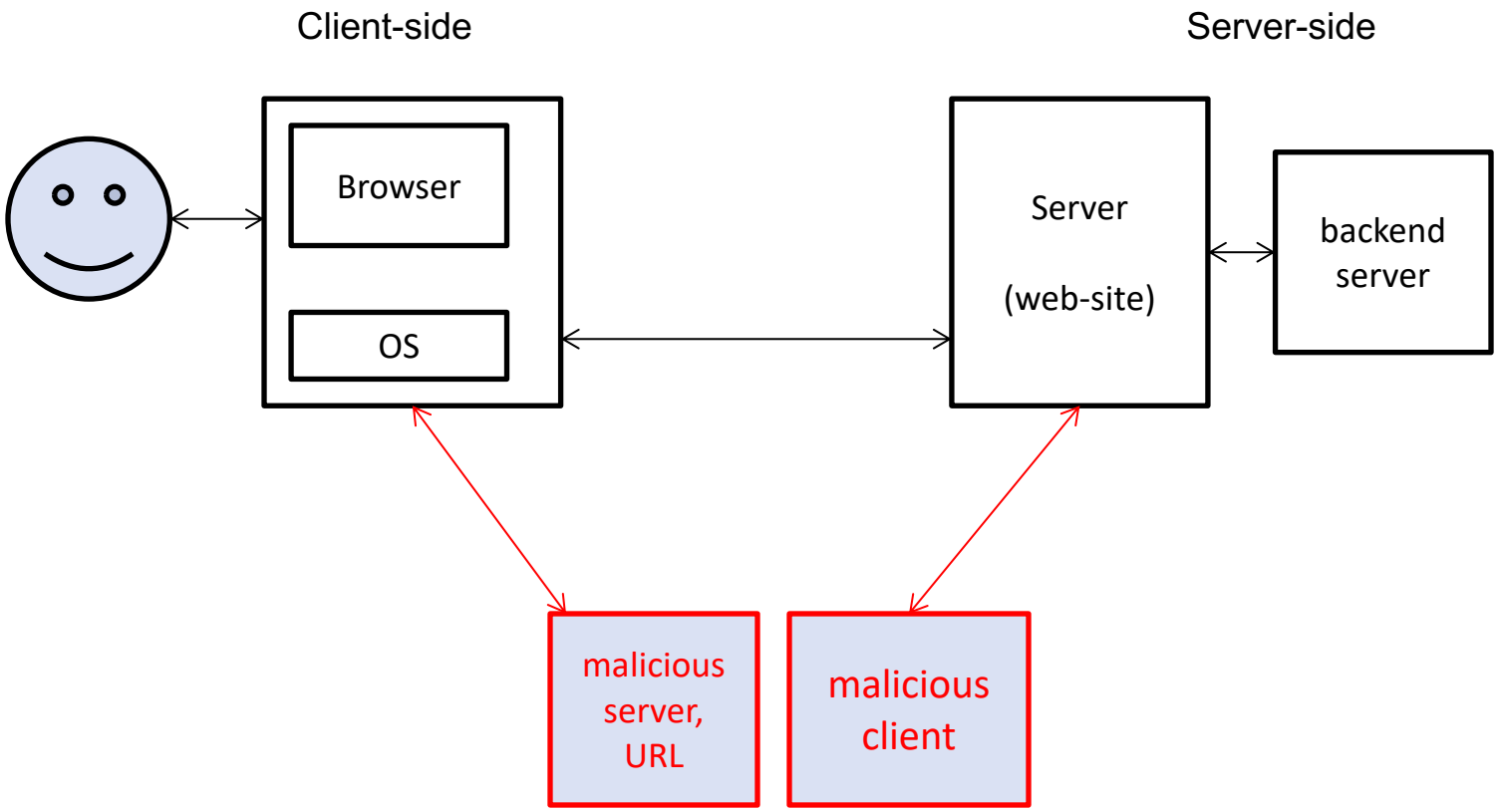
4) Note that additional information is added as parameters. These info is useful for the server. The info could even be in the form of script.
5) The server constructs a reply. (in some cases, the reply contain substrings in (3)).

