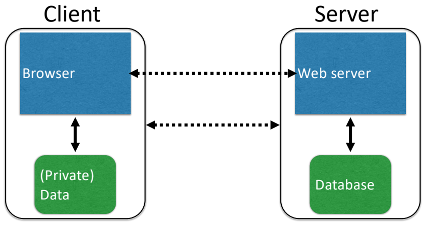
ECE/CS230
Computer Systems Security

Charalambos (Harrys) Konstantinou

https://sites.google.com/view/ececs230kaust

Web

The web, basically



(Much) user data is part of the browser

DB is a separate entity, logically (and often physically)

Interacting with web servers

Resources **which are identified by a** ***URL***

(Universal Resource Locator)

`https://www.kaust.edu.sa/en/study/faculty/charalambos-konstantinou`

Protocol

`ftp`
`https`

Hostname/server

Translated to an IP address by DNS
(e.g., `128.8.127.3`)

Path to a resource

static content

i.e., a fixed file returned by the server

Interacting with web servers

Resources **which are identified by a** ***URL***

(Universal Resource Locator)

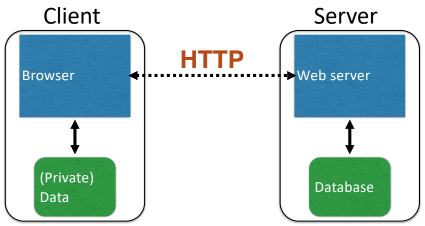
Path to a resource

`http://facebook.com/delete.php?f=joe123&w=16`

Arguments

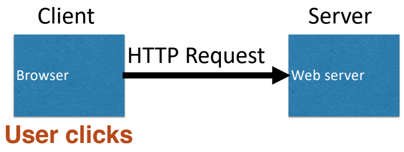
Here, the file `delete.php` is dynamic content
i.e., the server generates the content on the fly

Basic structure of web traffic



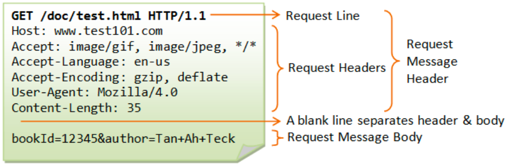
- HyperText Transfer Protocol (**HTTP**)
  - An “application-layer” protocol for exchanging data

Basic structure of web traffic



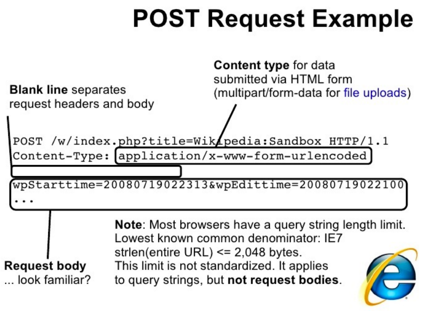
- Requests contain:
  - The **URL** of the resource the client wishes to obtain
  - **Headers** describing what the browser can do
- Request types can be **GET** or **POST**
  - **GET**: all data is in the URL itself
  - **POST**: includes the data as separate fields

HTTP GET requests



```
GET /doc/test.html HTTP/1.1
Host: www.test101.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Content-Length: 35

bookId=12345&author=Tan+Ah+Teck
```

HTTP POST requests



POST Request Example

```
POST /w/index.php?title=Wikipedia:Sandbox HTTP/1.1
Content-Type: application/x-www-form-urlencoded

wpStarttime=20080719022313&wpEdittime=20080719022100
...
```

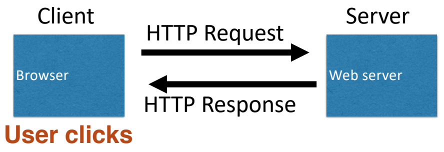
Content type for data submitted via HTML form (multipart/form-data for file uploads)

Blank line separates request headers and body

Request body ... look familiar?

