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

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"
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
```

  - 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
- Numerous complex applications
  - Office 360, Google Maps, …
- Most web pages today are mainly Javascript
  - "Content" is fetched and displayed under the control of Javascript (AJAX)

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

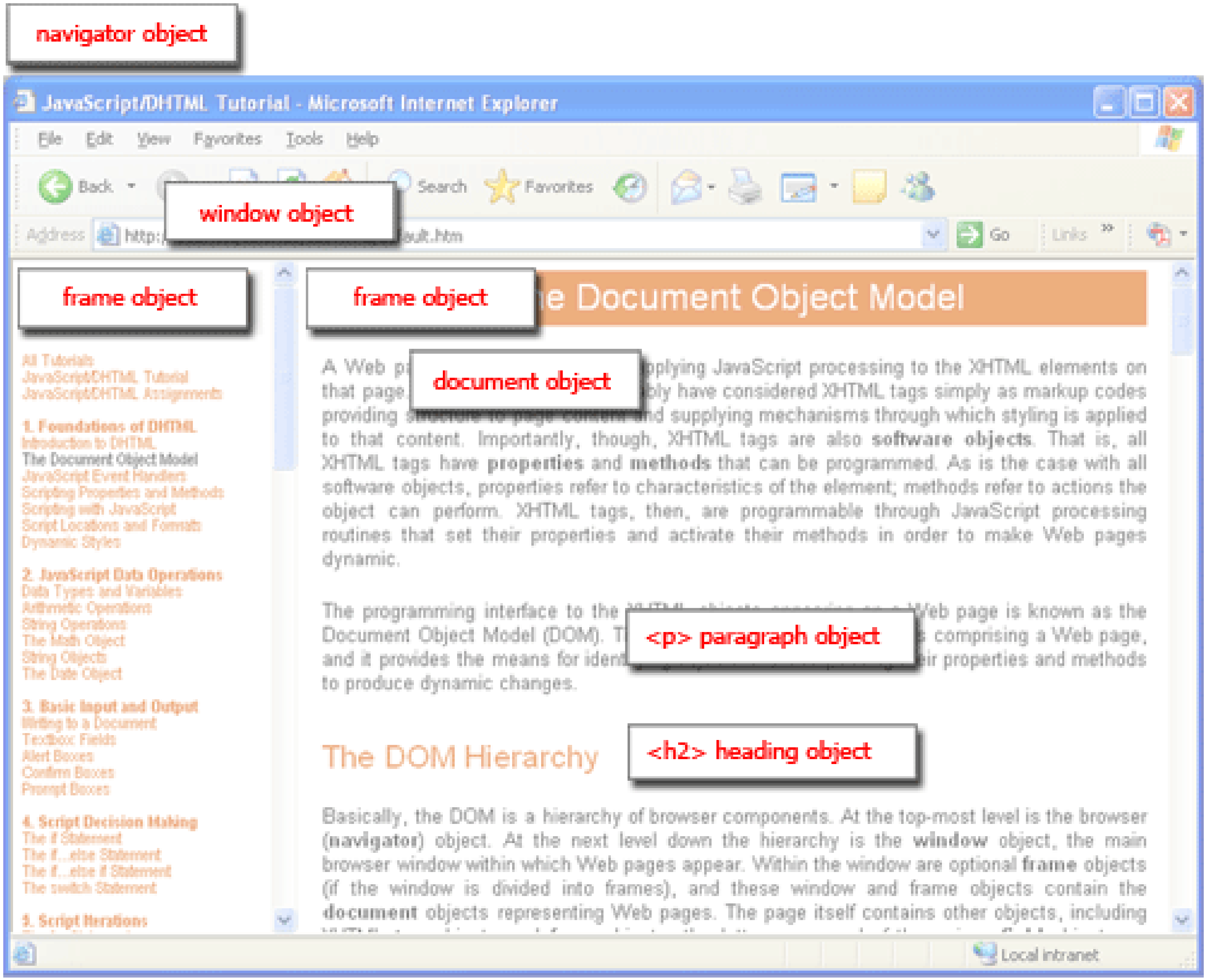
    <a href="http://www.yahoo.com" onmouseover="alert('hi');">
- Pseudo-URL referenced by a link

    Click me

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



Reading Properties with JavaScript

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

Sample HTML

<ul id="t1">
 Item 1

- 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

- Some possibilities
  - createElement(elementName)
  - createTextNode(text)
  - appendChild(newChild)
  - removeChild(node)
- Example: add a new list item

Sample HTML

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

```
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?