Complications of Web Security.

- In many OS'es, browsers run with the same privileges as the user. Hence, the browser can access the user's files. (note: this is not the case in Android).
- Browsers support a rich commands/controls set for content providers to render the content.
- Browsers manage sensitive user's info and secrets (stored in cookies).
- At any particular instance, many servers provide the content (*e.g. different advertisements appear at the same time)*
- For enhanced functionality, many browsers support add-ons, plugins, extensions by third parties. *(definitions and differences of add-ons, plugins, extensions are not clear and depends on the browsers.)*
- Users could update content in the server (e.g. forum, names to be displayed).
- More and more sensitive data information is in the web/cloud.

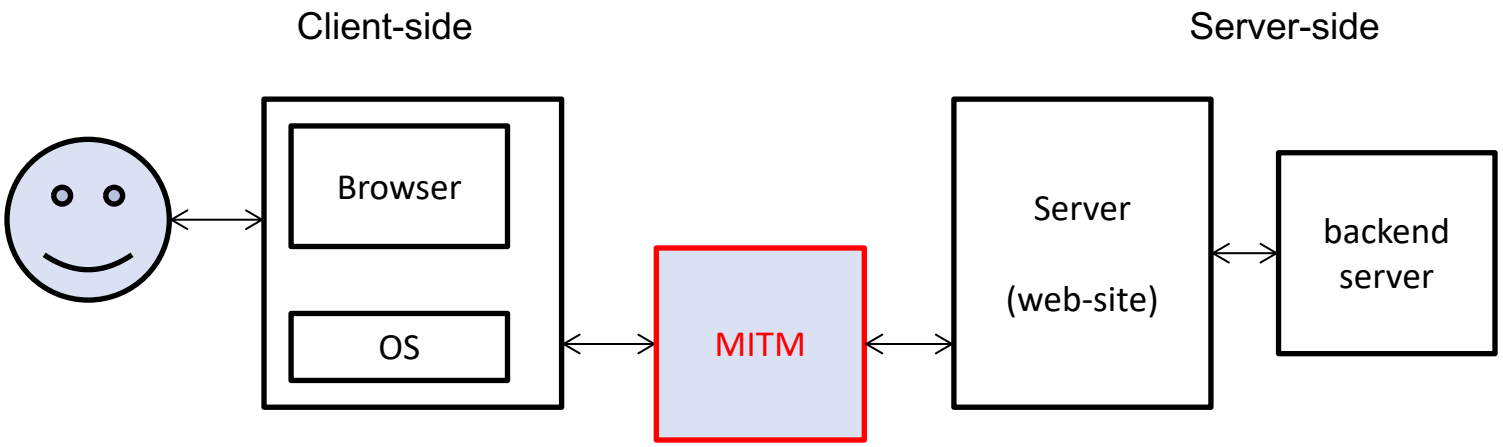
For PC, browser is becoming the main application. (Situation slightly different in mobile os where users also interact with the web using "apps".)

Threat model 1: Attackers as another end systems.



Here, the attacker is just another end system. For e.g. a malicious web-server that the victim is lurked to visit, or attackers who has access to the targeted server.

Threat model 2: Attackers as MITM.



Here, the attacker is a Man-in-the-middle at the IP layer. For e.g., the malicious café-owner who provides the free wifi in many of our previous examples, last hop in the TOR network, etc.

9.2 Vulnerability in the secure communication channel (SSL/TLS)

- Pre-condition of the attack: the attacker is a MITM between the browser and server and able to sniff, spoof packets at the TCP/IP layers. Note that if the connection is HTTPS, such MITM is unable to compromise both confidentiality and authentication. This might not be the case when there are vulnerabilities in the implementation or protocol.

- We have already covered a number of examples: Superfish (presentation), Re-negotiation attack (tutorial), Freak attack.

- Other well-known attacks (not required in this module): BEAST attacks (attack on crypto).

