CSE509 : Computer System Security

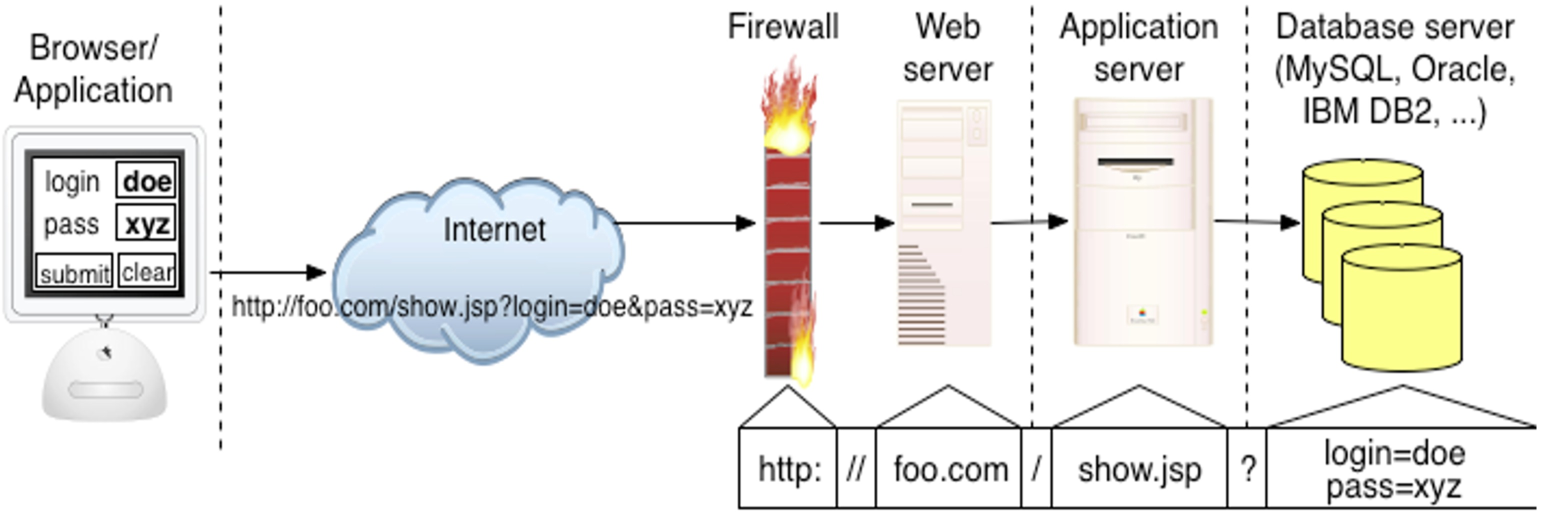
Web Security

Historical Web

- Historically, the web was just a request response protocol
- HTTP is stateless, which means that the server essentially processes a request independent of prior history
- Envisioned as a way for exchanging information

Current Web

- Evolving into a platform for executing programs that support day-to-day tasks
- A lot of state needs to be maintained
- Distributed computation, and trust model



HTTP Requests

- A request has the form:

```
METHOD> /path/to/resource?query_string HTTP/1.1
<header>*
<BODY>
```

- HTTP supports a variety of methods, but only two matter in practice:
  - GET: intended for information retrieval
    - Typically the BODY is empty
  - POST: intended for submitting information
    - Typically the BODY contains the submitted information

Structure of HTTP GET request

- Connect to: www.example.com
  - TCP Port 80 is the default for http, others may be specified explicitly in the URL
- Send: GET /index.html HTTP/1.1
- Server Response:

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Etag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Content-Length: 438
Connection: close Content-Type: text/html;
charset=UTF-8
```

GET with parameters

- GET /submit_order?sessionid=79adjadf888888768&

  pay=yes
  HTTP/1.1
- User Inputs sent as parameters to the request

POST Requests

- Another way of sending requests to HTTP servers
- Commonly used in FORM submissions
- Message written in the BODY of the request
- Sending links with malicious parameter values is difficult when a web site accepts only POST requests.
- But a script running on a malicious web site can as easily send a POST request (as a GET request) to another web site.

HTTP Responses

- A response has the form

```
HTTP/1.1 <STATUS CODE> <STATUS MESSAGE>
<header>*
<BODY>
```

- Important response codes:
  - 2XX: Success, e.g. 200 OK
  - 3XX: Redirection, e.g. 301 Moved Permanently
  - 4XX: Client side error, e.g. 404 Not Found
  - 5XX: Server side error, e.g. 500 Internal Server Error

HTTP response

```
HTTP/1.1 200 OK
Date: Tue, 21 Oct 2014 16:21:44 GMT
Server: Apache/2.2.25 (Unix) mod_ssl/2.2.25 OpenSSL/1.0.1h PHP/5.2.17
Last-Modified: Tue, 21 Oct 2014 15:37:09 GMT
ETag: "3aaa5c-850-505f09ab7f211"
Accept-Ranges: bytes
Content-Length: 2128
Content-Type: text/html

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html><head>

  <title>Is The Internet On Fire?</title>
  <meta http-equiv="content-type" content="text/html; charset=UTF-8">
  <link rev="made" href="mailto:jschauma@netmeister.org">
```

Cookies

- HTTP is stateless, therefore client needs to remember state and send this with every request
- Cookies are the common way of keeping state

- Client:

```
GET /index.html HTTP/1.1
Host: www.example.org
```

- Server:

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: sess-id=3773777adbdad

(content of page)
```

Cookies…

- Browsers send cookie with every subsequent request

```
GET /spec.html HTTP/1.1
Host: www.example.org
Cookie:  sess-id=3773777adbdad
```

- Now server can look up stored state through sess-id
- Alternative to cookies: hidden form fields.

What Are Cookies Used For?

- Authentication
  - The cookie proves to the website that the client previously authenticated correctly
- Personalization
  - Helps the website recognize the user from a previous visit
- Tracking
  - Follow the user from site to site; learn his/her browsing behavior, preferences, and so on

Sessions

- As long as different users have different session identifiers (present in their cookies), the web server will be able to tell them apart
  - Regardless of their IP address
- When users delete their cookies, the browsers no longer send out the appropriate session identifier, and thus the web server "forgets" about them

Session Identifiers

- Long pseudo-random strings
- Unique per visiting client
- Each identifier is associated with a specific visitor
  - ID A -> User A
- As sensitive as credentials (per session)

One missing piece

- We can create websites
- And we can have state, enabling us to have a personalized web
  - Banking ,Email, Social networks, etc.

- But our pages are still static
  - The server sent some HTML, the browser drew it on the screen, and that's it

JavaScript

- "The world's most misunderstood programming language"
- Language executed by the Web browser
  - Scripts are embedded in webpages
  - Can run before HTML is loaded, before page is viewed, while it is being viewed, or when leaving the page
- Used to implement "active" webpages and Web applications
- A potentially malicious webpage gets to execute some code on user's machine

JavaScript History

- Developed by Brendan Eich at Netscape
  - Scripting language for Navigator 2
- Later standardized for browser compatibility
  - ECMAScript Edition 3 (aka JavaScript 1.5)
- Related to Java in name only
  - Name was part of a marketing deal
  - "Java is to JavaScript as car is to carpet"
- Various implementations available
  - SpiderMonkey, RhinoJava, others

Aside: Java Security

- With binary code, memory and type safety issues complicate the problem of untrusted code
- Java and Javascript rely on safe languages
  - avoid low-level issues arising in C, C++ and binary code
    - No buffer overflows.
  - code can be created and executed only through sanctioned pathways, e.g., class loader
  - access-control restrictions associated with classes will be strictly and fully enforced
    - Can't circumvent public/private restrictions by casting etc.

Java Vs JavaScript

- Java originally developed to support "active web pages"
  - Applets were intended to allow local execution of untrusted code
  - Security was achieved by restricting access to local resources, e.g., files
- Drawbacks
  - did not provide good integration with the browser environment
  - focus was more on (OS) integrity rather than confidentiality
  - these factors led to the development of Javascript
  - Today, Adobe flash is closer in many ways to Java than Javascript

Java Vs JavaScript

- Javascript takes a different approach
  - Language safety is still the basis
  - Use this basis to provide safe interface to the browser environment
  - The security model is object-oriented
  - What are the browser resources, which ones are accessible to untrusted code
- Browser is the platform, not the underlying OS
- It is not about whether untrusted code can access local files, but whether the browser permits it to do so ("trusted dialogs")
- Cookie-based model of browser security evolved in conjunction with Javascript, leading to excellent support for the same.

Common Uses of JavaScript

- Page embellishments and special effects
- Dynamic content manipulation
- Form validation
- Navigation systems
- Hundreds of applications
    - Google Docs, Google Maps, dashboard widgets in Mac OS X, Philips universal remotes …

JavaScript in Webpages

- Embedded in HTML as a <script> element
  - Written directly inside a <script> element
    - <script> alert("Hello World!") </script>
  - In a file linked as src attribute of a <script> element

    <script type="text/JavaScript" src="functions.js"></script>
- Event handler attribute

