The Web...

About Project 2...
To be Released Real Soon Now

- Not only is it to teach you the difficulty of implementing crypto systems
  - Real-world cruel grading would be "1 security bug -> 0 credit!"
  - This is the "Kobyashi Maru" project: I don't want anybody to get 100%!
    You learn from failure, not just success
- But to also test/teach by doing some important software engineering skills
  - Using a safe language (Go)
  - Developing good tests
    - Go has an excellent testing infrastructure
  - Design ***first!***
  - Serialization & Deserialization of Data
    - How to go from program internal representations to blobs-of-data and back...

Don't write code first, ***design*** first!

- Read all parts...
- Write your design document ***first***
  - When you ask the TAs for help, they are instructed to start with your design document!
- Good design makes the project easy
  - My 100% solution for the slightly simpler version last year is <400 LOC...
  - But of course my initial 100% solution actually had a bug!
- Couple more hints on the design...
  - What do HMAC and Argon2 do?
  - When in doubt there is the universal CS solution:
    add another layer of indirection!

The Data Storage Problem:

- You have some ugly internal data structures...
  - It doesn't really matter what it is, but lots of pointers, arrays, and other ugly things...
- You need to convert it to a single string of bits
  - For storage, transmission, whatever
  - And go the other way, turn it back into the data structure
- This is called ***serialization*** and ***deserialization***: Turning your data into a sequence of bits

Paradigm #1:
Do It Manually...

- The C/C++ traditional world
  - Also very common in network programming
  - Python's **`struct`** module as well
- Define a byte order
  - If you need to go between different instruction sets!
- Pack/unpack data into bytes
  - If you may have endianness, use **`ntoh`** and **`hton`**
- Generally safe when adversaries hand you data...
  - Assuming you don't do classic memory screwups that is
- Generally a PitA!

Paradigm #2:
Java `serialize` & python `pickle`...

- Nice and convenient:
  - Allows you to dump and restore arbitrary objects
- But ***horribly dangerous!***
  - If an adversary provides an object, it can deserialize to ***basically anything they want***!
- Never, ***ever ever ever*** use these if you are communicating outside your own program!
  - They are not suitable for a malicious environment!
- Common programmer F-up: Use serialize or pickle ***thinking*** it will only have trusted input...
  - And then another programmer creates a path where the function is reachable from untrusted input
- So add these to your "search" list for 'you just got handed a new project'
  - Along with `system()` and direct calls to SQL databases, along with unsafe C string operations

Paradigm #3:
Google Protocol Buffers

- Provides a compiler to compile code to pack/unpack structures
  - Highly efficient binary encoding
  - Available for C++, python, java, go, ruby, Objective-C, C#
- Safe, but requires using an external compiler to create code to pack/unpack structures
  - And its not human readable in the slightest

Paradigm #4: XML and JSON

- Text based formats
  - Human readable-ish:
    Don't underestimate the value in being able to read your computer data directly!
- JSON is small and simple
  - Just a few types in key/value pair structures
- XML is grody and complex...
  - XML parsers tend to have bugs.
- Both are less compact
  - Lots of useless text as they are ASCII format, not binary
- So we provide you with Json marshal/unmarshal!
  - Hint: You can coerce the bytes to a string if you want to print what is being written!

Personal Preference:
When in doubt, use json.

- It is cross platform like Google Protocol Buffers
  - But doesn't require any external compiler support
- It is simple
- It is "geek readable"
  - Especially if you turn on pretty-printing to add newlines
- It is really easy for web applications to use
  - JavaScript directly recognizes it! "JavaScript Object Notation"
- Space overhead pretty much goes away with compression
  - And the web compresses everything pretty much by default

Web Security:
Web History...

- Often one needs to start with history to realize why present day is so incredibly **`fsck`**'ed...
  - And the web, is indeed, strongly **`fsck`**'ed up
- We saw that on the back end on Monday...
  - **`system`** and SQL were designed for non-secure environments

The Prehistory Idea: Memex...