9.3 Misleading the user

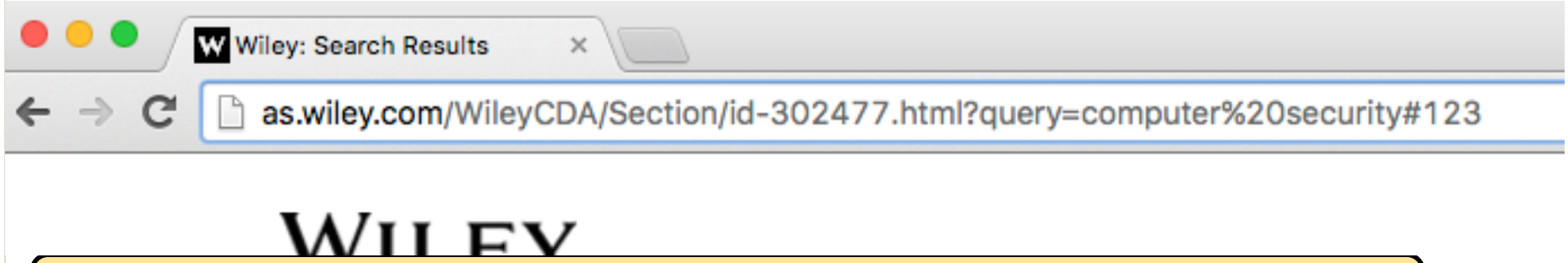
URL (Uniform Resource Locator)

An url consists of a few components. (see https://en.wikipedia.org/wiki/Uniform_Resource_Locator)

1. scheme,
2. authority (a.k.a the hostname) ,
3. path
4. query
5. fragment

e.g.

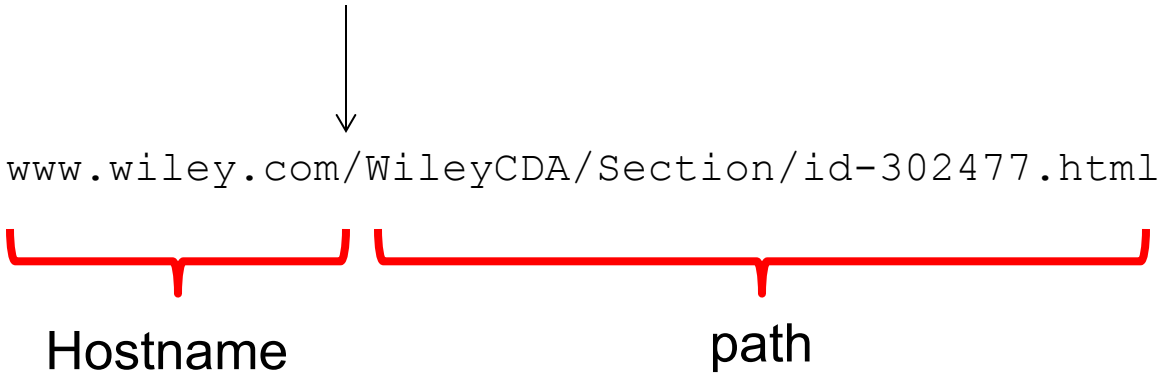
http://www.wiley.com/WileyCDA/Section/id-302477.html?query=computer%20security#12



Question: Why the url is typically displayed using two levels of intensity?

Source of confusion…

Visually, no clear distinction of “Hostname” and “path”. This is because delimiters that separate hostname & the path, can be characters in the path