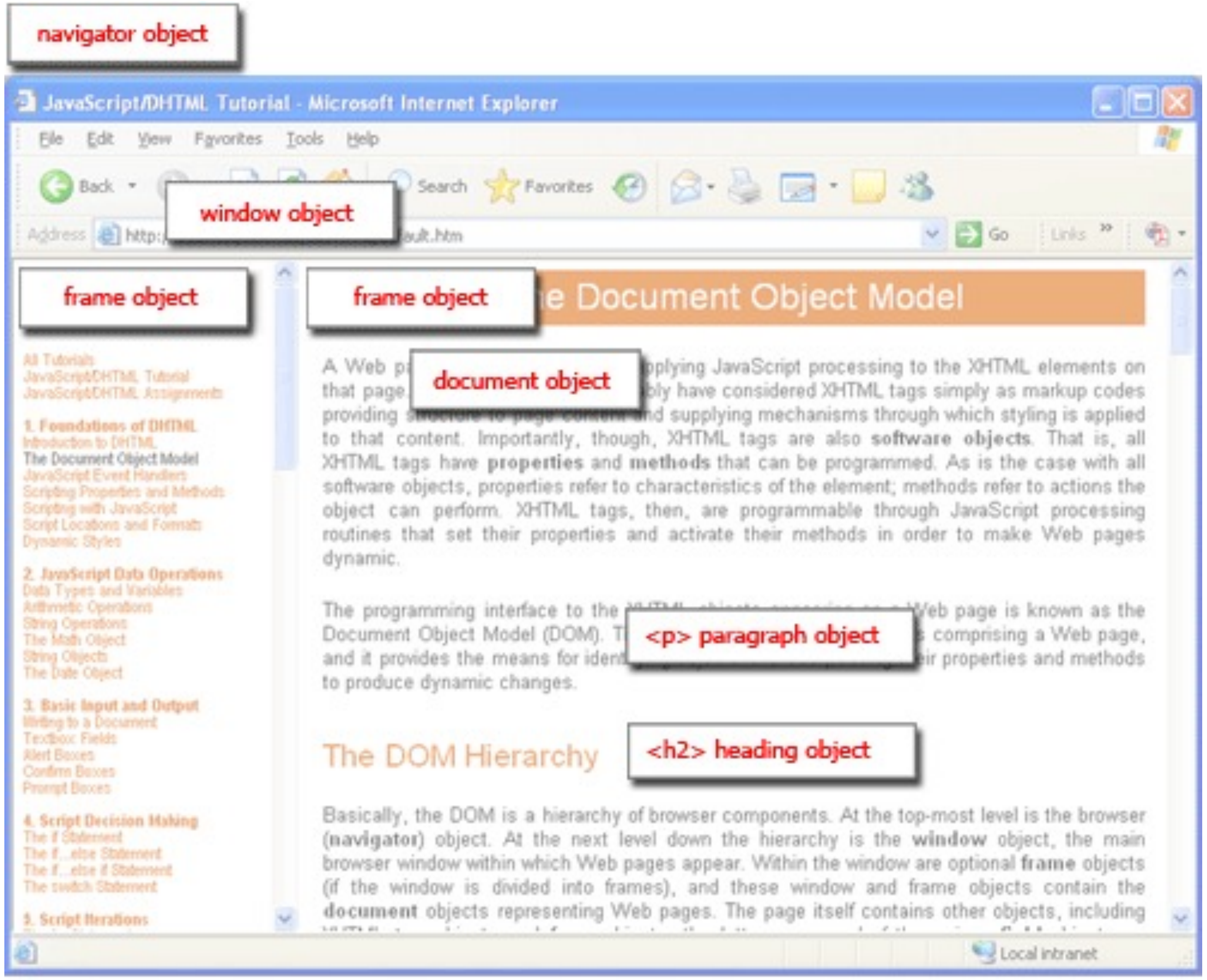
    <a href=http://www.yahoo.com onmouseover="alert('hi');">
- Pseudo-URL referenced by a link

    <a href="javascript: alert('You clicked');">Click me</a>

Document Object Model (DOM)

- HTML page is structured data
- DOM is object-oriented representation of the hierarchical HTML structure
  - Properties: document.alinkColor, document.URL, document.forms[ ], document.links[ ], ...
  - Methods: document.write(document.referrer)
    - These change the content of the page!
- Also Browser Object Model (BOM)
  - Window, Document, Frames[], History, Location, Navigator (type and version of browser)

Browser and Document Structure



All Tutorials
JavaScript/DHTML Tutorial
JavaScript/DHTML Assignments

1. Foundations of DHTML
Introduction to DHTML
The Document Object Model
JavaScript Event Handlers
Scripting Properties and Methods
Scripting with JavaScript
Script Locations and Formats
Dynamic Styles

2. JavaScript Data Operations
Data Types and Variables
Arithmetic Operations
String Operations
The Math Object
String Objects
The Date Object

3. Basic Input and Output
Writing to a Document
Textbox Fields
Alert Boxes
Confirm Boxes
Prompt Boxes

4. Script Decision Making
The if Statement
The if...else Statement
The if...else if Statement
The switch Statement

5. Script Iterations

e Document Object Model

A Web p pplying JavaScript processing to the XHTML elements on that page. bly have considered XHTML tags simply as markup codes providing nd supplying mechanisms through which styling is applied to that content. Importantly, though, XHTML tags are also **software objects**. That is, all XHTML tags have **properties** and **methods** that can be programmed. As is the case with all software objects, properties refer to characteristics of the element; methods refer to actions the object can perform. XHTML tags, then, are programmable through JavaScript processing routines that set their properties and activate their methods in order to make Web pages dynamic.

The programming interface to the Veb page is known as the Document Object Model (DOM). T comprising a Web page, and it provides the means for ident ir properties and methods to produce dynamic changes.

The DOM Hierarchy

Basically, the DOM is a hierarchy of browser components. At the top-most level is the browser (**navigator**) object. At the next level down the hierarchy is the **window** object, the main browser window within which Web pages appear. Within the window are optional **frame** objects (if the window is divided into frames), and these window and frame objects contain the **document** objects representing Web pages. The page itself contains other objects, including

Reading Properties with JavaScript

Sample
Sample script

```
1. document.getElementById('t1').nodeName
2. document.getElementById('t1').nodeValue
3. document.getElementById('t1').firstChild.nodeName
4. document.getElementById('t1').firstChild.firstChild.nodeName
5. document.getElementById('t1').firstChild.firstChild.nodeValue
```

HTML

```
<ul id="t1">
<li> Item 1 </li>
</ul>
```

- Example 1 returns "ul"
- Example 2 returns "null"
- Example 3 returns "li"
- Example 4 returns "text"
  - A text node below the "li" which holds the actual text data as its value
- Example 5 returns " Item 1 "

Page Manipulation with JavaScript

- Sample
- Some possibilities
  - createElement(elementName)
  - createTextNode(text)
  - appendChild(newChild)
  - removeChild(node)

```
HTML
 <ul id="t1">
 <li> Item 1 </li>
 </ul>
```

- Example: add a new list item

```
var list = document.getElementById('t1')
var newitem = document.createElement('li')
var newtext =
document.createTextNode(text)
list.appendChild(newitem)
newitem.appendChild(newtext)
```

All the functional pieces are in place

- Now we can create personalized and dynamic websites. Yay!
- But what about security?
  - How do we stop websites from snooping around in each other's business?

Goals of Web Security

- Safely browse the Web
  - A malicious website cannot steal information from or modify legitimate sites or otherwise harm the user…
    - … even if visited concurrently with a legitimate site - in a separate browser window, tab, or even iframeon the same webpage
    - Based on Same Origin Policy (SOP)
  - A malicious website cannot steal or modify informationon the local machine, nor can it interact in any way withlocal applications
    - Based on JavaScript safety and web browser designand implementation (Browser security)

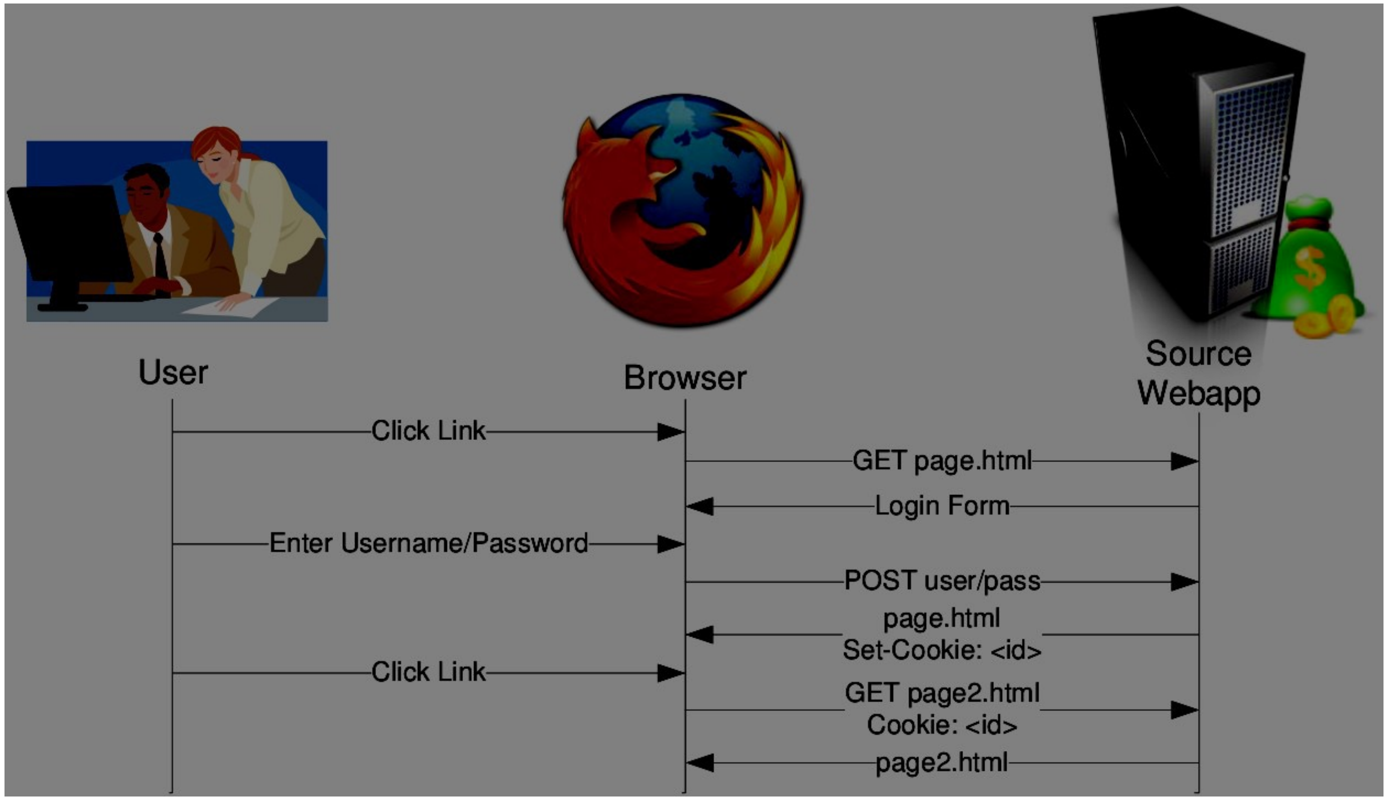
Web Security Concerns

- Web Security is concerned with ensuring the following 3 properties for web applications:
  - Authentication: securely identify users on top of HTTP, which is a stateless protocol.
  - Confidentiality: protect any sensitive data that websites serve to the browser from other websites, and the user's own sensitive data outside the browser from any website.
  - Integrity: ensure that the data and the code served to users cannot be tampered with.

Authentication Methods

- HTTP authentication: username/passwd supplied in HTTP header
- Cookie authentication (most common):
  - username/password (login credentials) requested via a HTML form
  - server checks the credentials and then sets a cookie that identifies the user and his/her successful login
  - Browser returns this cookie with each subsequent request
- Hidden-form authentication
  - Similar to cookie authentication, but the server includes the session info in a hidden form field.