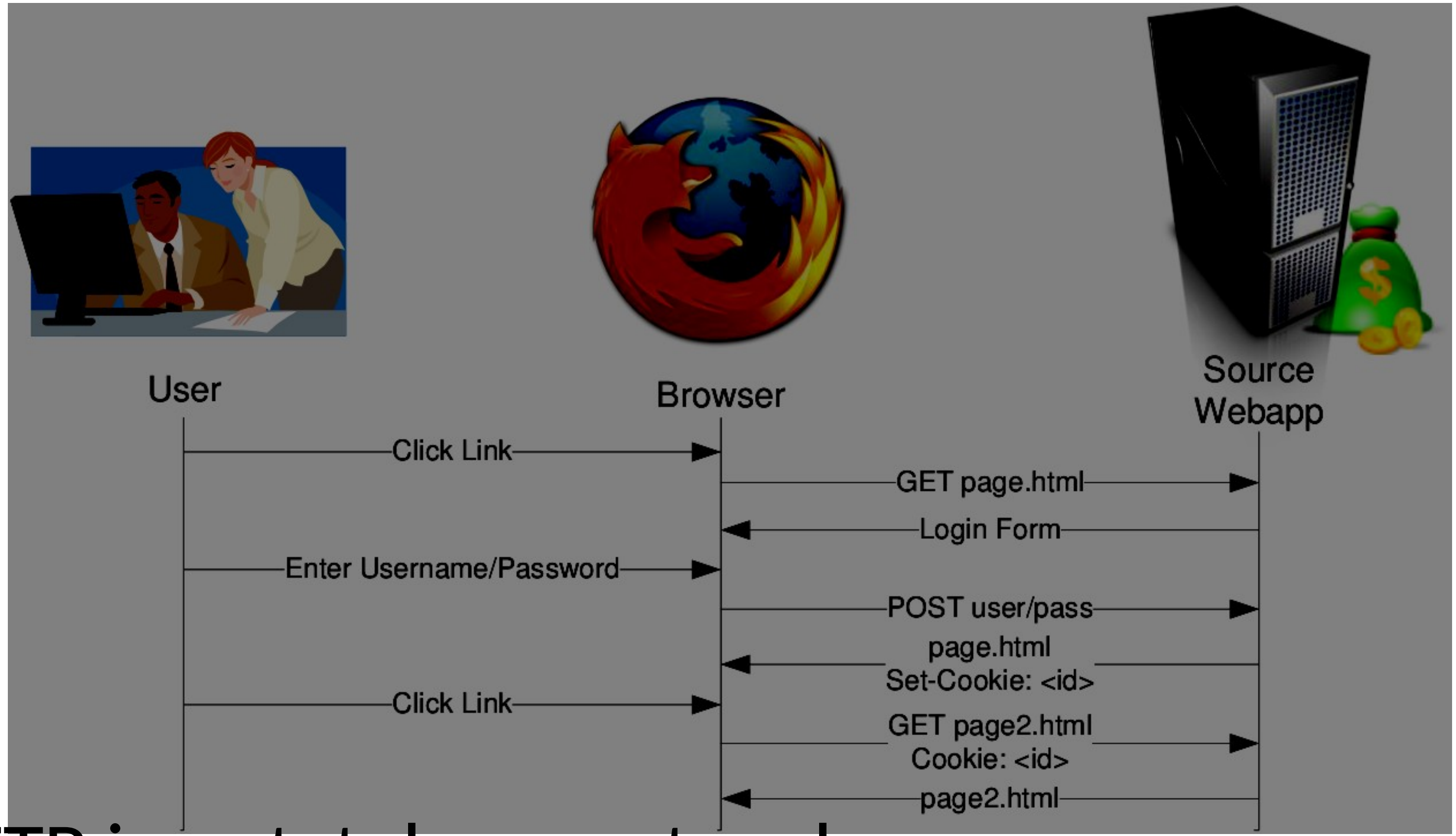
Web Security Concerns

- 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

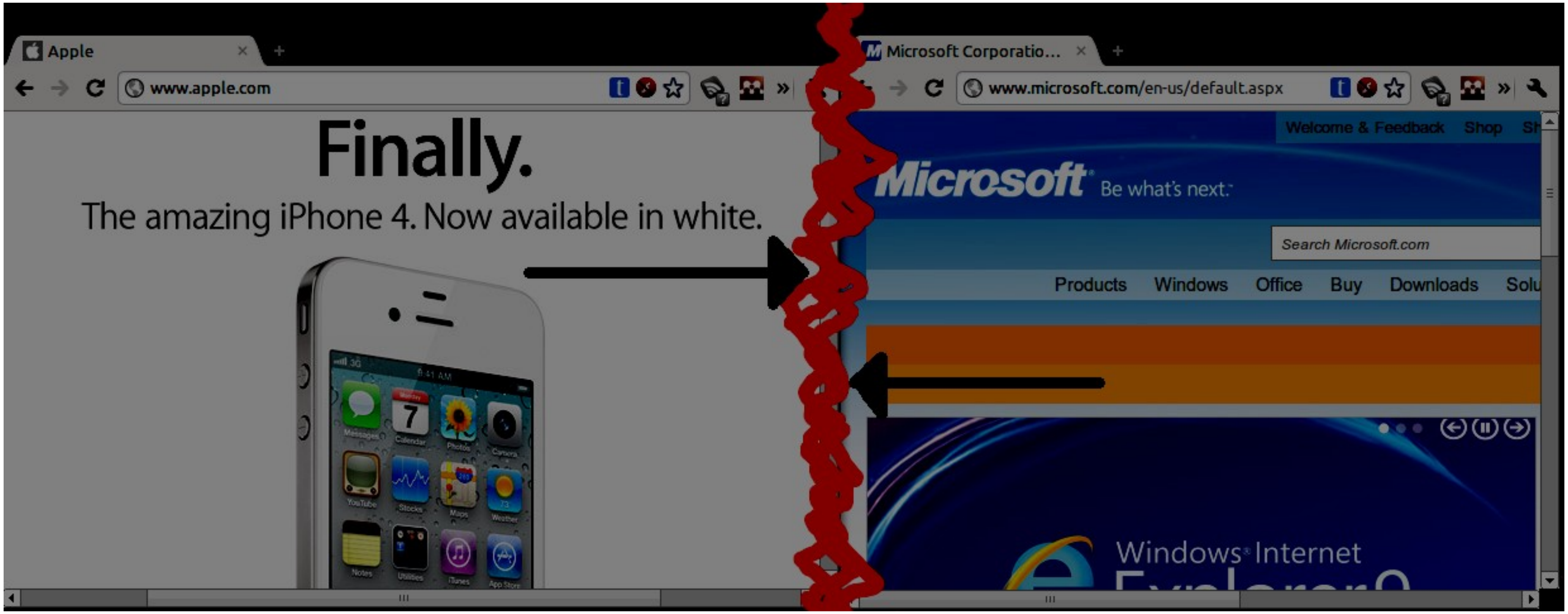
Lifetime of Cached Cookies and HTTP Authentication Credentials