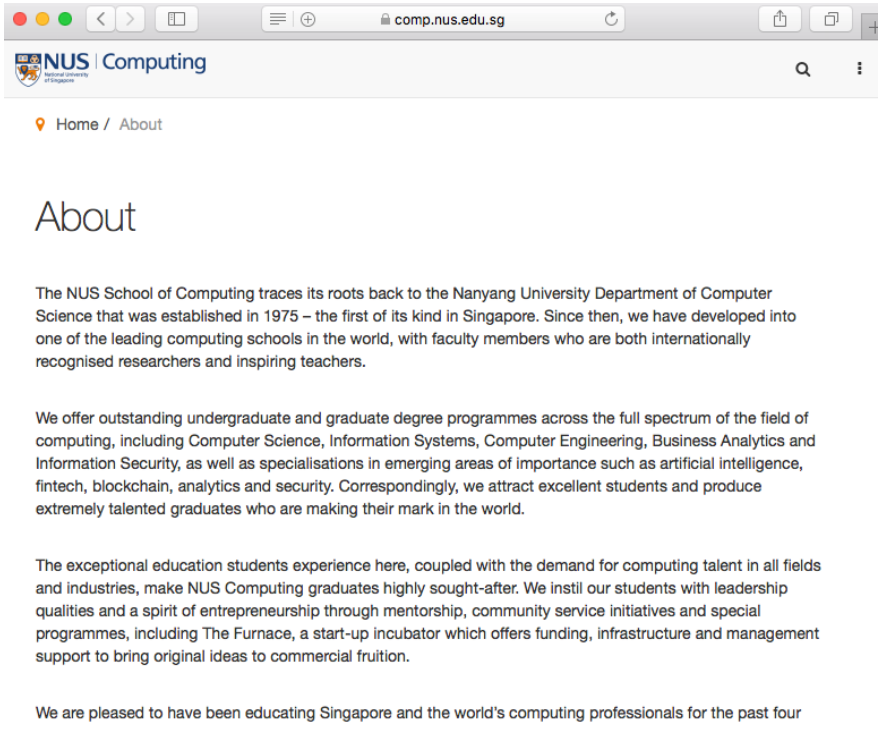
- A malicious website may registered a hostname that contains the targeted hostname with a character that resembles the delimiter “/”

  www.wiley.comlwiley.66785689.in/Section/id-302477.html

The different intensities could help user to spot the attack.

e.g. from phishing email: nuslogin.789greeting.co.uk

- The browser Safari chooses to display the hostname *only* in the address bar. What would be a possible reason behind this design decision?



Address bar spoofing

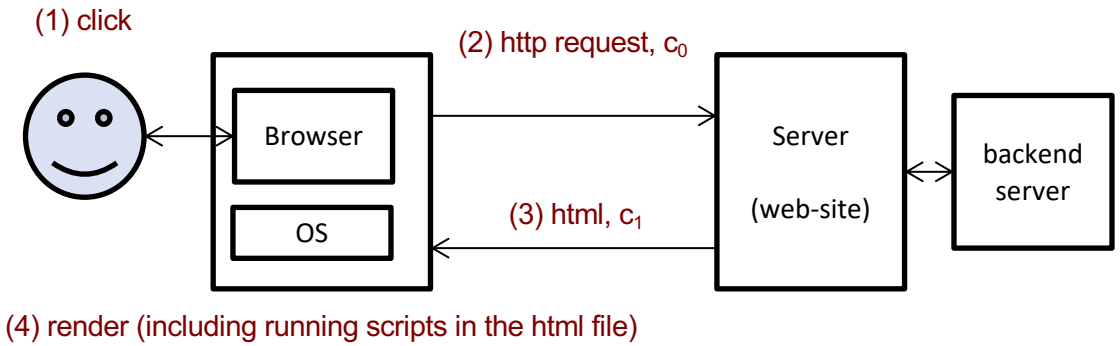
- Address bar is an important component to protect. The address bar is the only indicator of which url the page is rendering. If the address bar can be "modified" by the webpage, an attacker could trick the user to visit a malicious url **X**, while making the user to falsely believe that the url is **Y**.

- A poorly-designed browser may allow attacker to achieve that. E.g. in early design of some browsers, a web page could render objects/pop-up in arbitrary location, and thus allowing a malicious page to overlay spoofed address bar on top of the actual bar. Current version of popular browser have mechanisms to prevent that.

- See a more recent e.g.: **Android Browser All Versions - Address Bar Spoofing Vulnerability - CVE-2015-3830**

http://www.rafayhackingarticles.net/2015/05/android-browser-address-bar-spoofing-vulnerability.html

9.4 Cookie and Same-origin policy



Remark: Many website started to ask consents to keep cookies. Why is this so?

EU's GDPR (General Data Protection Regulation).