Cookie-Based Authentication

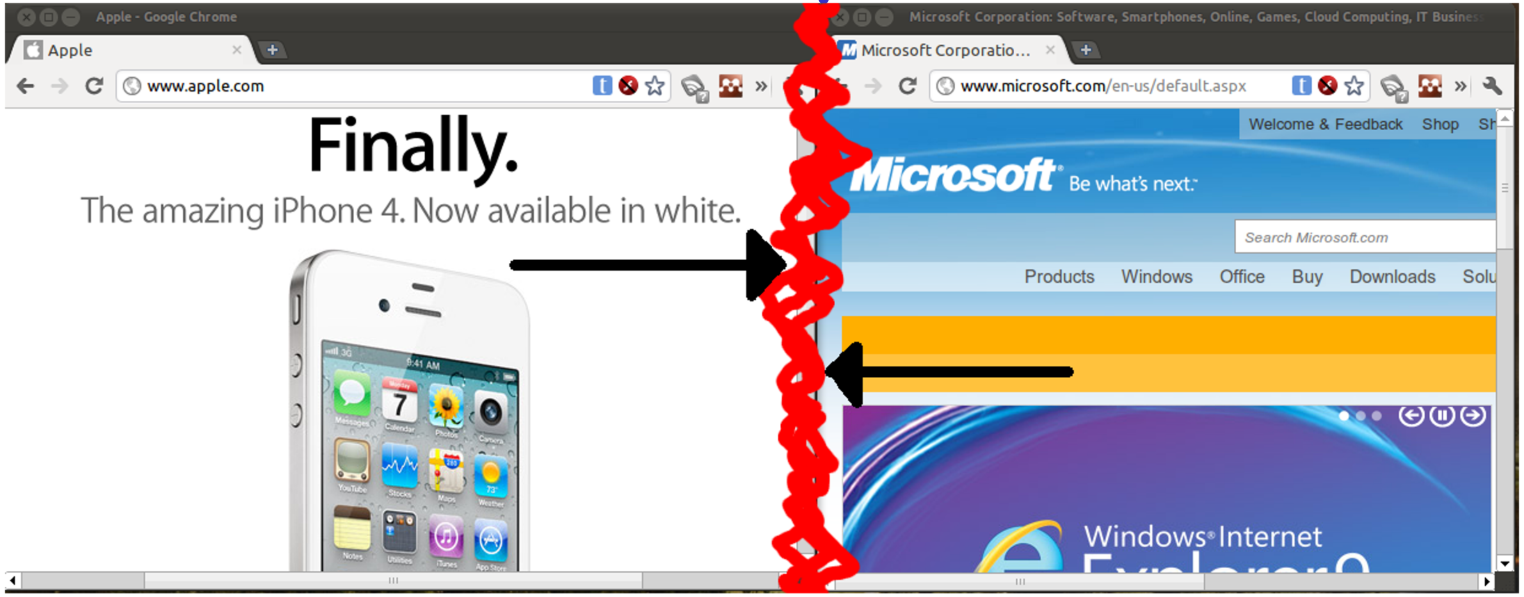


- HTTP is a stateless protocol.
  - User Authentication: Use cookies and send them implicitly for convenience.
- Server Authentication: SSL + Certification Authorities

Lifetime of Cached Cookies and HTTP Authentication Credentials

- Temporary cookies cached until browser shut down, persistent ones cached until expiry date
- HTTP authentication credentials cached in memory, shared by all browser windows of a single browser instance
- Caching depends only on browser instance lifetime, not on whether original window is open

Confidentiality (Browser)



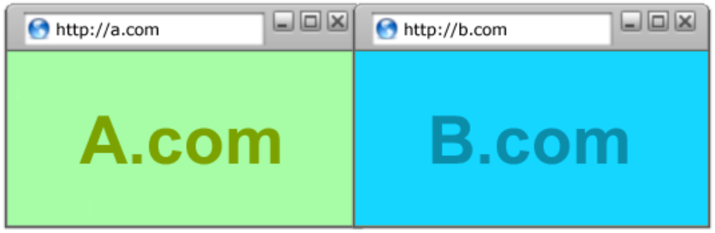
- No mutual trust among parties.
- Confidentiality through Isolation: Same-Origin Policy (SOP)
- Partition the Web into domains and isolate sensitive data such as cookie, network data and DOM nodes.

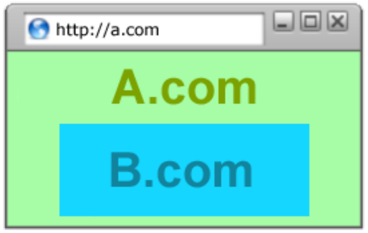
All of These Should Be Safe

- Safe to visit an evil website

http://a.com
A.com

- Safe to visit two pages at the same time

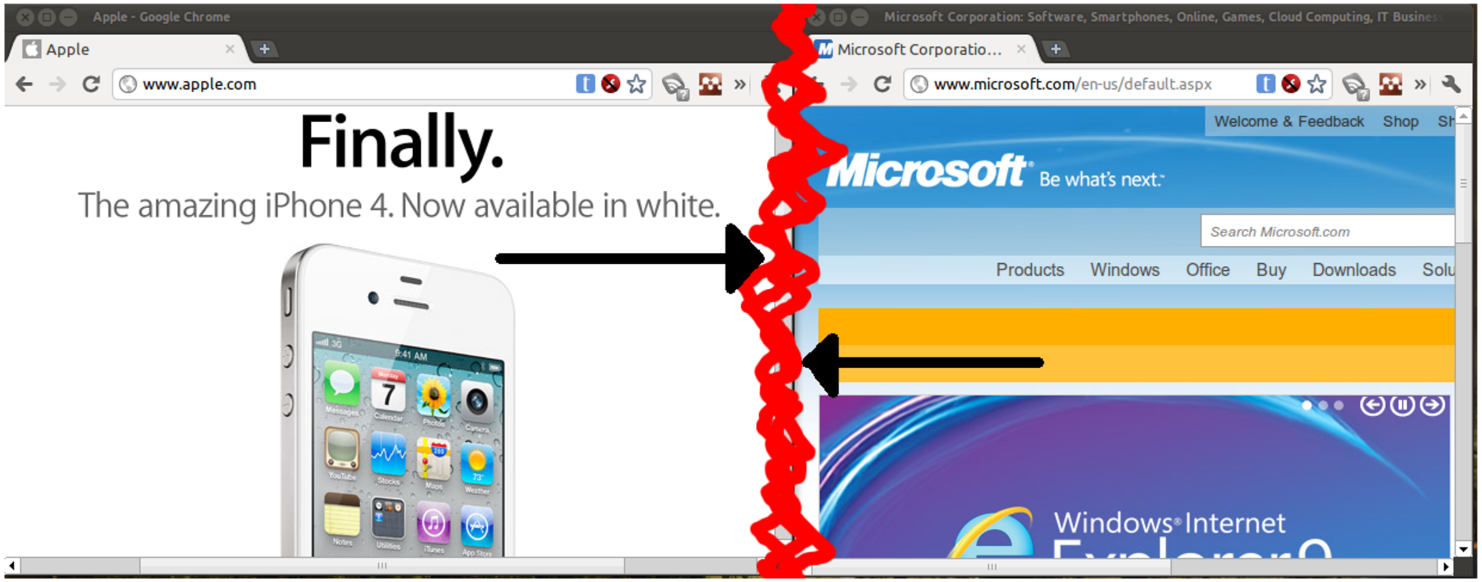




- Safe delegation

Same-Origin Policy (SOP)

- The SOP partitions the web into domains (according to their DNS origin) and isolates sensitive data from scripts running in other domains.
- What is sensitive data?
  - Cookies
  - Web page content (DOM isolation)
  - Web site response (Network isolation)


Apple - Google Chrome
Apple
www.apple.com
Finally.
The amazing iPhone 4. Now available in white.
Microsoft Corporation: Software, Smartphones, Online, Games, Cloud Computing, IT Business
Microsoft Corporatio...
www.microsoft.com/en-us/default.aspx
Welcome & Feedback
Shop
Microsoft Be what's next.
Search Microsoft.com
Products
Windows
Office
Buy
Downloads
Windows Internet Explorer 9

SOP: Cookie Isolation

- Each domain has its own set of independently managed cookies, and these are embedded only in requests to the same domain.
- Only scripts running from the same domain and responses from the same domain can read and write cookies
- HTTP-Only cookies

SOP: Page content isolation

- Basic unit of isolation in a browser is a <frame>
  - document.write – refers to the current frame
- DOM Isolation
  - Scripts only have access to DOM elements on the same domain.
  - Frames embedded in a page are part of the DOM tree of the parent, but the policy still applies:
  - document.frames[0].title
  - Only accessible if the parent is from the same origin.

SOP: Network isolation

- Script can send requests to arbitrary sites
- But scripts cannot read responses from any server
  - They can still send blind requests to other domains.
  - Is it safe for a malicious script to issue a request if it cannot read the response?
  - CSRF
- Exception: XmlHttpRequests permit a script to read from its origin server

Embedding and SOP: Caveats

- For embedded content, origin of the content may be different from the domain used for SOP checks
- Scripts retrieved from B and embedded in A run with A privileges.
  - Akin to user A running an executable written by B in a UNIX environment.
  - Plugins implement their own SOP-like policies.
    - Flash keeps its server origin.
  - Cross-site scripting attacks exploit this

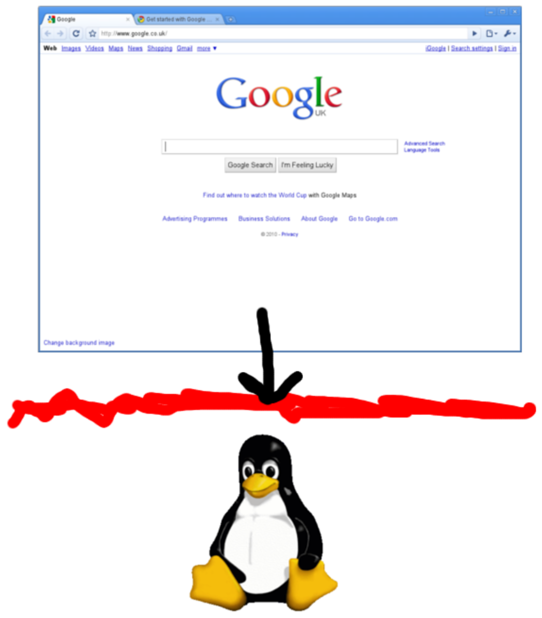
Same-Origin Policy: Exceptions

- Some resources are not considered sensitive and can be accessed across domains
- Browser History: CSS allows website to use different rules for visited and unvisited links.
- CSS rules: they can be read even when importing a cross-origin stylesheet
- Unsurprisingly, two attacks use these exceptions for information leaks
- Cross-origin CSS and CSS history hacks exploit these exceptions

Limitations of SOP

- A very rigid policy that imposes an all-or-nothing approach:
  - The developer can embed the resource (allow all) or open it in an iframe (allow none).
  - Cannot import script libraries without trusting them blindly.
- Does not limit outgoing requests

Confidentiality (OS)



- Users do not trust the websites they visit.
- Again: Confidentiality through Isolation
- Sandboxing: only expose a safe API to web application that limits their interaction with the browser
- DOM manipulation, cookie storage, drawing inside the browser window, etc.
- Recent developments: HTML5, WebGL, NaCl. Web developers need more capabilities for dynamic applications.