- Observation from 1945:
  We need a conceptual way to organize data
  - A reference library may have a ton of stuff, but how do you find something?
  - Microfilm is even more compact
  - E.g. a single microfiche card is a 105mm x 148mm piece of film
    - That can hold photos of 100 ***pages*** of text!
- But how do we find and understand things?
  - Idea from Vannevar Bush:
    WW2 head of the primary military R&D office
- https://en.wikipedia.org/wiki/Memex

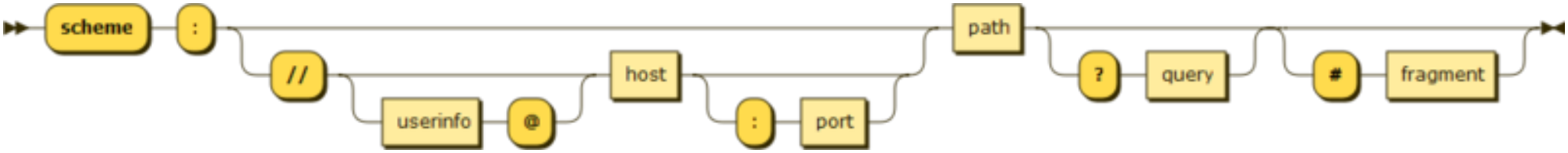
The Memex...

- A big integrated desk that can store and access microfilm...
  - The most compact storage available at the time
- Idea #1: "Trails"
  - Rather than just view pages of data linearly...
  - You could follow a "trail": A linear path through an arbitrary sequence of actual film
    - This is what we'd now call a "hyperlink":
      Refer to another piece of data by location
  - You could also create "personal trails": your own custom path for
- Idea #2: "Upload data"
  - It would also include a photographic hood:
    You could then add it to the collection in the Memex
- Never actually ***built*** but conceptually very important
- Note that it was ***only*** about accessing data, not code!

HTML, HTTP, and URLs

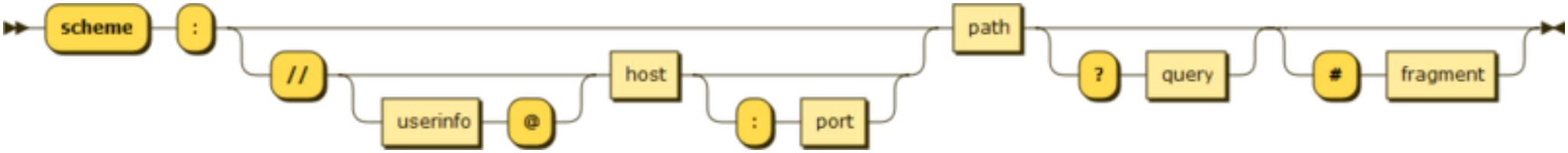
- HTML: Hyper Text Markup Language
  - A text-based representation with "tags" (e.g. <TITLE>this is a title</TITLE>, <A HREF="....">, <IMG SRC="...">)
- HTTP: Hyper Text Transfer Protocol
  - A (cleartext) protocol used to fetch HTML and other documents from a remote server (the "Web server")
- URLs: Uniform Resource Locators
  - A text format for identifying where a piece of data is in the world...

The URL, which is a URI (Uniform Resource Identifier



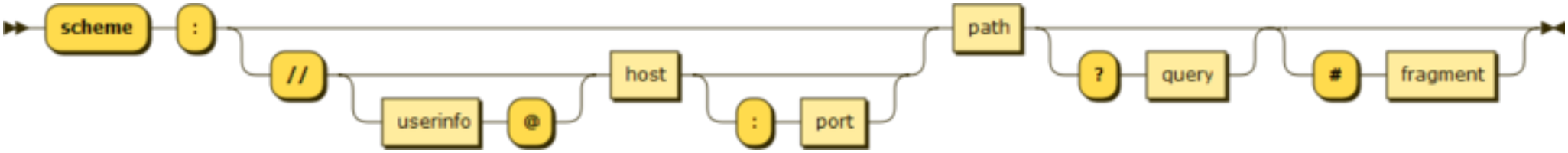
- https://en.wikipedia.org/wiki/Uniform_Resource_Identifier
- Scheme: What protocol to use, e.g.
  - "ftp" File Transfer Protocol
  - "http" Hyper Text Transfer Protocol
  - "https" Encrypted HTTP
  - "file" A local file on the network