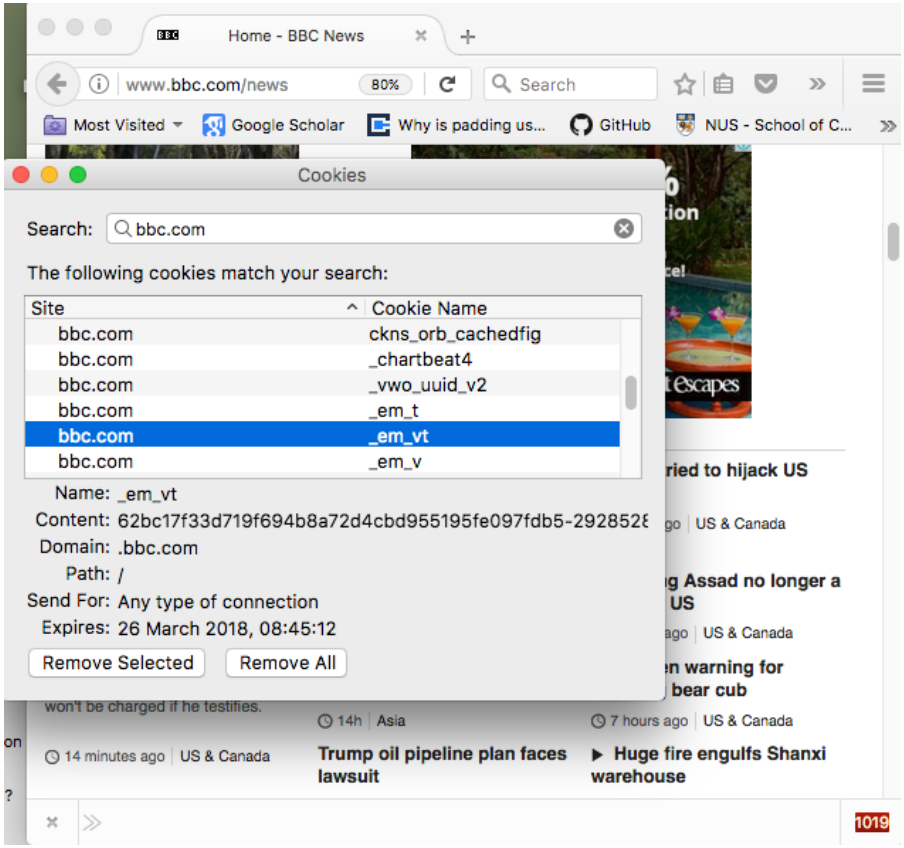
https://www.cookiebot.com/en/gdpr-cookies/

Cookies

- A http cookie is a piece of data sent by the server in the response of a HTTP request. It is under the “Set-Cookie” header field. The browser permanently stores the cookie.
- Whenever a client revisits the website (i.e. submit another http request) the browser automatically sends the cookie back to the server.

Remarks:

- Note that the cookie are sent back only to the “same origin”, in the sense that, it is sent to the server which is the “origin” of the cookie.
- There are also a few types of cookie, eg. session cookies (deleted after the session ends), secure cookies (can only be transmitted over https), etc. See https://en.wikipedia.org/wiki/HTTP_cookie#Session_cookie



Demo on Firefox

- right-click -> view page info -> security -> cookie
- Tools -> web developer -> Developer toolbar

Same-Origin Policy

A “script” which runs in the browser could access cookies. Which scripts can access the cookie? Due to security concern, browser employs this access control policy:

The script in a web page A (identified by URL) can access cookies stored by another web page B (identified by URL), only if both A and B have the same ***origin.***

Origin is defined as the combination of

protocol, hostname, and port number.

The above access policy is simple and thus seeming safe. However, there are a number of complications.

Definition of “same-origin”.