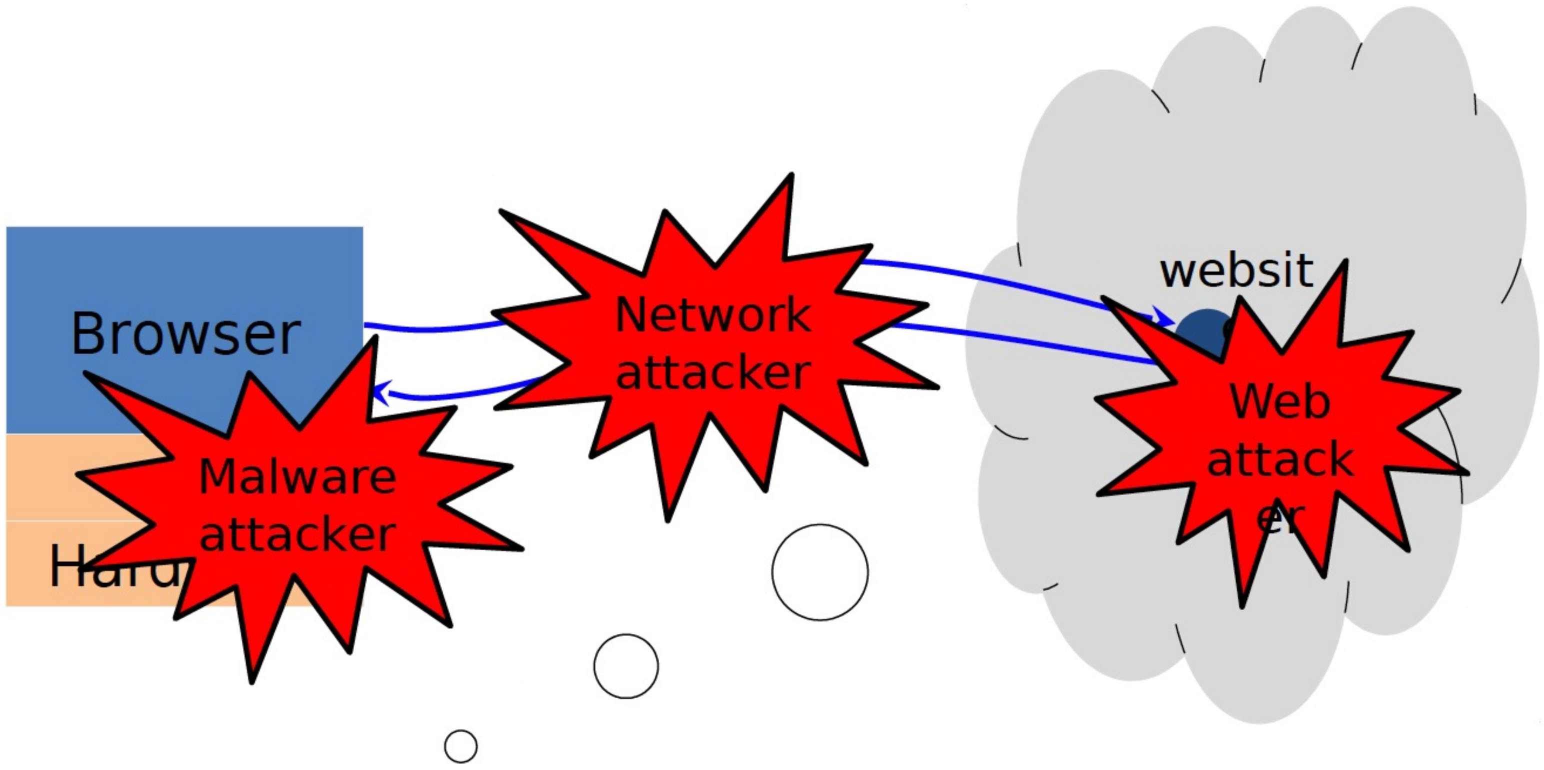
Integrity

- Network data integrity: HTTPS/DNSSEC
- Also used to authenticate the server (e.g Banks) and ensure network confidentiality.
- Public-key protocol used to establish a session key to encrypt traffic.
- Browser data integrity: SOP
- ``Integrity" as write access on confidential resources.

Despite the same origin policy

- Many things can go wrong at the client-side of a web application
- Popular attacks
  - Cross-site Scripting
  - Cross-site Request Forgery
  - Session Hijacking
  - Session Fixation
  - SSL Stripping
  - Clickjacking

Where Does the Attacker Live?



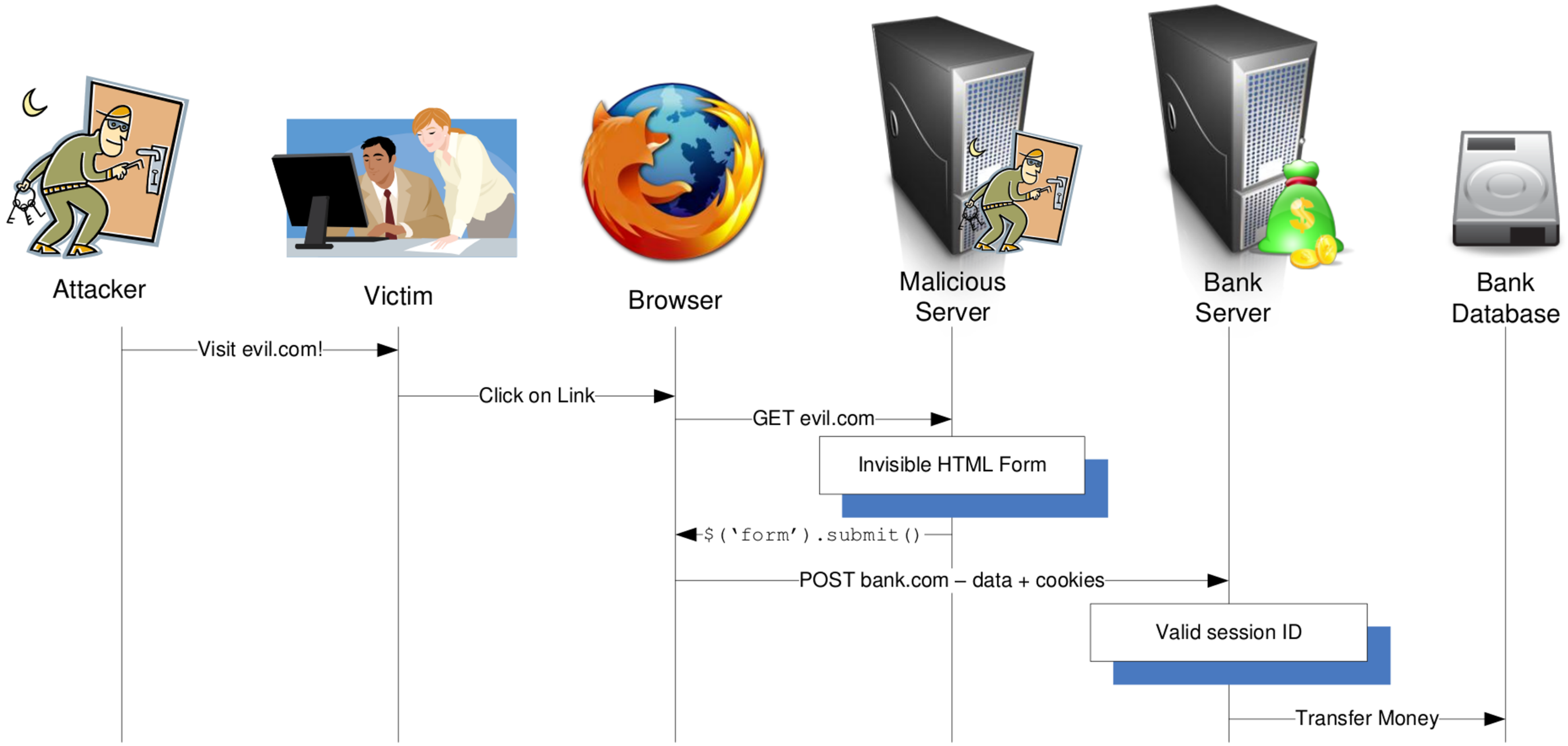
Threat Model 1: Web Attacker

- User, network and server are benign
- Attacker controls a malicious website (attacker.com)
  - Can even obtain an SSL/TLS certificate for his site ($0)
- Entices user to visit attacker.com
  - Phishing email, enticing content, search results, placed by an ad network, blind luck …
  - Attacker's Facebook app
- Attacker has no other access to user machine!
- Variation: "iframe attacker"
  - An iframe with malicious content included in an otherwise honest webpage
    - Syndicated advertising, mashups, etc.

Attacks on Authentication

- CSRF and Clickjacking
- Confused deputy attacks that cause the victim browser to send authenticated requests for the attacker's benefit
- CSRF: Cross-site request forgery: attacker sends requests to another web site, impersonating browser user
- Clickjacking: User intends to click on one link, but the browser recognizes a link on another site
- Achieved using overlaid frames and by manipulating visibility related attributes

CSRF


Attacker
Victim
Browser
Malicious Server
Bank Server
Bank Database
Visit evil.com!
Click on Link
GET evil.com
Invisible HTML Form
$('form').submit()
POST bank.com – data + cookies
Valid session ID
Transfer Money

Cross-site Request Forgery (CSRF)

- <form method="POST" action="/changepass">

...