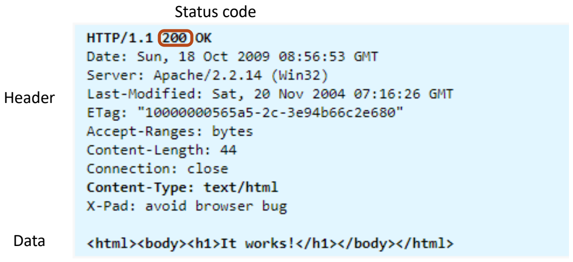
Note: Most browsers have a query string length limit. Lowest known common denominator: IE7 strlen(entire URL) <= 2,048 bytes. This limit is not standardized. It applies to query strings, but **not request bodies**.

Basic structure of web traffic



- **Responses** contain:
  - **Status** code (https://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html)
  - **Headers** describing what the server provides
  - **Data**
  - **Cookies** (much more on these later)
    - Represent *state* the server would like the browser to store

HTTP responses



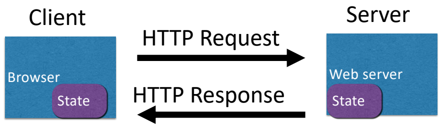
```
HTTP/1.1 200 OK
Date: Sun, 18 Oct 2009 08:56:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Sat, 20 Nov 2004 07:16:26 GMT
ETag: "10000000565a5-2c-3e94b66c2e680"
Accept-Ranges: bytes
Content-Length: 44
Connection: close
Content-Type: text/html
X-Pad: avoid browser bug

<html><body><h1>It works!</h1></body></html>
```

Adding state to the web

HTTP is *stateless*

- The lifetime of an HTTP session is typically:
  - Client connects to the server
  - Client issues a request
  - Server responds
  - Client issues a request for something in the response
  - …. repeat ….
  - Client disconnects
- No direct way to ID a client from a previous session
  - So why don’t you have to log in at every page load?

Maintaining State

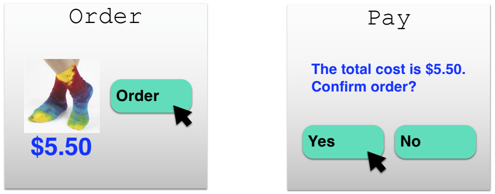


- **Web application maintains *ephemeral* state**
- Server processing often produces intermediate results
- Send state to the client
- Client returns the state in subsequent responses

Two kinds of state: **hidden fields**, and **cookies**

Ex: Online ordering

socks.com/order.php socks.com/pay.php



Separate page

Ex: Online ordering

What’s presented to the user

pay.php

```
<html>
<head> <title>Pay</title> </head>
<body>

<form action=“submit_order” method=“GET”>
The total cost is $5.50. Confirm order?
<input type=“hidden” name=“price” value=“5.50”>
<input type=“submit” name=“pay” value=“yes”>
<input type=“submit” name=“pay” value=“no”>

</body>
</html>
```

Ex: Online ordering

The corresponding backend processing

```
if(pay == yes && price != NULL)
{
    bill_creditcard(price);
    deliver_socks();
}
else
    display_transaction_cancelled_page();
```

Anyone see a problem here?

Ex: Online ordering

Client can change the value!

```
<html>
<head> <title>Pay</title> </head>
<body>

<form action=“submit_order” method=“GET”>
The total cost is $5.50. Confirm order?
<input type=“hidden” name=“price” value=“0.01”
<input type=“submit” name=“pay” value=“yes”>
<input type=“submit” name=“pay” value=“no”>

</body>
</html>
```

Solution: *Capabilities*

- Server maintains ***trusted*** state
  - Server stores intermediate state
  - Send a pointer to that state (**capability**) to client
  - Client **references** the capability in next response
- Capabilities should be **hard to guess**
  - Large, random numbers
  - To prevent illegal access to the state

Using capabilities

Client can no longer change price

```
<html>
<head> <title>Pay</title> </head>
<body>

<form action=“submit_order” method=“GET”>
The total cost is $5.50. Confirm order?
<input type=“hidden” name=“sid” value=“781234”>
<input type=“submit” name=“pay” value=“yes”>
<input type=“submit” name=“pay” value=“no”>

</body>
</html>
```

Using capabilities

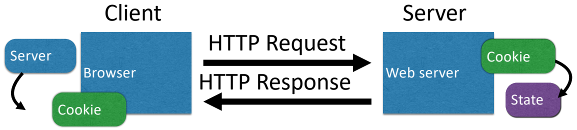
The corresponding backend processing

```
price = lookup(sid);
if(pay == yes && price != NULL)
{
    bill_creditcard(price);
    deliver_socks();
}
else
    display_transaction_cancelled_page();
```

But we don’t want to use hidden fields all the time!