Example (from http://en.wikipedia.org/wiki/Same-origin_policy)

URL’s with the same origin as: http://www.example.com

Compared URL	Outcome	Reason
http://www.example.com/dir/page2.html	Success	Same protocol, host and port
http://www.example.com/dir2/other.html	Success	Same protocol, host and port
http://username:password@**www.example.com** /dir2/other.html	Success	Same protocol, host and port
http://www.example.com:**81**/dir/other.html	Failure	Same protocol and host but different port
https://www.example.com/dir/other.html	Failure	Different protocol
http://**en.example.com**/dir/other.html	Failure	Different host
http://**example.com**/dir/other.html	Failure	Different host (exact match required)
http://**v2.www.example.com**/dir/other.html	Failure	Different host (exact match required)
http://www.example.com:**80**/dir/other.html	Depends	Port explicit. Depends on implementation in browser.

Limitations:

- *Confusing definition*: There are many exceptions, and exceptions of exceptions. Very confusing and thus prone to errors.
- Different browsers may adopt different definitions.

E.g. of Cookie application: Token-based authentication

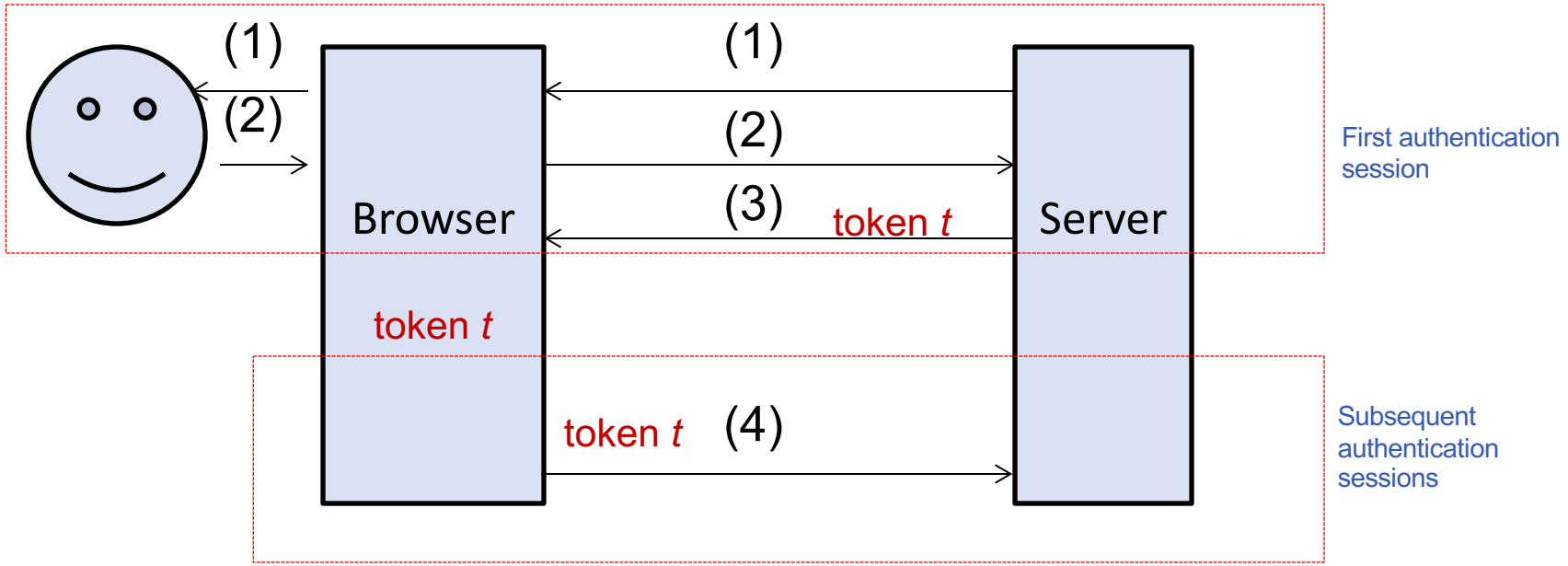
Cookie is useful. Here is one application for Single-Signed-On. *(This example appeared in the tutorial on renegotiation attack)*

To ease user tedious task of repeated login, many websites use "token-based" authentication. That is,

1. After a user ***A*** is being authenticated (for e.g. password verified), the server sends a value ***t***, know as token, to the user.
2. In subsequent connection, whoever presents the correct value is accepted as the authentic user ***A***. (note that user does not has to perform the tedious password authentication again).