- 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

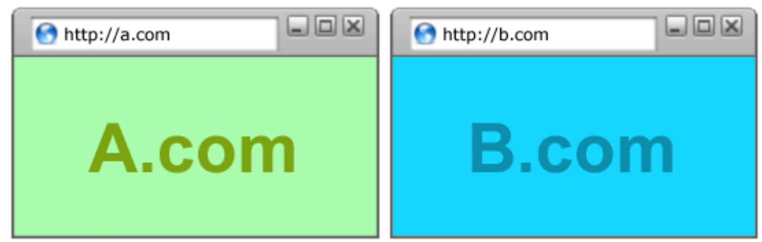


- No mutual trust among parties.
- Confidentiality through Isolation: Same-Origin Policy (SOP)
  - Partition the Web into domains and isolate sensitive data such as cookie, network data and DOM nodes.

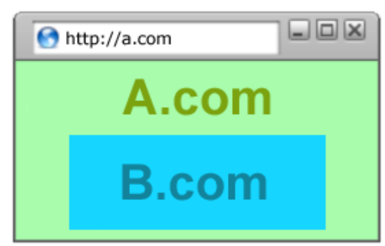
All of These Should Be Safe

- Safe to visit an evil website

- Safe to visit two pages at the same time

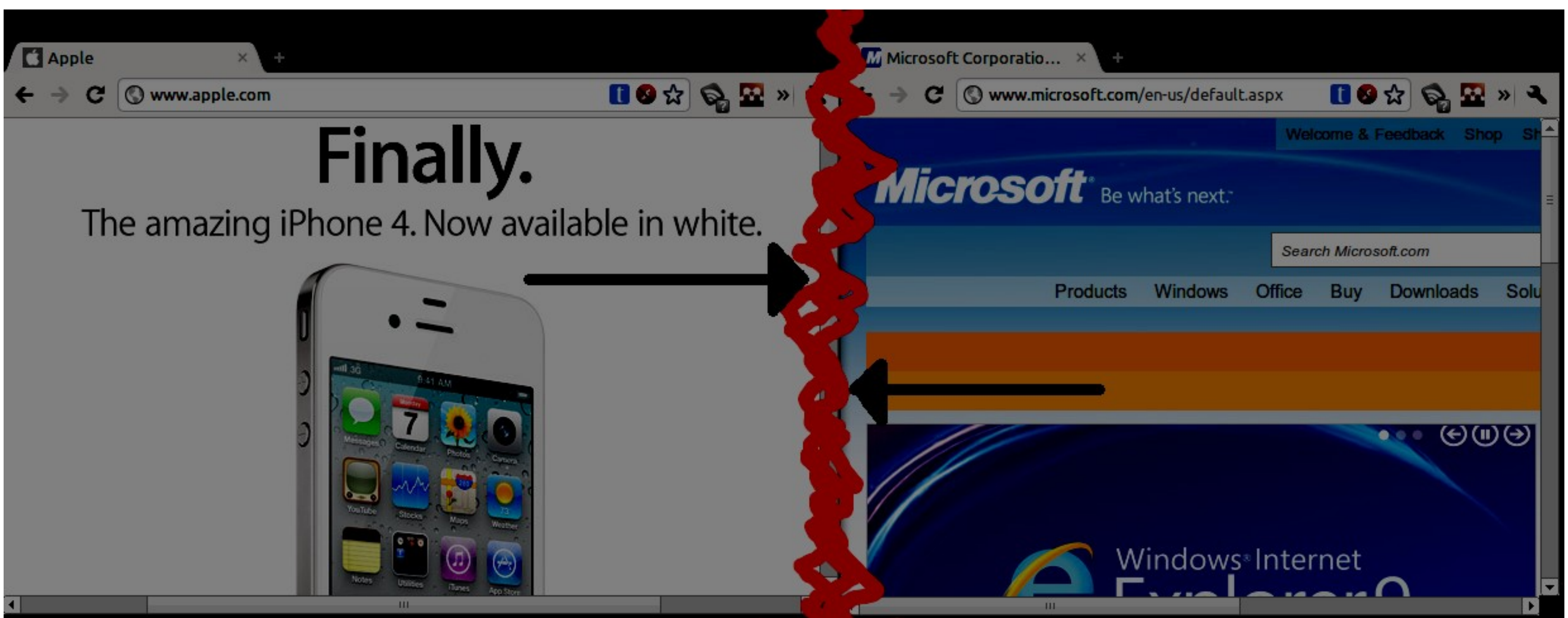


- Safe delegation



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)



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
  - Weakest aspect of SOP, but stricter isolation will prohibit many legitimate uses
- 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 (discussed later)
- Exception: XmlHttpRequests (XHR) 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.
    - Cross-site scripting attacks exploit this!
    - as do script inclusion attacks!
- Plugins implement their own SOP-like policies.
  - Flash keeps its server origin.