The URL Continued: Location



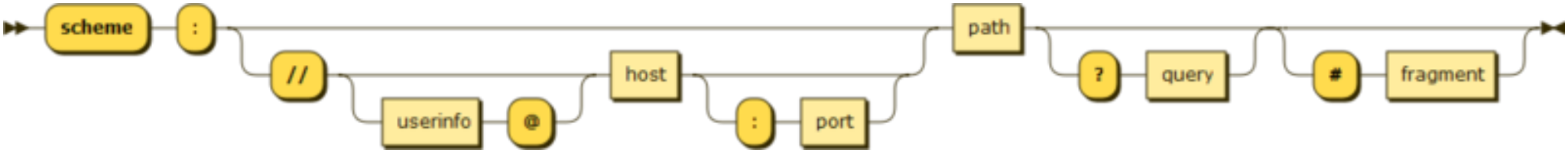
- Remote location: "//.."
  - Username followed by @ if there is one (optional)
  - Host, the remote computer (mandatory if remote)
    - Either a hostname or an IP (Internet Protocol) address
  - Remote port (if different from the default, optional)
    - Networking speaks in terms of remote computers and ports
- This is "where to find the remote computer"

The URL Continued: Path



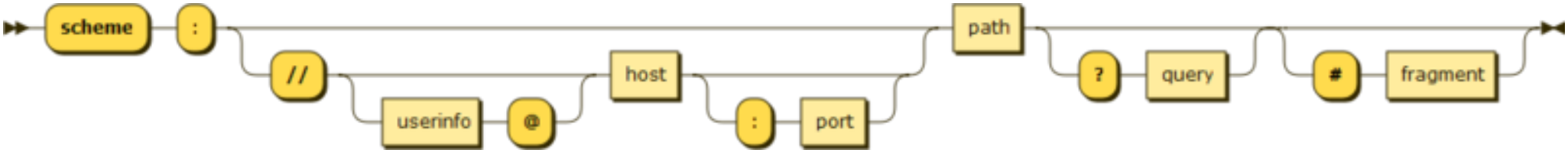
- Path is mandatory and starts with /
  - "/" alone is the "root" of the directory tree, and must appear
- Directory entries are separated with /
  - Unix style rather than Window's style \
- Sent to the remote computer to tell it where to look for the file starting at its own root directory for data that it is sharing

The URL Continued: Query



- Query is optional and starts with ?
  - Need to encode ? as %3f if elsewhere in the URI
- This is sent to the remote server
  - Commonly designed as a set of key/value pairs... EG, Name=Nick&Role=SuperGenius
  - Remote server will then interpret the data appropriately

The URL Continued: Fragment



- Fragment is optional and starts with #
- This is ***not sent*** to the remote server!
  - Only available to the local content
  - Initially intended just to tell the web browser where to jump to in a document...
  - But now used for JavaScript to have local content in the URL that isn't sent over to the server

URIs are ASCII text

- It is an ASCII (plain text) format: Only 7 bits with "printable" characters
- To encode non-printable characters, spaces, special characters (e.g. ?, #, /) you must "URL encode" as %xx with xx being the hexadecimal value
  - %20 = ' '
  - %35 = '#'
- Can optionally encode normal ASCII characters too!
  - %50 = '2'
- Can make it hard to detect particular problems...
  - EG, /%46%46/etc/password converts to: /../etc/password
    - Will go "above the root" if the web server is misconfigured to grab the password file!

HTTP
(Hypertext Transfer Protocol)