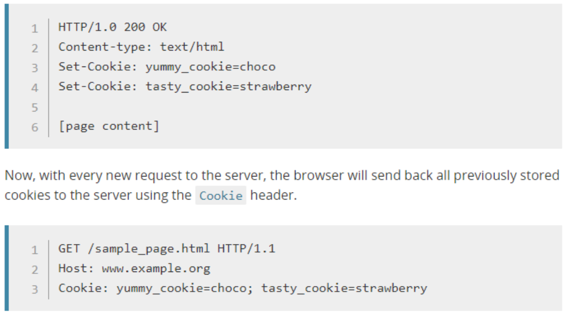
- Tedious to maintain on all the different pages
- Start all over on a return visit (after closing browser window)

Statefulness with Cookies



- Server maintains trusted state
  - Indexes it with a **cookie**
  - Sends cookie to the client, which stores it
  - Client returns it with subsequent queries to same server

Cookies



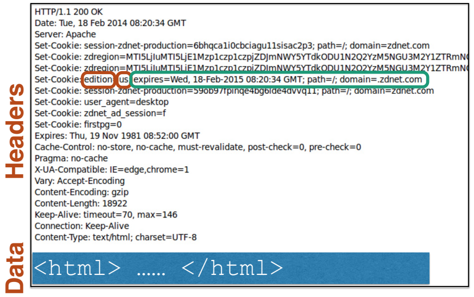
```
HTTP/1.0 200 OK
Content-type: text/html
Set-Cookie: yummy_cookie=choco
Set-Cookie: tasty_cookie=strawberry

[page content]
```

Now, with every new request to the server, the browser will send back all previously stored cookies to the server using the `Cookie` header.

```
GET /sample_page.html HTTP/1.1
Host: www.example.org
Cookie: yummy_cookie=choco; tasty_cookie=strawberry
```

Cookies are key-value pairs

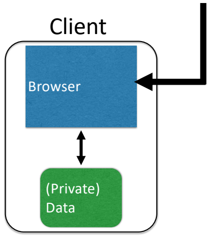
Set-Cookie:key=value; options; ….



```
HTTP/1.1 200 OK
Date: Tue, 18 Feb 2014 08:20:34 GMT
Server: Apache
Set-Cookie: session-zdnet-production=6bhqca1i0cbciagu11sisac2p3; path=/; domain=zdnet.com
Set-Cookie: zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN
Set-Cookie: zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN
Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT; path=/; domain=.zdnet.com
Set-Cookie: session-zdnet-production=59ob97fpinqe4bg6lde4dvvq11; path=/; domain=zdnet.com
Set-Cookie: user_agent=desktop
Set-Cookie: zdnet_ad_session=f
Set-Cookie: firstpg=0
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
X-UA-Compatible: IE=edge,chrome=1
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 18922
Keep-Alive: timeout=70, max=146
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8

<html> …… </html>
```

Cookies

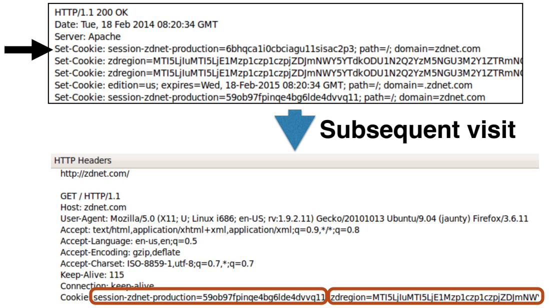
```
Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT; path=/; domain=.zdnet.com
```