Limitations of SOP

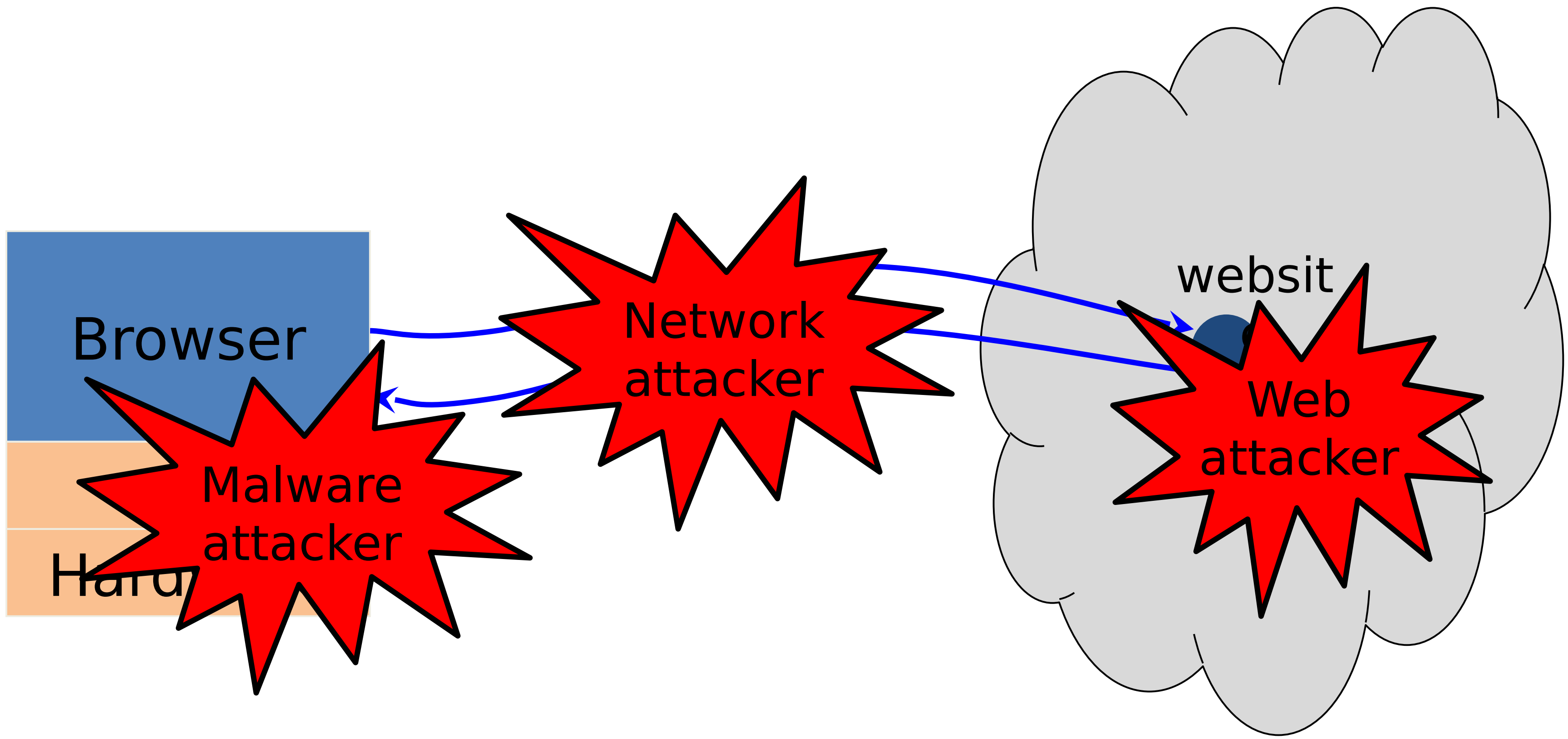
- A very rigid policy that imposes an all-or-nothing approach:
  - The developer can embed the resource (allow all) or open it in an iframe (allow none).
  - Cannot import script libraries without trusting them blindly.
- Does not limit outgoing requests

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?