A common data communication protocol on the web

HTTP

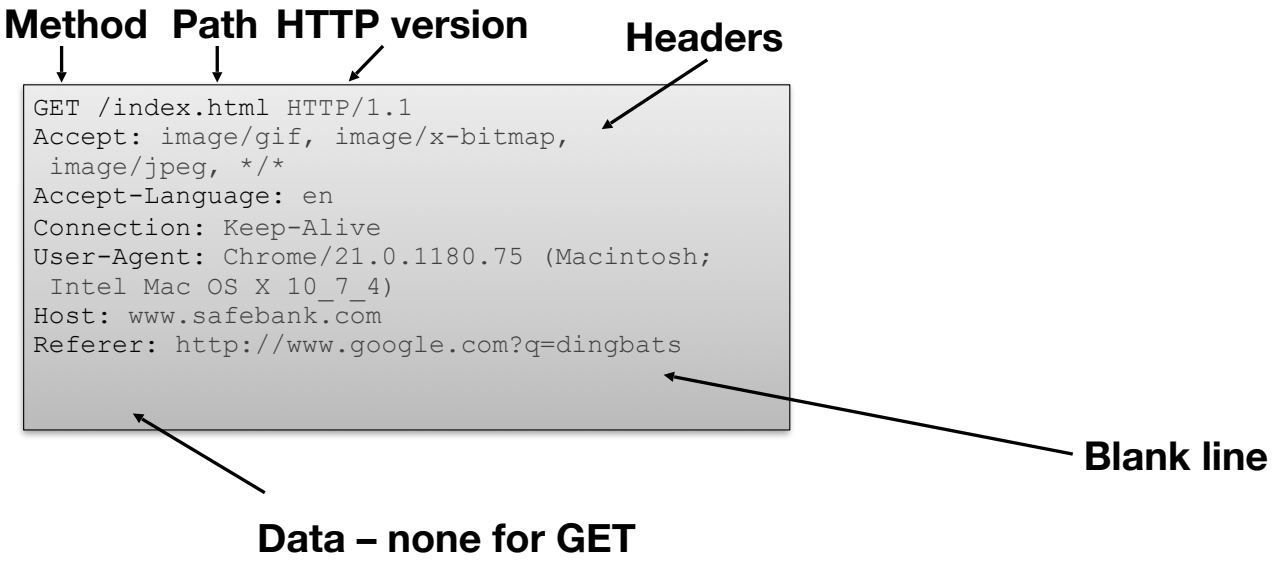

CLIENT BROWSER
WEB SERVER
safebank.com
safebank.com/account.html
SAFEBANK
Alice Smith
Accounts
Bill Pay
Mail
Transfers
HTTP REQUEST:
GET /account.html HTTP/1.1
Host: www.safebank.com
HTTP RESPONSE:
HTTP/1.0 200 OK
<HTML> . . . </HTML>

HTTP Request

GET: no side effect (supposedly, HA)

POST: possible side effect, includes additional data

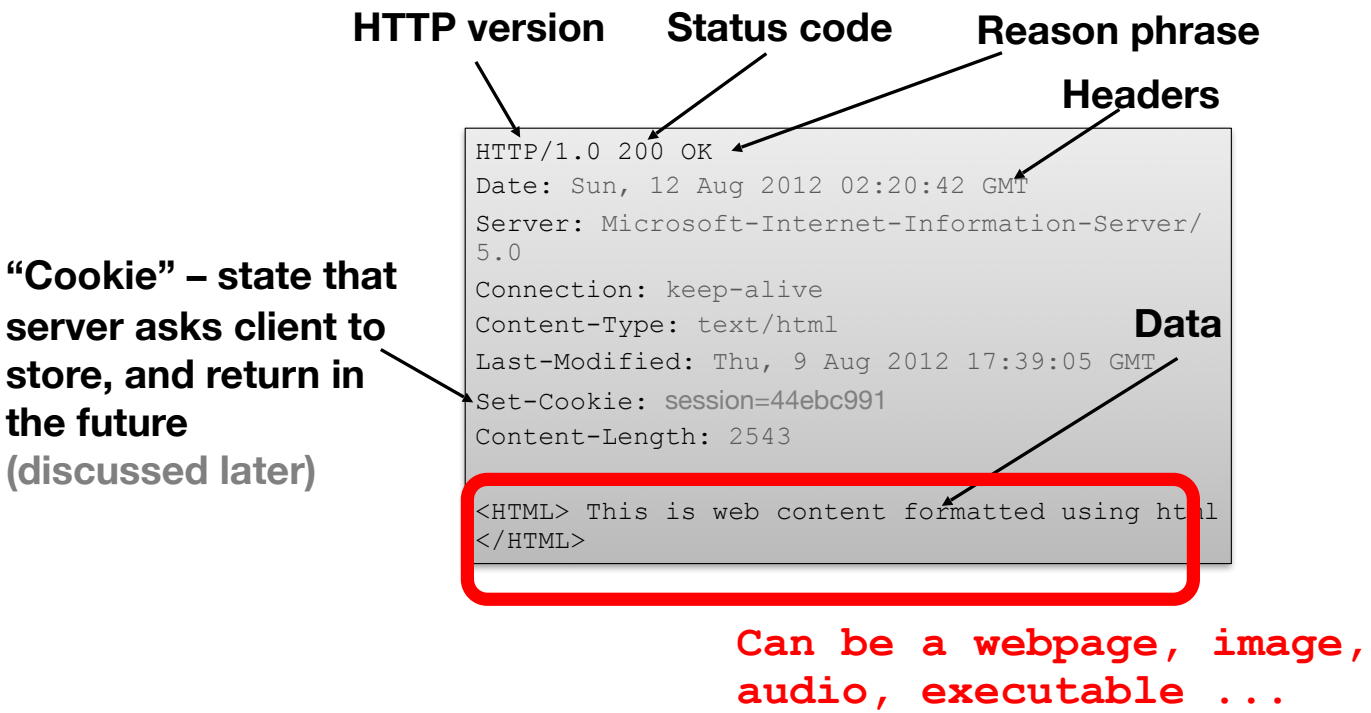
HEAD: only the first part of the content