New Password: <input type="password" name="password">

</form>

- Browser makes the following request :
- GET http://www.examplesite.com/changepass?password=new password HTTP 1.1
- Let's say the application didn't authenticate password change request using any means other than cookies
- An attacker can easily forge request!
  - Attack works because (a) cookies are sent by default, and (b) SOP does not restrict cross-origin submissions

POST Example

- POST requests can also be forged
- Attacker lures the client to visit his/her web page

```
  <iframe name=“hiddenframe” style=“display:none”>
    <form method=“POST” name=“evilform”.                    ₩
      target=“hiddenframe”
      action=http://www.examplesite.com/update_password>
      <input type=“hidden” name=“password”.
         value=“evilhax0r”>
   </form>
   <script>document.evilform.submit()</script>
  </iframe>
```

CSRF and Authentication status

- The classic CSRF attack abuses a user's existing session cookies with a victim website
- Does that mean that CSRF is a non-issue when a user is logged out?
  - No! (although many still think "yes")
  - In certain cases, an attacker can log in a victim with his credentials using an unprotected login form and still manage some sort of abuse
    - Login CSRF

Possible targets of CSRF

- Banks
  - Attacker can issue a request to transfer money from victim's bank account to attacker's
- E-commerce sites
  - Purchase items using victim's account, ship to attacker
- Forums and Social network sites
  - Post articles using victim's identity
- Home/Intranet firewall
  - Reconfigure firewall to permit connections from the Internet to a host behind the firewall
  - Note that victim user's location is exploited: the attacker (typically) cannot communicate with the firewall, but the user's browser can

Preventing CSRF

- HTTP requests originating from user action are indistinguishable from those initiated by attacker
- Need own methods to distinguish valid requests
  - Inspecting Referer Headers
  - Validation via User-Provided Secret
  - Validation via Action Token

Inspecting Referrer Headers

- Referer header specifies the URI of document originating the request

- Assuming requests from our site are good, don't serve requests not from our site
- Unfortunately, Referrer information may be suppressed by browsers (or firewalls) for privacy reasons

Validation via User-Provided Secret

- Can require user to enter secret (e.g. login password) along with requests that make server-side state changes or transactions
- Ex: The change password form could ask for the user's current password
- Security vs convenience: use only for infrequent, "high-value" transactions
  - Password or profile changes
  - Expensive commercial/financial operations

Validation via Action Token

- Add special action tokens as hidden fields to "genuine" forms to distinguish from forgeries
- Same-origin policy prevents 3rd party from inspecting the form to find the token

- Need to generate and validate tokens so that
  - Malicious 3rd party can't guess or forge token
- Browser's Same Origin Policy prevents attacker from "reading" the token
  - Then can use to distinguish genuine and forged forms

Clickjacking

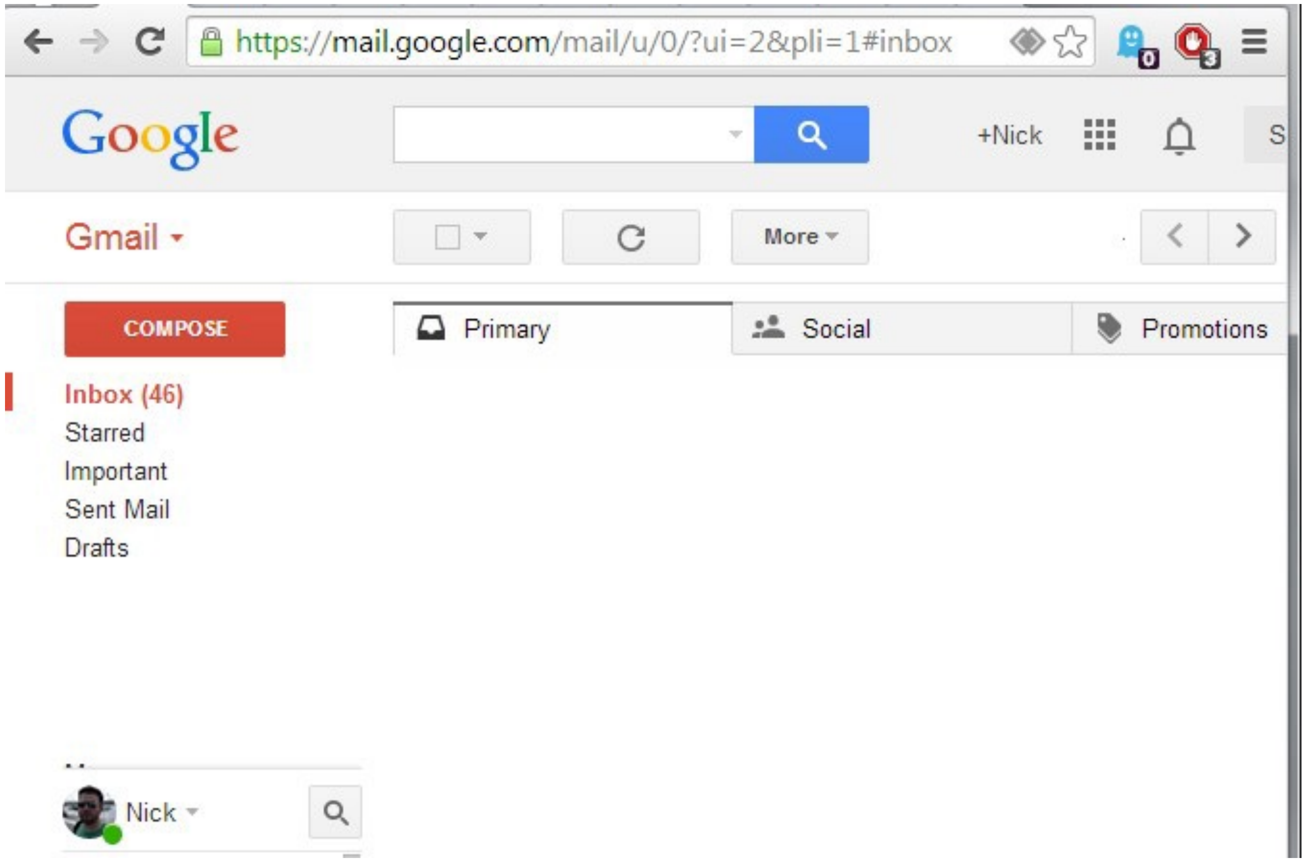
Wina free iphone! Justclickonredandgreen!

Quickwhile the offer lasts!

So you click…

- Nothing happens.
  - Or something happens
  - But you don't get that free iphone that you were promised
- • Continue browsing
- • Time to check email
  - Go to GMail

Where are my mails bro?!?



Clickjacking Defenses

- Disallow hidden frames
  - There are many ways to make a frame imperceptible
- Restrict framing
  - X-Frame-Options header
    - SAMEORIGIN;
  - Allow-from <uri>;
  - DENY;
- Content security policy (supersedes X-frame)
  - Content-Security-Policy: frame-ancestors 'self'
  - Content-Security-Policy: frame-ancestors a.com b.org
  - Content-Security-Policy: frame-ancestors 'none'

Cross-Site Scripting (XSS)

- Different types of script injection
  - Reflected: part of the URI used in the response
  - Persistent: stored data used in the response
  - DOM-based: data used by client-side scripts

What can an attacker do with XSS?

- Long answer (non exhaustive):
  - Exfiltrate your cookies (session hijacking)
  - Make arbitrary changes to the page (phishing)
  - Steal all the data available in the web application
  - Make requests in your name
  - Redirect your browser to a malicious page
  - Tunnel requests to other sites, originating from your IP address (BEEF)

Reflected XSS Example

- Host www.vulnerable.site displays name submitted using a web form
  - With benign data, following request may result

GET /welcome.cgi?name=Joe%20Hacker HTTP/1.0

- And the web site responds:

  <H TML> <Title>Welcome!</Title> Hi Joe Hacker<BR>

  Welcome to our System</HTML>

- What if the attacker submits

  GET welcome.cgi?name=<script>...<script> HTTP/1.0

Reflected XSS Summary

- Attacker causes victim to click on maliciously crafted link
  - Typically contains a malicious script as a parameter
- request goes to vulnerable web site
- web site does not properly check its input
- returns a page that contains the malicious script
  - which operates with privileges of the vulnerable site
    - can perform any action that the user can perform
    - send the cookie (or other private info) to the attacker

    -- perform sensitive action, e.g., withdraw money

Persistent XSS

- Malicious script permanently stored on server
- Still requires
  - An attack that causes the script to be stored
  - Script should be used in a page visited by victim user
- User totally unaware of the vulnerability/exploit
  - More stealthy, damaging and long-lasting
  - How can this be possible?
    - Think of a blog, or social networking web site: input from one user is rendered in the page shown to another

DOM-Based XSS

- DOM-Based refers to how the script comes about
  - Plain XSS: malicious script is already present in the page from server
  - DOM-based XSS: ●

    server delivers an initial page content and a legitimate script
  - execution of this script constructs the rest of the page using DOM operations
    - document.write
    - document.createElement
    - document.appendChild ...
  - malicious script content manifests during this construction
- Orthogonal to reflected vs persistent categorization
  - DOM-based XSS can be of either kind

Preventing XSS

- Server should not send untrusted data to the browser that could result in the creation of an unintended (and unauthorized) script
  - Usually can just suppress certain characters, but this is not enough in the case of DOM-based
- We show examples of various contexts in HTML document as *templatesnippets*
  - Variable substitution placeholders: %(var)
  - evil-script; will denote what attacker injects
  - Contexts where XSS attack is possible

Simple Text

- Most straightforward, common situation Example Context:

  `<b>Error: Your query '%(query)' did not return any results.</b>`

  - Attacker sets query = <script>evil-script;</script>
  - HTML snippet renders as

  `<b>Error: Your query '<script>evil-script;</script>' did not return any results.</b>`

- Prevention: HTML-escape untrusted data
  - Rationale: If not escaped <script> tags evaluated, data may not display as intended

Tag Attributes (e.g., Form field values)

- Contexts where data is inserted into tag attribute Attacker able to

  “close the quote”, insert script Example HTML
Fragment:

  <form ...><input name="city" value="%(city)"></form>

- Attacker sets

  city = xyz"><script>evil-script;</script>

  <form ...>

- Renders as

  <input name="city" value="xyz"> <script>evil-script;</script>">
</form>

More Attribute Injection Attacks

- Image Tag: <img src=%(image_url)>
- Attacker sets image_url = http://www.examplesite.org/ onerror=evil-script;
- After Substitution: <img src=http://www.examplesite.org/ onerror=evil-script;>
- Lenient browser: first whitespace ends src attribute onerror attribute sets handler to be desired script

- Attacker forces error by supplying URL w/o an image
- Can similarly use onload, onmouseover to run scripts
- Attack string didn't use any HTML metacharacters!