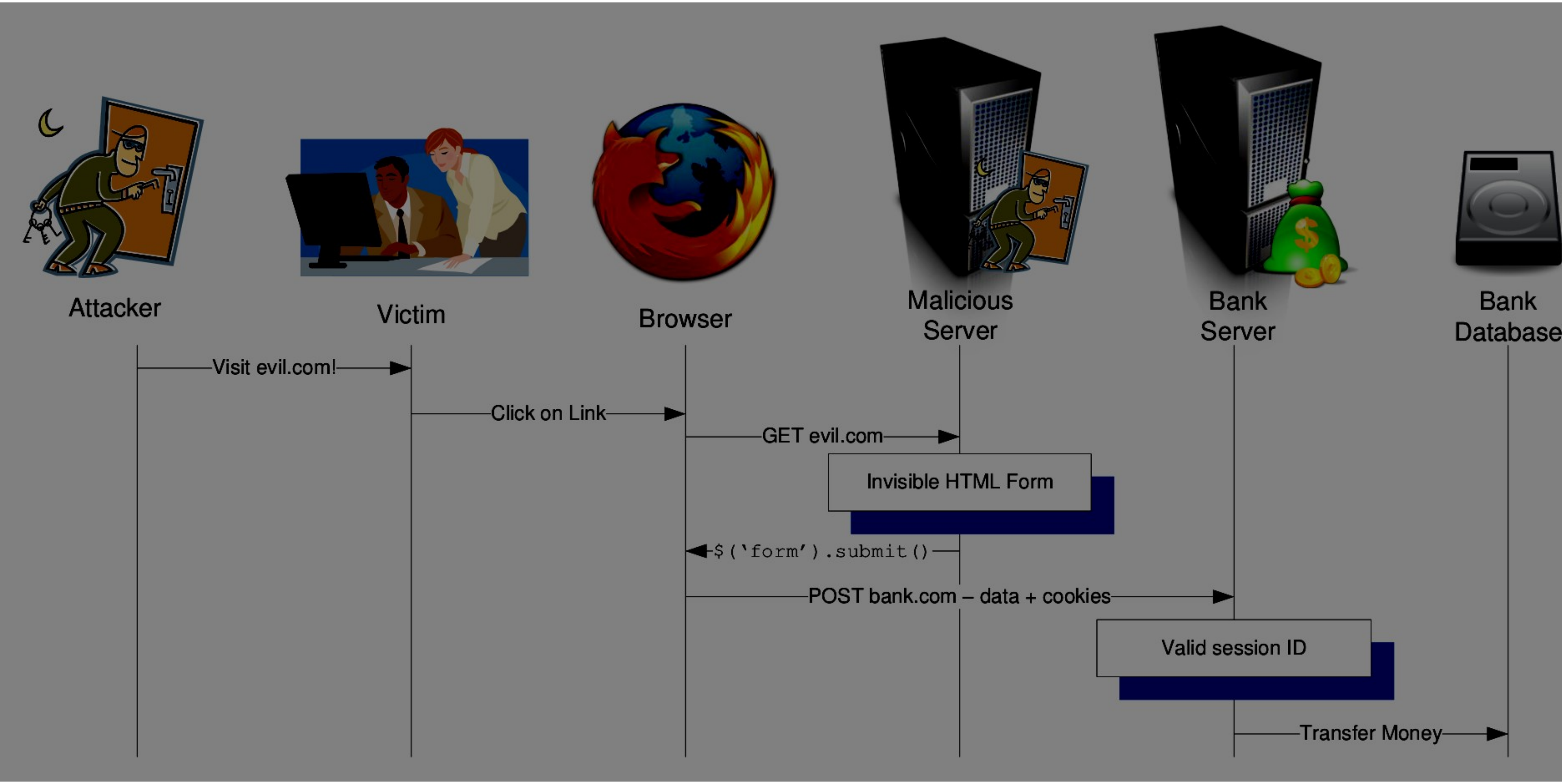
Threat Model 1: Web Attacker

- Benign: User, network *and* the web site
- Malicious: an unrelated web site "attacker.com"
  - Can obtain an SSL/TLS certificate for this site ($0)
- Entices user to visit attacker.com
  - Phishing email, search results, ad, 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

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 methods to distinguish valid requests
  - Inspecting Referrer Headers
  - Validation via User-Provided Secret
  - Validation via Action Token