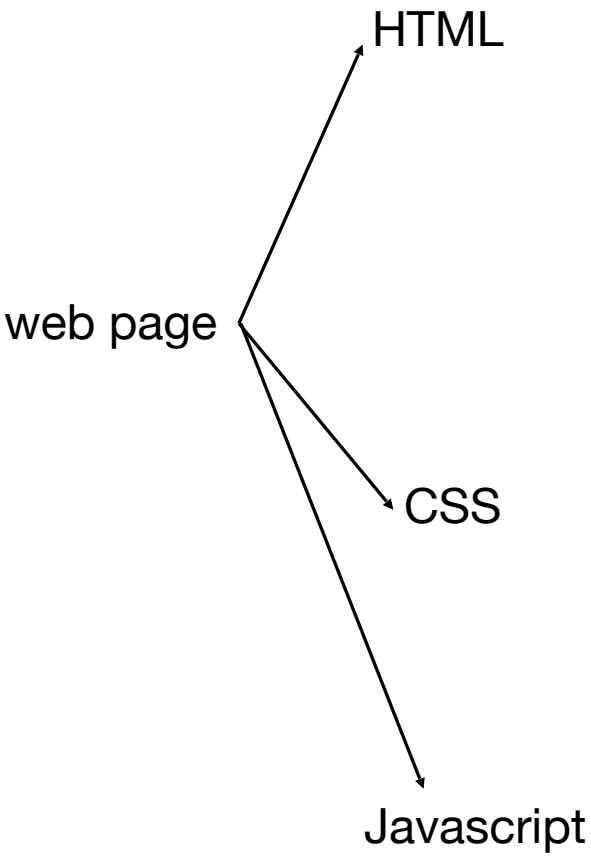
HTTP


CLIENT BROWSER
WEB SERVER
safebank.com
safebank.com/account.html
SAFEBANK
Alice Smith
Accounts
Bill Pay
Mail
Transfers
HTTP REQUEST:
GET /account.html HTTP/1.1
Host: www.safebank.com
HTTP RESPONSE:
HTTP/1.0 200 OK
<HTML> . . . </HTML>

HTTP Response



Web page



HTML

A language to create structured documents
One can embed images, objects, or create interactive forms

index.html

```
<html>
    <body>
        <div>
            foo
            <a href="http://google.com">Go to Google!</a>
        </div>
        <form>
            <input type="text" />
            <input type="radio" />
            <input type="checkbox" />
        </form>
    </body>
</html>
```

CSS (Cascading Style Sheets)

Language used for describing the presentation of a document

index.css

```
p.serif {
font-family: "Times New Roman", Times, serif;
}
p.sansserif {
font-family: Arial, Helvetica, sans-serif;
}
```

Originally There Was Only HTTP...

- It was a way of expressing the text of the documents
  - With other embedded content like images...
- And it was good, but...
- Sun had a programming language called "Java"
  - Designed to compile to an intermediate representation and run on a lot of systems
- They built a web browser that could also fetch and execute Java...
  - But Java was too powerful: It was designed to do everything a host program could do
- So they created a language called "JavaScript"
  - Only thing in common with "Java" is the name and bits of the syntax

Javascript

Programming language used to manipulate web pages. It is a high-level, dynamically typed and interpreted language with support for objects. It is why web sites are now programs running in the browser