Remarks

- Token typically has an expiry date.
- An identification of the session can be used as the token. So the token is also called SID.
- In web applications, the token is often stored in the cookie.



(1) Authentication challenge (e.g. asking for password).

(2) Authentication Response that involves the user.

(3) Server sends a token *t*. Browser keeps the token

(4) Browser presents the token *t*. Server verifies the token.

We assume that the communication channel is secure. (i.e. all communication over HTTPS (with server authenticated) and the HTTPS is free from vulnerabilities).

Choices of token

- Suppose *t* is a randomly chosen number, then the server needs to keep a table storing all the tokens it has issued. To avoid storing the table, one could use
  - (secure version) A message authentication code (MAC). The token ***t*** consists of two parts: a randomly chosen value, or meaningful information like the expiry date, concatenated with the mac computed using the server secret key.

    e.g ***t*** = "alicetan:16/04/2015:adc8213kjd891067ad9993a"
  - (insecure version) the cookie is some meaningful information concatenated with a sequence number that can be predicted.

    e.g t= "alicetan:16/04/2015:128829"

- For both methods, when the server finds out that the token is not in the correct format (or not the correct mac), the server rejects. The first method relies on the security of mac, while the second method relies on obscurity – attackers don't know the format.
- The second method is insecure because an attacker, who knows how the token is generated (for e.g. by observing its own token), can forge it. This further illustrates the weakness of security by obscurity.

9.5 Cross Site Scripting (XSS) Attacks

Background

In some websites, if the browser sends a text that contains a substring ***s,*** the replying html sent by the server would also contain the same substring ***s.***

E.g.

- Enter a wrong address.

  http://www.comp.nus.edu.sg/nonsense_test

- Search for a book in library

  http://nus.preview.summon.serialssolutions.com/#!/search?q=heeheeheee

What if the string ***s*** contains a script?

e.g. http://www.comp.nus.edu.sg/**<script>alert("heehee!");</script>**

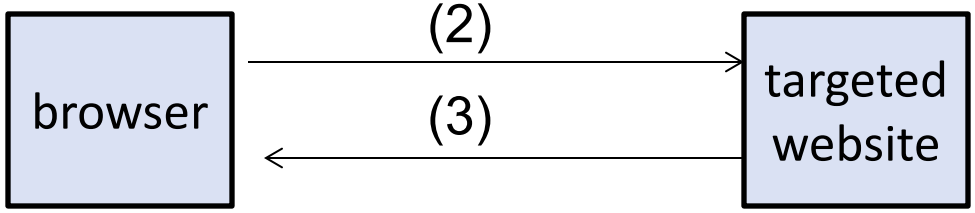
(the above attack won't work as the server replaces the special character "<" by &lt)

Attack

1. Attacker tricks a user to click on an url, which contains the target website, and a malicious script ***s.*** e.g the link could be sent via email with "click me", or a link in a malicious website.
2. The request is sent to the server.
3. The server constructs a response html. The response contains the script ***s***.
4. The browser renders the html page and runs the script ***s***

(1) click me

the url of "click me" contains a script s



(4) browser runs the script s

why this is an attack?

The malicious script could

1. deface the original webpage;
2. steal cookie.

- The control that protects access to the cookie is the same-origin policy.
- Now, since the malicious script is coming from the same-origin (the victim clicked on it..), it can now access cookie previously sent by the website.
- This is yet another example of *privilege escalation*. A malicious script has elevated privilege to read the cookie.

This attack exploits the client's trusts of the server.

types of XSS

- Reflection (aka non-persistent). The attack described before.

- Stored XSS (aka persistent). The script ***s*** is stored in the targeted website. For instance, in forum page.

Defense

- Most defense rely on mechanisms carried out in the server-side. That is, the server filters and removes malicious script in the request while constructing the response page.

  However, this is not a fool proof method.

9.6 Cross Site Request Forgery (XSRF)

- Aka “sea surf”, cross-site reference forgery, session riding.
- This is the reverse of XSS. It exploits the server’s trust of the client. Previous attack of XSS exploits the client’s trust of the server.