Inspecting Referrer Headers

- Referrer 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

- Example: 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

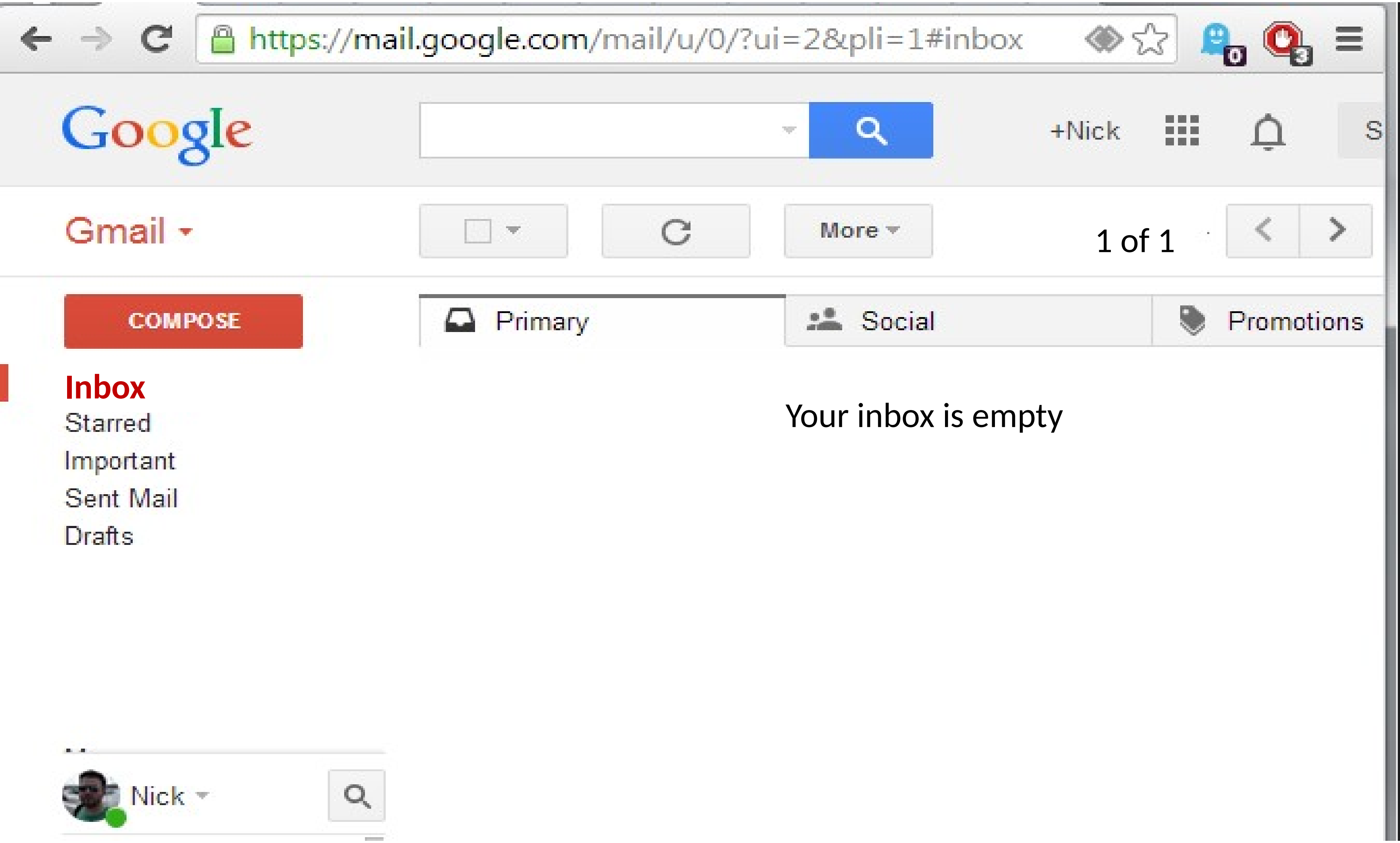
- Add special action tokens as hidden fields to authorized forms to distinguish from forgeries
- Need to generate and validate tokens so that malicious 3rd party can't guess or forge token
  - Token should be a nonce that is unpredictable
  - Same-origin policy prevents 3rd party from inspecting the form to find the token
- This token can be used to distinguish genuine and forged forms