Supported by all web browsers

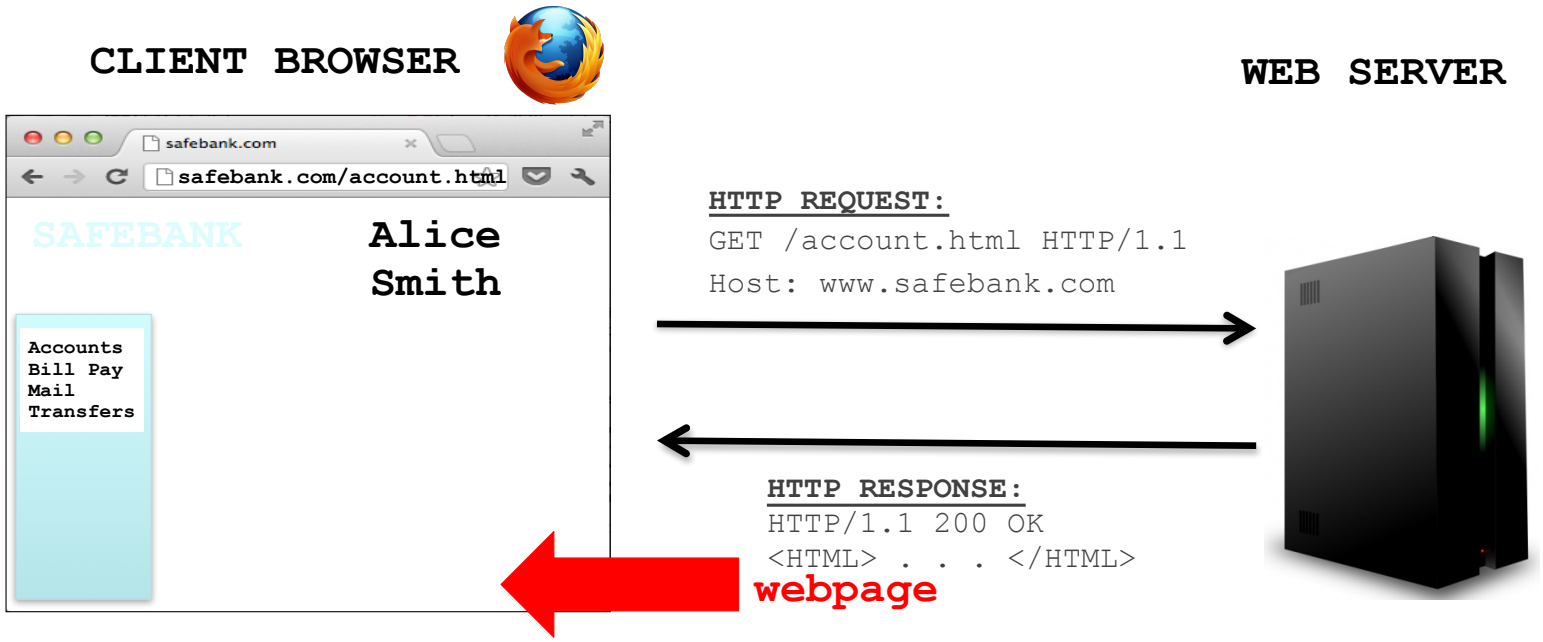
```
<script>
function myFunction()
{    document.getElementById("demo").innerHTML = ”Text
changed.";
}
</script>
```

Very powerful!

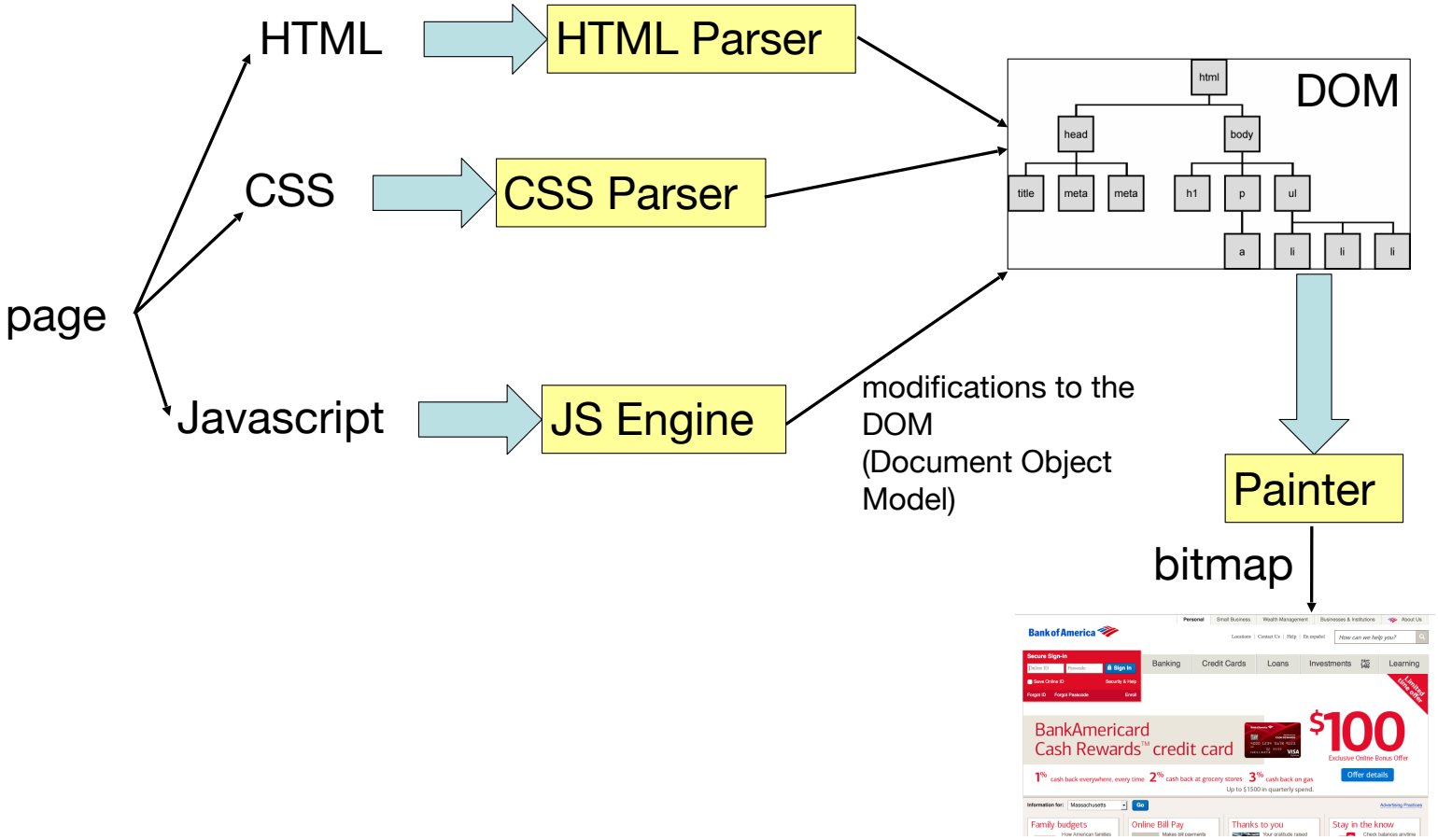
Lots Of Work To Make This Fast...