URL Attributes (href and src)

- Dynamic URL attributes vulnerable to injection
  - Script/Style Sheet URLs: <img src="% (script_url)">
  - Attacker sets script_url = http://hackerhome.org/evil.js

  - javascript: URLS -<img src="%(img_url)">
  - By setting img_url = javascript:evil-script;
  - we get <img src="javascript:evil-script;">
- And browser executes script when loading image

Style Attributes

- Dangerous if attacker controls style attributes
  - Attacker injects:
  - Browser evaluates:

  <div style="background: %(color)">I like colors.</div>

  color = green; background-image: url(javascript:evil-script;)

  <div style="background: green; background-image: url(javascript:evil-script;)"> I like colors. </div>
- In IE 6 (but not Firefox 1.5), script is executed!
  - Prevention: whitelist through regular expressions
    - Ex: ^([a-z]+)|(#[0-9a-f]+)$ specifies safe superset of possible color names or hex designation

In JavaScript Context

- Be careful embedding dynamic content <script> tags or handlers (onclick, onload, ...)

```
<script>
   var msg_text = '%(msg_text)'; // do something with msg_text
 </script> msg_text = oops'; evil-script; //
```

- Attacker injects:

```
<script>
var msg_text = 'oops'; evil-script; //'; // do something with msg_text
</script>
```

- And evil-script; is executed!

Another JavaScript Injection Example

- From previous example, if attacker sets

```
msg_text = foo</script><script>evil-script;</script><script>
```

- the following HTML is evaluated:

```
<script>var msg_text = 'foo</script>
<script>evil-script;</script>
<script>'// do something with msg_text</script>
```

- Browser parses document as HTML first divides into 3 <script> tokens before interpreting as JavaScript
  - Thus 1st & 3rd invalid, 2nd executes as evil-script

Defending against XSS

- Blacklisting
  - E.g. No <, >, script, document.cookie, etc.
  - Intuitively correct, but it should NOT be relied upon
    - As we saw in the last few slides, there are too many ways to insert script content. The XSS Cheat Sheet lists hundreds of possibilities
- Whitelist whenever possible
  - E.g. this field should be a number, nothing more nothing less
- Always escape user-input
  - Neutralize "control" characters for all contexts
- Content Security Policy
  - Whitelist for resources
  - Belongs in the "if-all-else-fails" category of defense mechanisms

A web site vulnerable to XSS

- Host: www.vulnerable.site
- GET /welcome.cgi?name=value HTTP/1.0
- Displays name submitted in the web page
- Example
- GET /welcome.cgi?name=Joe%20Hacker HTTP/1.0

Web site response

```
<HTML>
<Title>Welcome!</Title>
Hi Joe Hacker
<BR>
Welcome to our system
...
</HTML>
```

How can this be abused??

Summary

- Attacker causes victim to click on maliciously crafted link
- request goes to vulnerable web site
- web site does not perform input filtering
- returns a page that contains executable code that sends private information to attacker

Attack details

- Above attack requires victim to click on attacker link
  - Easy way: use email messages with enticing information
  - victim clicks on link
  - Variation: Attacker provides scripting code as input to vulnerable web application

How to run passive attacks?

- These are attacks where user will not perform explicit actions
- How can this be possible?
- Think of a blog, where user input becomes part of the page's comments
- Stealthy, and mostly unknown to user browsing the page

XSS

- Unauthorized scripts come from user input
- Can we identify scripts that are legitimate vs. those that are injected?
- If so, the web site can reject any script content that did not come from it
- This requires “tracking” user input as it flows through the application

References

- XSS (Cross Site Scripting) Cheat Sheet Esp: for filter evasion http://ha.ckers.org/xss.html
- Technical Explanation of The MySpace Worm http://namb.la/popular/tech.html
- Malicious Yahooligans www.symantec.com/avcenter/reference/malicious.yahooligans.pdf

Content Security Policy

- Example

```
Content-Security-Policy: default-src
https://cdn.example.net; frame-src 'none'; object-src
'none'; image-src self;
```

- CSP is very powerful
  - Great if you are writing something from scratch
  - Not so great if you have to rewrite something to CSP
    - E.g. Convert all inline JavaScript code to files

Content Security Policy v2

- CSP was great in theory but still hasn't caught up in practice

- CSP v2.0 supports two new features to help adopt CSP
  - Script nonces for inline scripts
  - Hashes for inline scripts
  - Read more here:
    - https://blog.mozilla.org/security/2014/10/04/csp-for-the-webwe-have/

Content Security Policy v2

- Script nonces for inline scripts

  [HTTP Header] Content-security-policy: default-src 'self'; script-src 'nonce-2726c7f26c'

  [HTML] <script nonce="2726c7f26c">… </script>

- Hashes for inline scripts
  - [HTTP Header] content-security-policy: script-src 'sha256-
  - cLuU6nVzrYJlo7rUa6TMmz3nylPFrPQrEUpOHllb5ic='
  - [HTML] <script> … </script>

Browser XSS filters

- Some browsers try to help by attempting to detect reflected XSS and stop them
  - Internet Explorer was the first to introduce this
  - Chrome followed a bit later, with a more complete approach that addressed some of IEs problems
    - Unfortunately, over the years, Chrome's filter seems to have gone back on some of the improvements. Its filter stops fewer attacks than IE in our experiments
  - Firefox invested in an XSS filter for some time, but then seems to have abandoned its efforts
    - PaleMoon, a Firefox clone, imported the XSS filter for Firefox developed at Stony Brook.

Browser XSS filters

- Attempt 1: Use string (or regexp) matching to identify suspicious content within request parameters (NoScript)

Example: excise "<script>", "data:", etc. from parameters Problem: High False Positives make it unsuitable for general use

- Attempt 2: Filter only if suspicious parameter is reflected, i.e., its value appears in the HTML response (IE/Edge) FPs can still be too high Mitigate using very strict matching rules (IE/Edge, Chrome) Unfortunately, this leads to false negatives and filter evasion
- Attempt 3: Filter if suspicious reflected content is used in a dangerous context in the HTML response (Firefox filter)

Example: "data:" can safely appear outside HTML tags In our filter, this reduces FPs sufficiently to enable use of approximate matching Result: Evasion resistant XSS filtering

Script Inclusion

- What if an attacker can't find an XSS vulnerability in a website
  - Can he somehow still get to run malicious JavaScript code?
- Perhaps… by abusing existing trust relationships between the target site and other sites

JavaScript libraries

- Today, a lot of functionality exists, and all developers need to do is link it in their web application
  - Social widgets
  - Analytics
  - JavaScript programming libraries
  - Advertising
  - ...

Remote JavaScript libraries

- mybank.com

```
<html> … <script src=http://www.foo.com/a.js> </script> … </html>
```

- The code coming from foo.com will be incorporated in mybank.com, as if the code was developed and present on the servers of mybank.com

Remote JavaScript libraries

- This means that if, foo.com, decides to send you malicious JavaScript, the code can do anything in the mybank.com domain
- Why would foo.com send malicious code?
  - Why not?
  - Change of control of the domain
  - Compromised

Timing attacks

- Because of the same-origin policy, scripts cannot access most resources in a cross-domain
  - Can still make the requests though, that's why CSRF is a problem
  - An attacker can still abuse the time it takes for a page to load, as a side-channe

Timing attacks

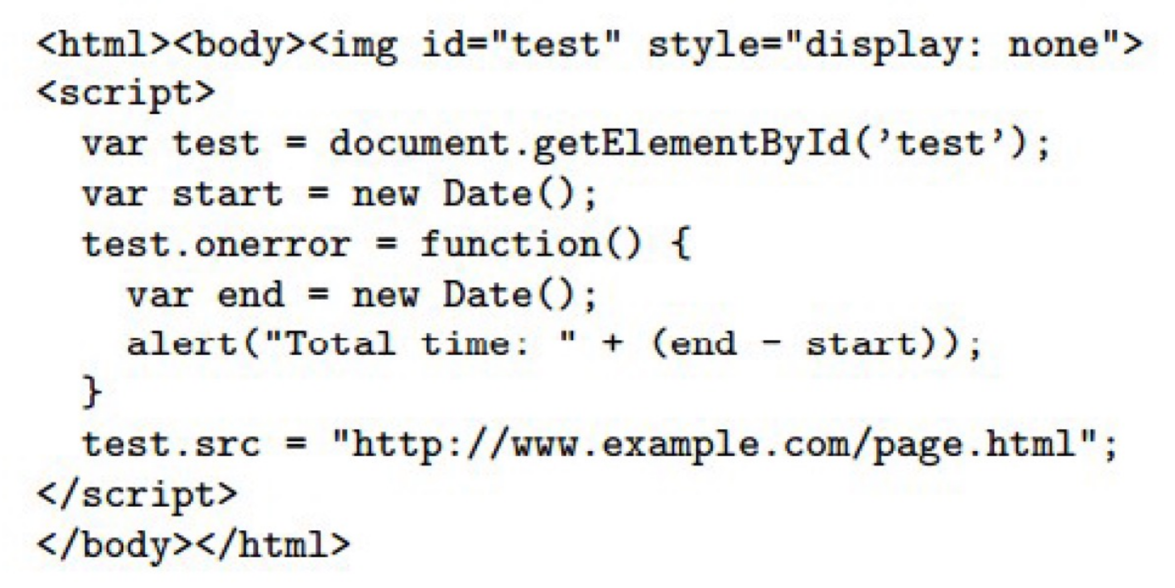
- Scenario: I want to know if you are logged into your Gmail
  - I may, or may not be able to load the page in an iframe, depending on the Xframe-options
  - Even if I can load it, I still can't peek in it
- What if I try to load mail.google.com as an image?
  - <img src=https://mail.google.com onError="func()"/>
  - The browser will fetch the page with your cookies and then the parser will at some point throw an error that this is not an image

Timing attacks

- The size of a page is often dependent on whether you are logged in or not
- Hence, for a large image, the browser will take a longer time to give you an error

- Oversimplified attack:
  - Fast error: not-logged in
  - Slow error: logged-in

Getting one measurement



```
<html><body><img id="test" style="display: none">
<script>
  var test = document.getElementById('test');
  var start = new Date();
  test.onerror = function() {
    var end = new Date();
    alert("Total time: " + (end - start));
  }
  test.src = "http://www.example.com/page.html";
</script>
</body></html>
```