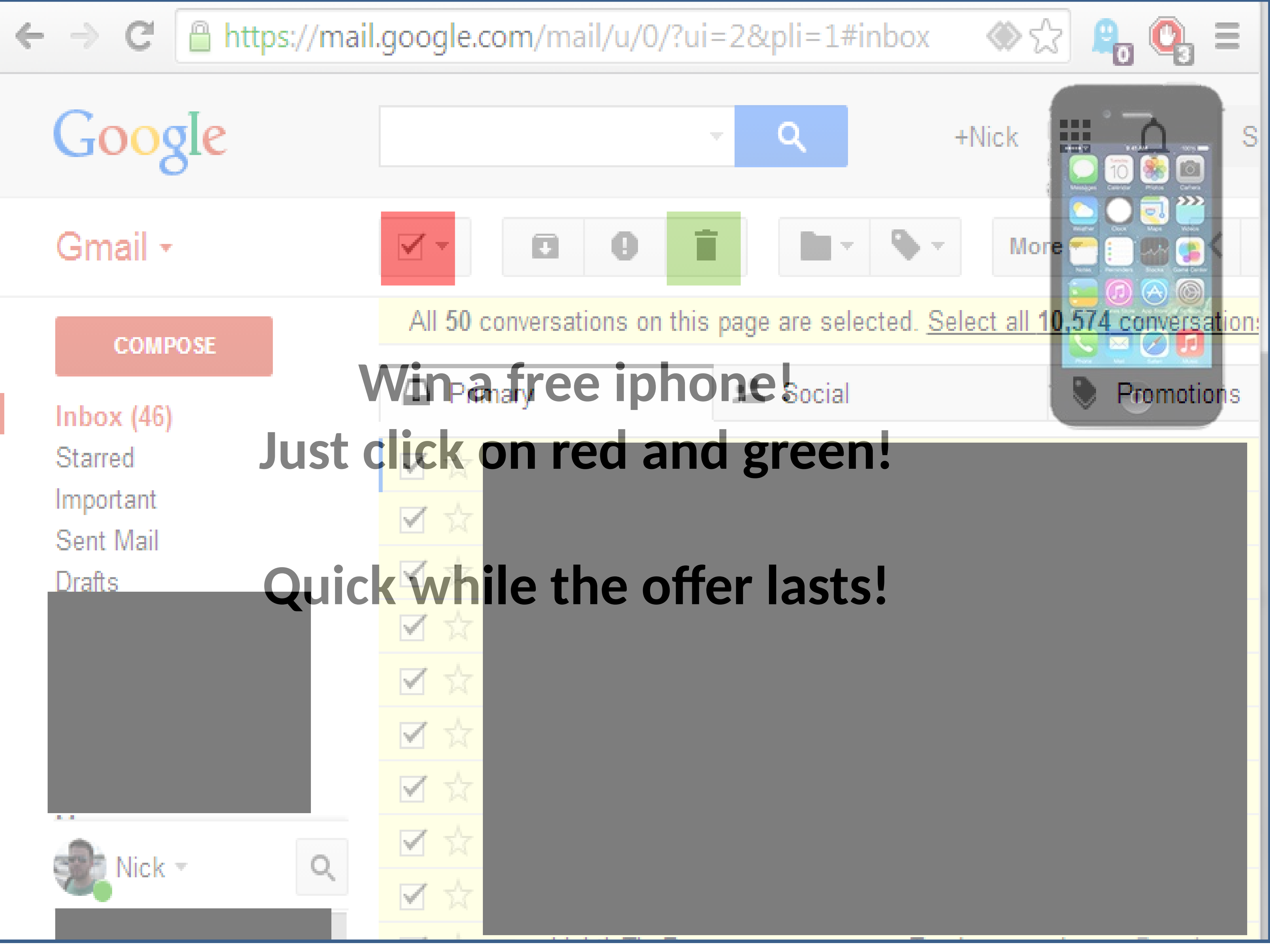
Clickjacking

Win a free iphone!
Just click on red and green!

Quick while the offer lasts!

Where are my mails bro?!?





https://mail.google.com/mail/u/0/?ui=2&pli=1#inbox
Google
+Nick
Gmail
More
All 50 conversations on this page are selected. Select all 10,574
COMPOSE
Primary
Social
Promotions
Inbox (46)
Starred
Important
Sent Mail
Drafts
Nick
Win a free iphone!
Just click on red and green!
Quick while the offer lasts!

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 (supercedes 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)

- Attacker manages to inject his/her script within the page delivered by another site
- 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?

- Short answer: “Almost anything”
  - Mother of all vulnerabilities (subsumes them all)
    - Naturally: this is attacker’s malicious code
- 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