- These days JavaScript is used just about everywhere
  - So a lot of work goes into making this execute quickly
- Common technique: "Just In Time Compiler"
- Initially interprets JavaScript
- After a function is interpreted enough, convert the function into native machine code
  - So need some memory that is both executable AND writeable...
- Which is why vulnerabilities in the JavaScript interpreter/compiler are so dangerous
  - Attacker is already running code, its just "sandboxed" to limit what it can do
  - Gain an arbitrary read/write primitive:
    EG "use after free" on a JavaScript object
  - Now can have the JavaScript program inspect memory!
  - Breaks ALSR: The attacker's program can examine memory to derandomize things
  - Breaks W^X: Find something in the W&X space to overwrite with the attacker's code...
    - No need to do those silly ROP chains...

HTTP


CLIENT BROWSER
WEB SERVER
safebank.com
safebank.com/account.html
SAFEBANK
Alice
Smith
Accounts
Bill Pay
Mail
Transfers
HTTP REQUEST:
GET /account.html HTTP/1.1
Host: www.safebank.com
HTTP RESPONSE:
HTTP/1.1 200 OK
<HTML> . . . </HTML>
webpage

Page rendering

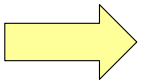


DOM (Document Object Model)

Cross-platform model for representing and interacting with objects in HTML

HTML

```
<html>
    <body>
        <div>
            foo
        </div>
        <form>
            <input type="text" />
            <input type="radio" />
            <input type="checkbox" />
        </form>
    </body>
</html>
```



DOM Tree

```
|-> Document
   |-> Element (<html>)
     |-> Element (<body>)
       |-> Element (<div>)
         |-> text node
       |-> Form
            |-> Text-box
            |-> Radio Button
            |-> Check Box
```

The power of Javascript

Get familiarized with it so that you can think of all the attacks one can do with it.

What can you do with Javascript?

Almost anything you want to the DOM!

A JS script embedded on a page can modify in almost arbitrary ways the DOM of the page.

The same happens if an attacker manages to get you load a script into your page.

w3schools.com has nice interactive tutorials

Example of what Javascript can do…

Can change HTML content:

```
<p id="demo">JavaScript can change HTML content.</p>

<button type="button"
onclick="document.getElementById('demo').innerHTML =
'Hello JavaScript!'">
   Click Me!</button>
```

DEMO from
http://www.w3schools.com/js/js_examples.asp

Other examples

Can change images
Can chance style of elements
Can hide elements
Can unhide elements
Can change cursor...