Figure 3: Example JavaScript timing code

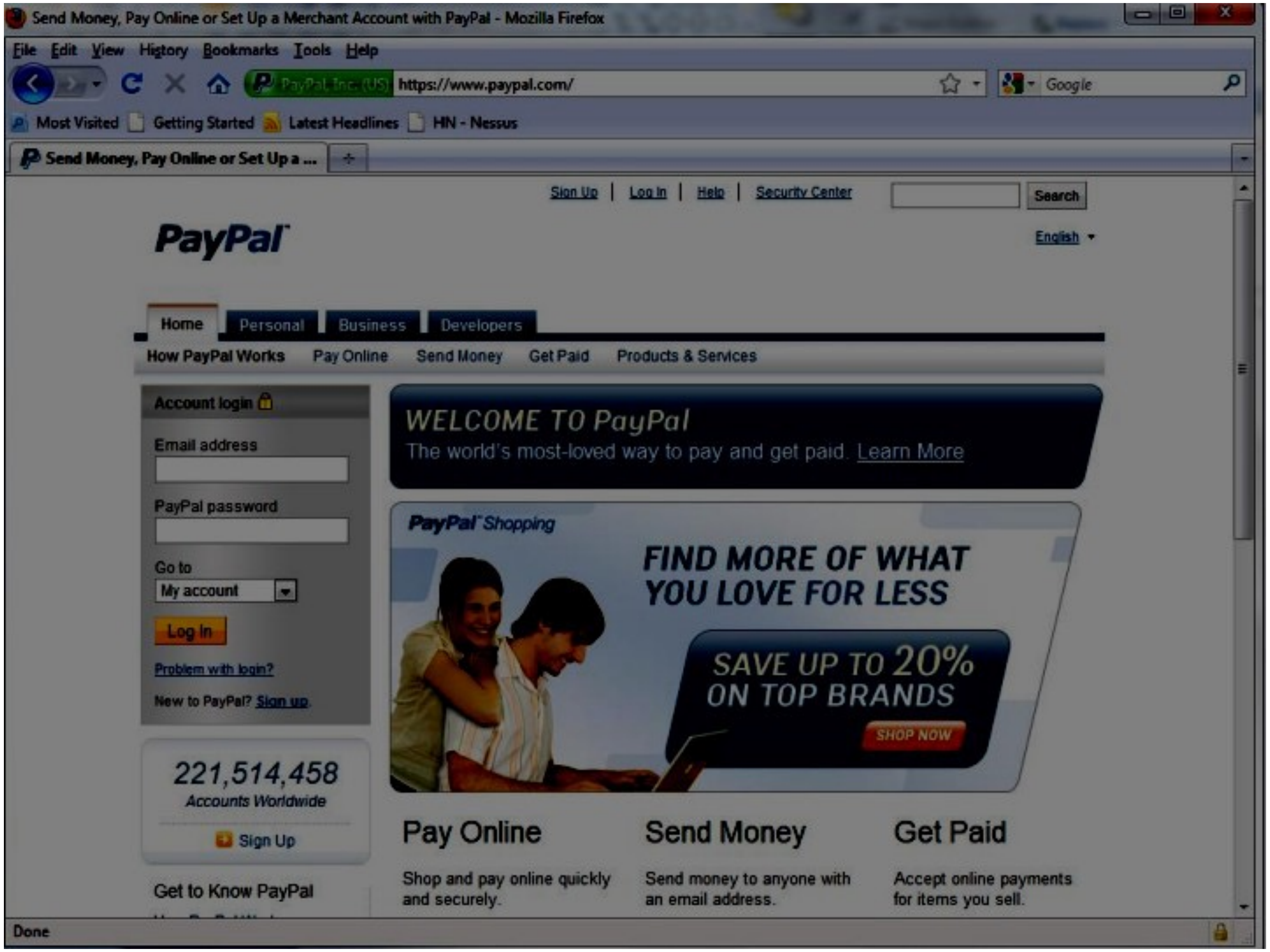
Code sample from: *ExposingPrivateInformationbyTimingWebApplications* By Bortz et al.

Threat Model 2: Network Attacker

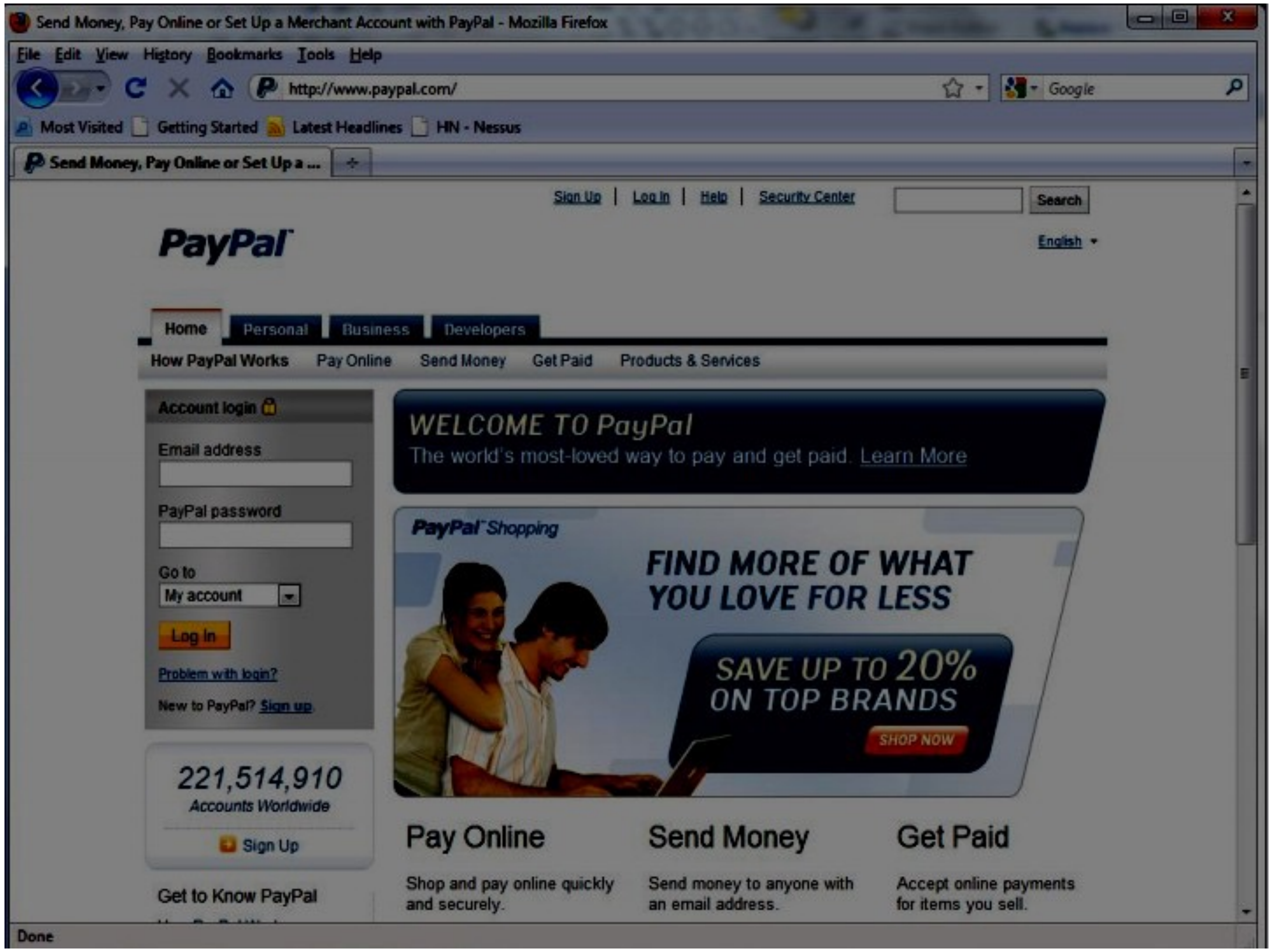
SSL Stripping

- Let's say that a website exists only over HTTPS
  - No HTTP pages
- Two scenarios
  1. User types https://www.securesite.com and the browser directly tries to communicate the remote server over a secure channel
  2. User types http://www.securesite.com (or just securesite.com) and the site will redirect the user to the secure version (using an HTTP redirection/Meta header)

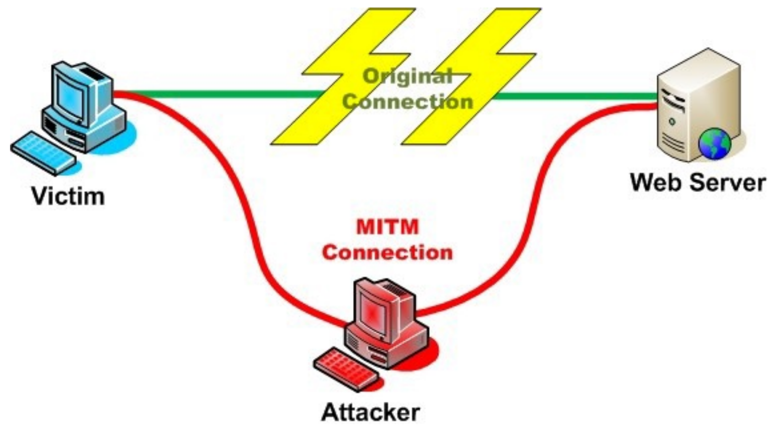
Normal page load


Send Money, Pay Online or Set Up a Merchant Account with PayPal - Mozilla Firefox
File Edit View History Bookmarks Tools Help
PayPal, Inc. (US) https://www.paypal.com/
Google
Most Visited Getting Started Latest Headlines HN - Nessus
Send Money, Pay Online or Set Up a ...
Sign Up | Log In | Help | Security Center
Search
PayPal
English
Home Personal Business Developers
How PayPal Works Pay Online Send Money Get Paid Products & Services
Account login
Email address
PayPal password
Go to
My account
Log In
Problem with login?
New to PayPal? Sign up.
WELCOME TO PayPal
The world's most-loved way to pay and get paid. Learn More
PayPal Shopping
FIND MORE OF WHAT YOU LOVE FOR LESS
SAVE UP TO 20% ON TOP BRANDS
SHOP NOW
221,514,458
Accounts Worldwide
Sign Up
Get to Know PayPal
Pay Online
Shop and pay online quickly and securely.
Send Money
Send money to anyone with an email address.
Get Paid
Accept online payments for items you sell.
Done