Example:

- Suppose a client A is already authenticated by a targeted website S, say `www.bank.com` and the site keeps a cookie as “token”.
- The attacker B tricks A to click on a url of S. The url maliciously requests for a service, say transferring $1000 to the attacker Bob’s

  www.bank.com/transfer?account=Alice&amount=1000&to=Bob

- The cookie will also be automatically sent to S which is sufficient to convince S that A is already being authenticated. Hence the transaction will be carried out.

See https://en.wikipedia.org/wiki/Cross-site_request_forgery

Defense

Relatively easier to prevent compare to XSS.

Include authentication information in the request. For e.g.

www.bank.com/transfer?account=Alice&amount=1000&to=Bob&**Token=xxk34n890ad7casdf897e324**

9.7 Other Attacks

Misconfiguration, searching for filename…

Other attacks and terminologies

Find out more about these terminologies:

- Drive-by-Download.
- Web Bug (aka web beacon, tracking bug, tag, page tag).
- Clickjacking (User Interface redress attack)

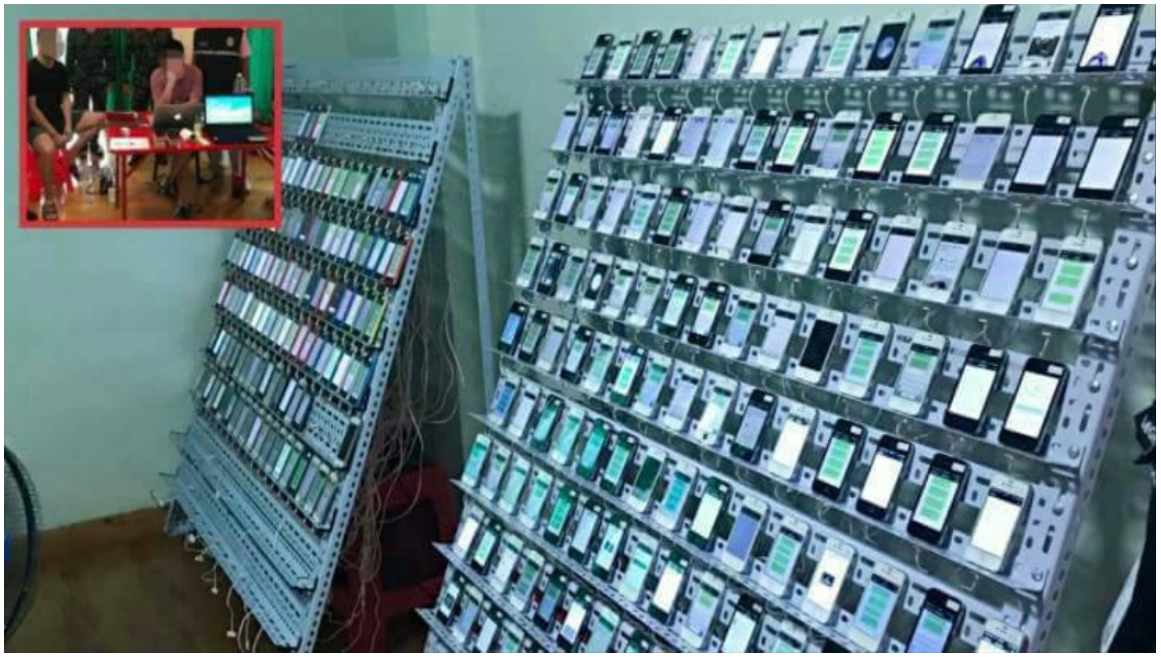
https://www.owasp.org/index.php/Clickjacking

- CAPTCHA
- Click fraud

Question: Could merely visiting a malicious web-site (for e.g. wrongly clicked on a phishing email) subjected to attack?

Common simple implementation mistakes

- Authentication/filtering at the client side.
- Security credential embedded in the public web pages.
- Server's secrets stored in cookies.
- Configuration errors.
- URL as secrets (which is acceptable and commonly used), but without adequate protection of filenames.



https://motherboard.vice.com/en_us/article/43yqdd/look-at-this-massive-click-fraud-farm-that-was-just-busted-in-thailand