Basically, can do ***anything it wants*** to the DOM

Another example: can access cookies (Access control tokens)

Read cookie with JS:

```
var x = document.cookie;
```

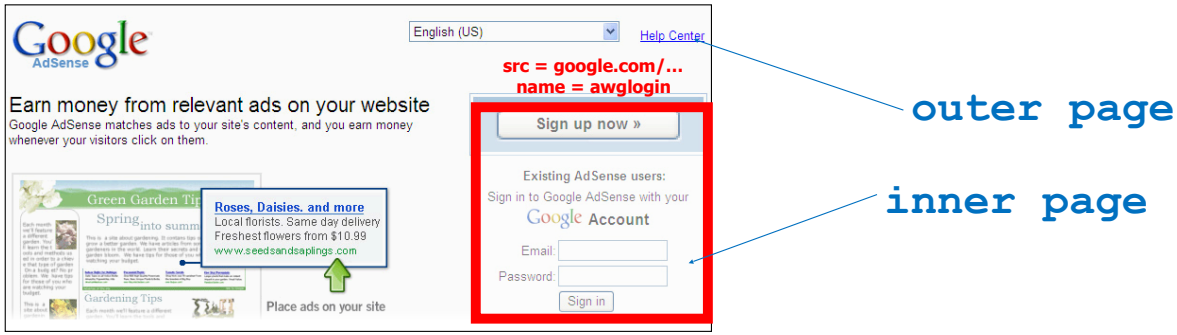
Change cookie with JS:

```
document.cookie = "username=John Smith; expires=Thu, 18
Dec 2013 12:00:00 UTC; path=/";
```

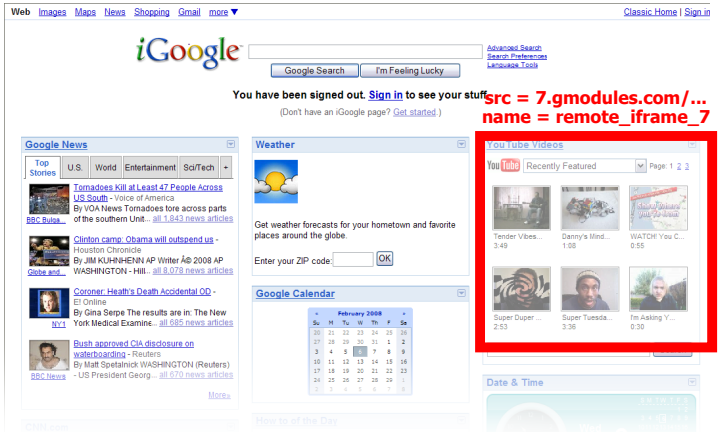
Frames

- Enable embedding a page within a page

```
<iframe src="URL"></iframe>
```



Frames



- Modularity
  - Brings together content from multiple sources
  - Client-side aggregation
- Delegation
  - Frame can draw only inside its own rectangle

Frames

- Outer page can specify only sizing and placement of the frame in the outer page
- Frame isolation: Outer page cannot change contents of inner page; inner page cannot change contents of outer page

Desirable security goals

- ***Integrity***: malicious web sites should not be able to tamper with integrity of our computers or our information on other web sites
- ***Confidentiality***: malicious web sites should not be able to learn confidential information from our computers or other web sites
- ***Privacy***: malicious web sites should not be able to spy on us or our online activities
- ***Availability***: malicious parties should not be able to keep us from accessing our web resources

Security on the web

- Risk #1: we don't want a malicious site to be able to trash files/programs on our computers
  - Browsing to **`awesomevids.com`** (or **`evil.com`**) should not infect our computers with malware, read or write files on our computers, etc...
  - We generally assume an adversary can cause our browser to go to a web page of the attacker's choosing
- Mitigation strategy
  - Javascript is sandboxed: it is ***not allowed*** to access files etc...
  - Browser code tries to avoid bugs:
    - Privilege separation, automatic updates
    - Reworking into safe languages (rust)

Security on the web

- Risk #2: we don't want a malicious site to be able to spy on or tamper with our information or interactions with other websites
  - Browsing to **`evil.com`** should not let **`evil.com`** spy on our emails in Gmail or buy stuff with our Amazon accounts
- Defense: Same Origin Policy
  - An ***after the fact*** isolation mechanism enforced by the web browser

Security on the web

- Risk #3: we want data stored on a web server to be protected from unauthorized access
- Defense: server-side security