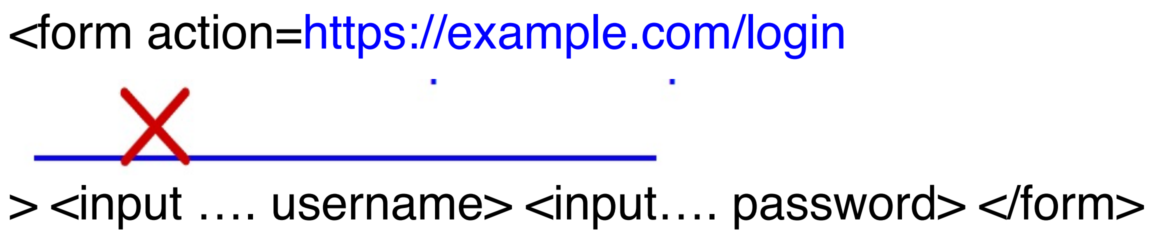
Page load when attacker is present


Send Money, Pay Online or Set Up a Merchant Account with PayPal - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://www.paypal.com/
Google
Most Visited
Getting Started
Latest Headlines
HN - Nessus
Send Money, Pay Online or Set Up a ...
Sign Up | Log In | Help | Security Center
Search
PayPal
English
Home
Personal
Business
Developers
How PayPal Works
Pay Online
Send Money
Get Paid
Products & Services
Account login
Email address
PayPal password
Go to
My account
Log In
Problem with login?
New to PayPal? Sign up.
WELCOME TO PayPal
The world's most-loved way to pay and get paid. Learn More
PayPal Shopping
FIND MORE OF WHAT YOU LOVE FOR LESS
SAVE UP TO 20% ON TOP BRANDS
SHOP NOW
221,514,910
Accounts Worldwide
Sign Up
Get to Know PayPal
Pay Online
Shop and pay online quickly and securely.
Send Money
Send money to anyone with an email address.
Get Paid
Accept online payments for items you sell.
Done

SSL Stripping



- Same thing can happen when sites deliver HTTPS-targeted forms over an HTTP connection (typically for performance or outsourcing purposes)



```
<form action=https://example.com/login
> <input …. username> <input…. password> </form>
```

Defenses

- Use full-site SSL in combination with Secure cookie and HTTP-only Cookie
- HSTS: HTTP Strict Transport Security
  - Force the browser to always contact the server over an encrypted channel, regardless of what the user asks

HTTP Header Strict-Transport-Security: max-age=31536000

Defenses

- What about the very first time you visit a website?
  - What if a MITM is located on your network and will therefore strip SSL and suppress HSTS?
- Answer:
  - Preloaded HSTS: Websites can ask browsers to mark them as HSTS in a special browservendorupdated database

Threat model 3: Malicious Client

- In these scenarios:
  - The server is benign
  - The client is malicious
    - The client can send arbitrary requests to the server, not bound by the HTML interfaces
- The attacker is after information at the server-side
  - Steal databases
  - Gain access to server
  - Manipulate server-side programs for gain

OWASP Top 10

A1 – Injection
A2 – Broken Auth and Session Management
A3 – Cross-site Scripting
A4 – Insecure Direct Object References
A5 – Security misconfiguration
A6 – Sensitive Data Exposure
A7 – Missing function level access control
A8 – Cross-site Request Forgery
A9 – Using components with kn. vulnerabilities
A10 – Unvalidated redirects and Forwards

Injection Attacks

- SQL injection
  - Steal sensitive data about specific user
  - All username, password (hashes) info
  - ...
- Command injection
  - Install malware on server, run reconnaissance commands, probe serverside network, inject into command streams for backend servers, ...
- We discussed these attacks and their defenses before
  - Defenses need to be mindful of trust boundaries, e.g., don't rely on client-side sanitization if the attacker is the client!

Redirects, Cookies, and Header Injection

- Need to filter and validate user input inserted into HTTP response headers
- Ex: servlet returns HTTP redirect

```
        HTTP/1.1 302 Moved Content-Type: text/html; charset=ISO-8859-1
         Location: %(redir_url)s
        <html> <head><title>Moved</title></head> <body>Moved <a
href='%(redir_url)s'>here</a></body> </html>
```

- Attacker Injects:

```
        oops:foo\r\nSet-Cookie: SESSION=13af..3b; (URI-encodes
        domain=mywwwservice.com\r\n\r\n
        <script>evil()</script>
```

newlines)

Logic Vulnerabilities

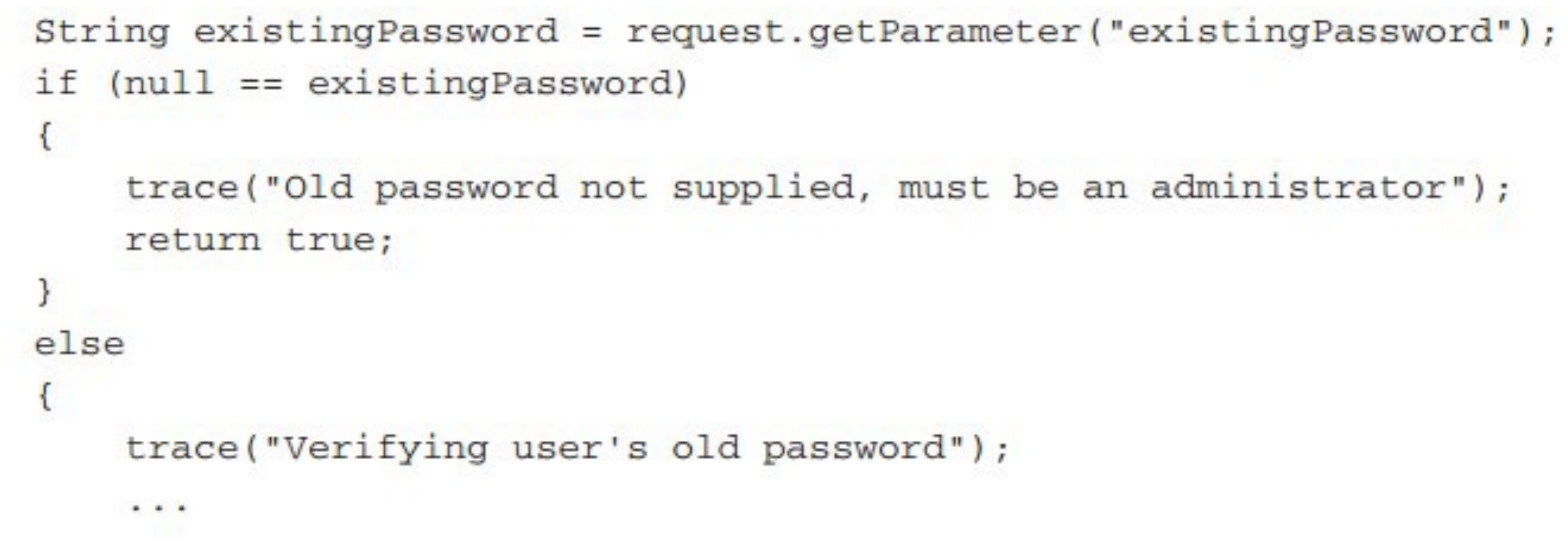
- HTTP parameter tampering vulnerabilities are a subset of logic vulnerabilities in web applications
- Logic vulnerabilities typically rely on breaking assumptions made by architects and developers
- Assumptions
  - Step 2 can only be performed after Step 1
  - The web application controls the navigation steps
  - Users cannot change parameters that they cannot see
  - Etc.

Examples of logic vulnerabilities

- Unlike vulnerabilities discussed so far, logic vulnerabilities don't have a clear, narrow definition
- This makes them hard to identify, especially b automated vulnerability discovery tools
- We will see a few real-world examples based on the book "The Web Application Hackers Handbook"

Case Study: Password change

- A website allows its users to change their password, by filling out a form with their current password, and their new password
    - Administrators can also change a user's password but they don't need to provide a user's current password



```
String existingPassword = request.getParameter("existingPassword");
if (null == existingPassword)
{
    trace("Old password not supplied, must be an administrator");
    return true;
}
else
{
    trace("Verifying user's old password");
    ...
```

Case Study: Password change

- The code that handles these two cases is the same and the developer assumes that if the "existingPassword" parameter is not present, this must be because the current request came from an administrative UI

- All the attackers has to do is drop the "existingPassword" HTTP parameter from the outgoing request

Case Study: Bulk Discounts

- An online shop gives users discounts when they buy some products together
  - E.g. If you purchase an antivirus solution, and a personal firewall, and antispam software then you are entitled to 25% discount on each product
- Abuse
  - Add all products in your basket to get the discount and then remove the ones you don't want

Case Study: Escaping from escaping

- A web application has to pass user-controllable input as an argument to an operating system command.
- The developer creates a list of special shell metacharacters that need escaping
  - ; | & < > ' space and newline
- If any of these are present in the input, the code escapes them by prepending them with a backslash
  - – \

Case Study: Escaping from escaping

- If an attacker types
  - foo;ls
- The code converts it to
  - foo\;ls
- What if an attacker types an escape character
  - foo\;ls
- Will become
  - foo\\;ls
- Which amounts to escaping the backslash but not the semicolon

Weaknesses Leading to Attacks

- Trusting embedded content
  - Embedded scripts have same privilege as surrounding page (XSS)
  - Embedded content can target browser flaws, e.g., buffer overflows
- Not restricting outgoing network requests
  - Unauthorized requests to third-party sites (CSRF)
  - Include trusted party content in a frame
    - Abuse trust in third party, e.g., to improve odds of successful phishing
    - Clickjacking
  - Attacking third-party sites, e.g., portscanning or launching exploits
  - Ease of leaking sensitive data acquired (e.g., send cookie to attacker)
- Allowing Turing-complete computation for arbitrary sites
  - Bitcoin mining
  - Side-channel attacks
  - JIT-ROP attacks

Weaknesses Leading to Attacks

- Weaknesses in lower layers
  - In-network attacks, e.g., man-in-the-middle
  - DNS compromise
- Application development environments that blur trust boundaries
    - Trusting client-side: browser and/or scripts running on a web page (Parameter tampering, …)
- Good old application logic or implementation vulnerabilities
  - SQL injection, command injection, HTTP parameter pollution, …

Credits

- Many of the slides here are the courtesy of Nick Nikiforakis and Venkat Venkatakrishnan

Questions?