Semantics

- Store "us" under the key "edition"
- This value was no good as of Feb 18, 2015
- This value should only be readable by any domain ending in `.zdnet.com`
- This should be available to any resource within a subdirectory of `/`
- Send the cookie with any future requests to `<domain>/<path>`

Requests with cookies



```
HTTP/1.1 200 OK
Date: Tue, 18 Feb 2014 08:20:34 GMT
Server: Apache
Set-Cookie: session-zdnet-production=6bhqca1i0cbciagu11sisac2p3; path=/; domain=zdnet.com
Set-Cookie: zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN
Set-Cookie: zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN
Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT; path=/; domain=.zdnet.com
Set-Cookie: session-zdnet-production=59ob97fpinqe4bg6lde4dvvq11; path=/; domain=zdnet.com
```

Subsequent visit

```
HTTP Headers
  http://zdnet.com/

  GET / HTTP/1.1
  Host: zdnet.com
  User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11) Gecko/20101013 Ubuntu/9.04 (jaunty) Firefox/3.6.11
  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
  Accept-Language: en-us,en;q=0.5
  Accept-Encoding: gzip,deflate
  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
  Keep-Alive: 115
  Connection: keep-alive
  Cookie session-zdnet-production=59ob97fpinqe4bg6lde4dvvq11 zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNW
```

Why use cookies?

- **Session identifier**
  - After a user has authenticated, subsequent actions provide a cookie
  - So the user does not have to authenticate each time
- **Personalization**
  - Let an anonymous user customize your site
  - Store language choice, etc., in the cookie

Why use cookies?

- **Tracking users**
  - Advertisers want to know your behavior
  - Ideally build a profile *across different websites*
  - Visit the Apple Store, then see iPad ads on Amazon?!
  - How can site B know what you did on site A?

- Site A loads an ad from Site C
- Site C maintains cookie DB
- Site B also loads ad from Site C

- **“Third-party cookie”**
- **Commonly used by large ad networks (doubleclick)**

http://live.wsj.com/video/how-advertisers-use-internet-cookies-to-track-you

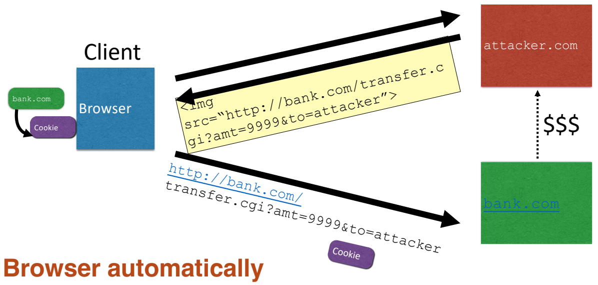
Cross-Site Request Forgery (CSRF)

URLs with side effects

`http://bank.com/transfer.cgi?amt=9999&to=attacker`

- GET requests often have **side effects on server state**
  - Even though they are not supposed to
- What happens if
  - the **user is logged in** with an active session cookie
  - a **request is issued for the above link?**
- How could you get a user to visit a link?

Exploiting URLs with side effects


Client
bank.com
Cookie
Browser
attacker.com
<img src="http://bank.com/transfer.cgi?amt=9999&to=attacker">
$$$
http://bank.com/transfer.cgi?amt=9999&to=attacker
Cookie
bank.com

Browser automatically visits the URL to obtain what it believes will be an image

Cross-Site Request Forgery

- Target: User who has an account on a vulnerable server
- Attack goal: Send requests to server *via the user's browser*
  - Look to the server like the user intended them
- Attacker needs: Ability to get the user to "click a link" crafted by the attacker that goes to the vulnerable site
- Key tricks:
  - Requests to the web server have predictable structure
  - Use e.g., `<img src=…>` to force victim to send it

Variation: Login CSRF

- Forge login request to honest site
  - Using ***attacker's*** username and password
- Victim visits the site under attacker's account
- What harm can this cause?

Google

PayPal