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

```
<HTML>
<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.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
- Easier said than done:
  - Scripts can appear in many contexts
    - <script> tags: inline scripts
    - src attributes: refer to external scripts by name
    - javascript URLs
    - event handlers
    - ...
  - Can be “injected” into non-script fields
    - Usual “close the quotes and new content” trick

Defending against XSS

- Blacklisting
  - E.g. No <, >, script, document.cookie, etc.
  - Intuitively correct, but it should **NOT** be relied upon
    - There are too many ways to insert script content.
    - See XSS Cheat Sheet for 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
  - Defense-in-depth: backup mechanism if primary defenses fail

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-web-we-have/

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?
- Perhaps… by abusing existing trust between the target site and other sites

mybank.com

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

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

- 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?
  -
  - 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
- (Over)simplified 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: *Exposing Private Information by Timing Web Applications*
By Bortz et al.

Web Threat Models

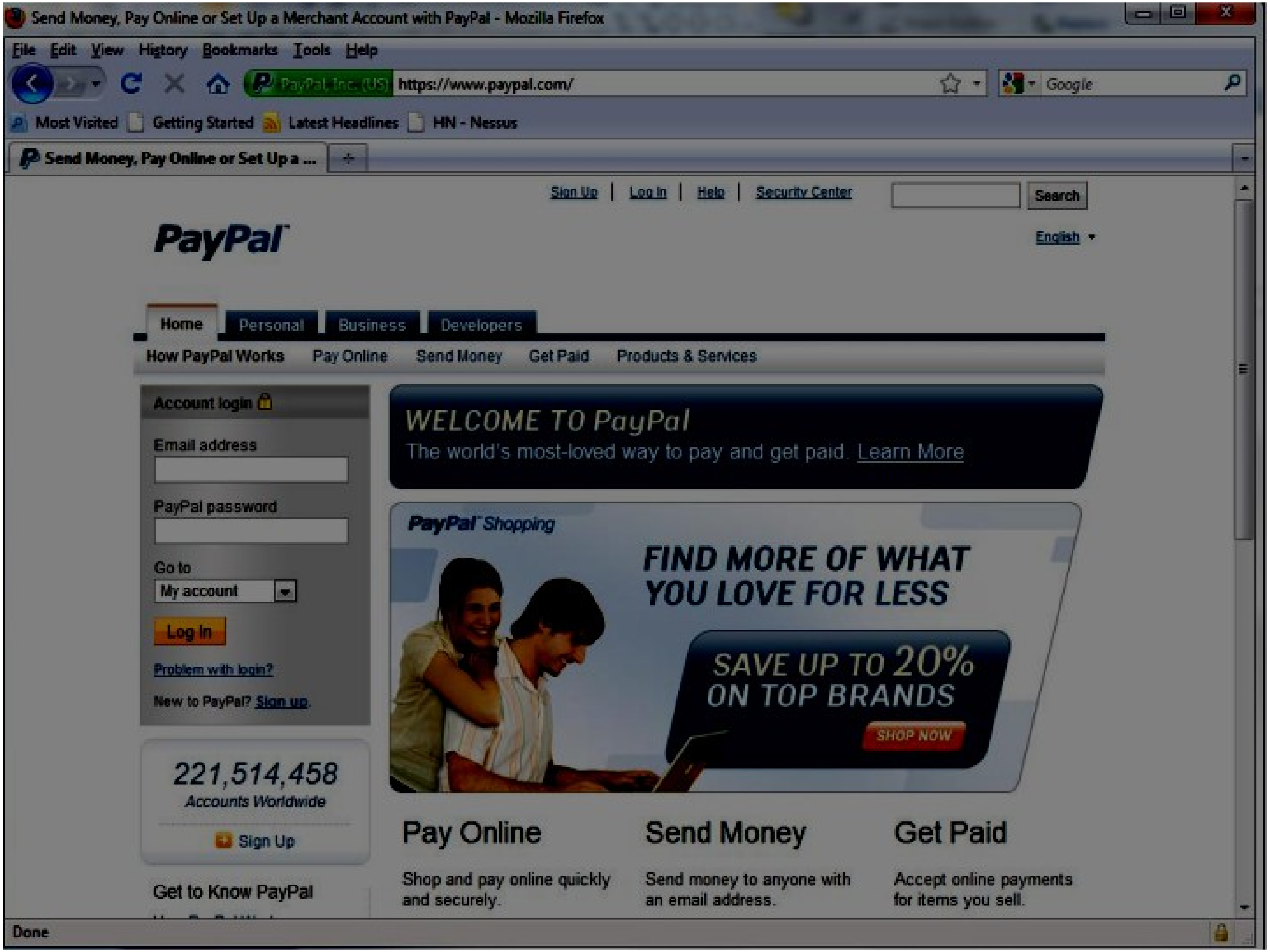
- Web attacker
- Network attacker
  - Passive: wireless eavesdropper
  - Active: evil Wi-Fi router, DNS poisoning
- Malware attacker
  - Malicious code executes directly on victim's computer
  - To infect victim's computer, can exploit software bugs (e.g., buffer overflow) or convince user to install malicious content (how?)
    - Masquerade as an antivirus program, video codec, etc.

SSL Stripping

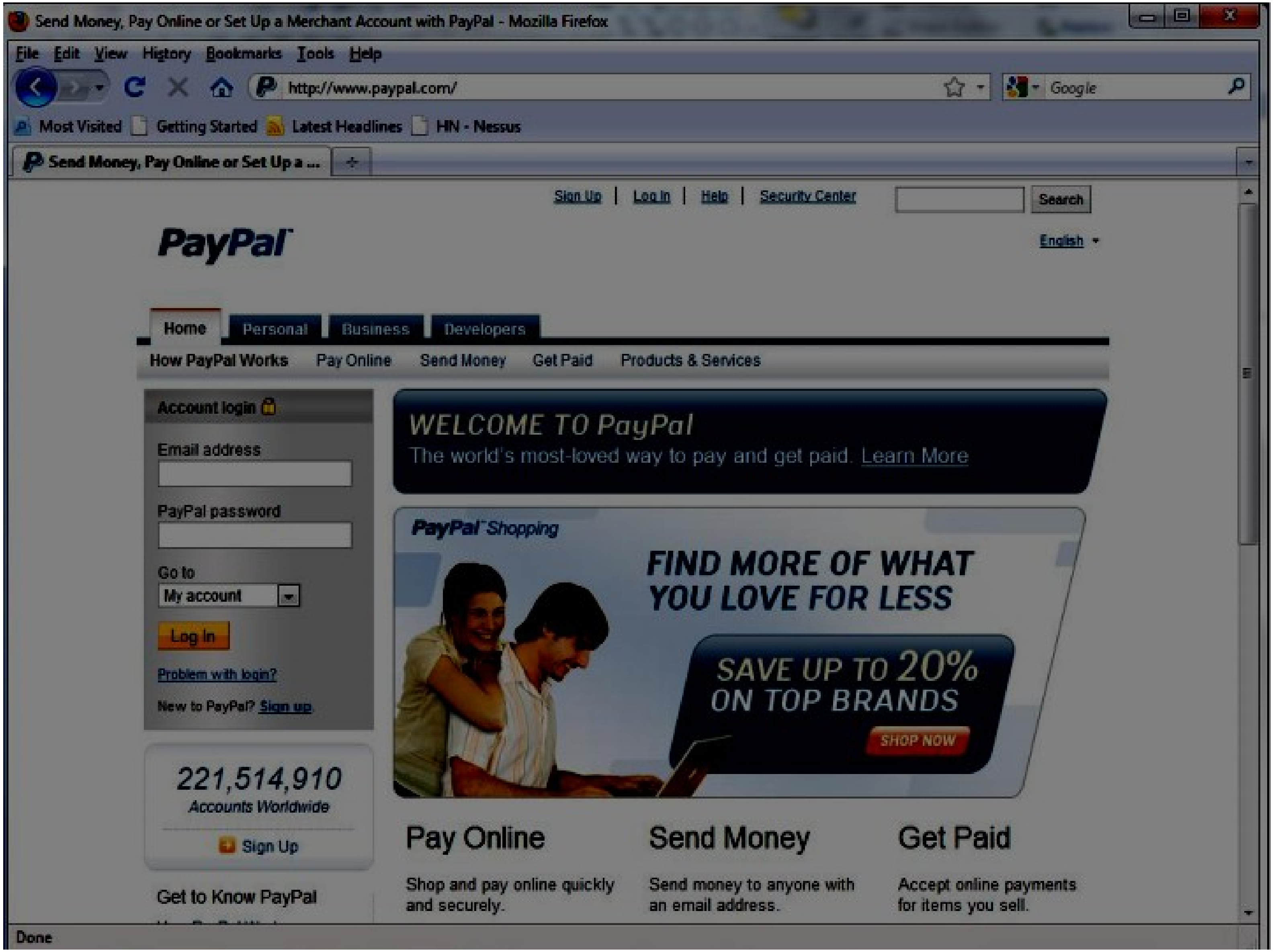
- Let's say that a website exists only over HTTPS
  - No HTTP pages

- 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



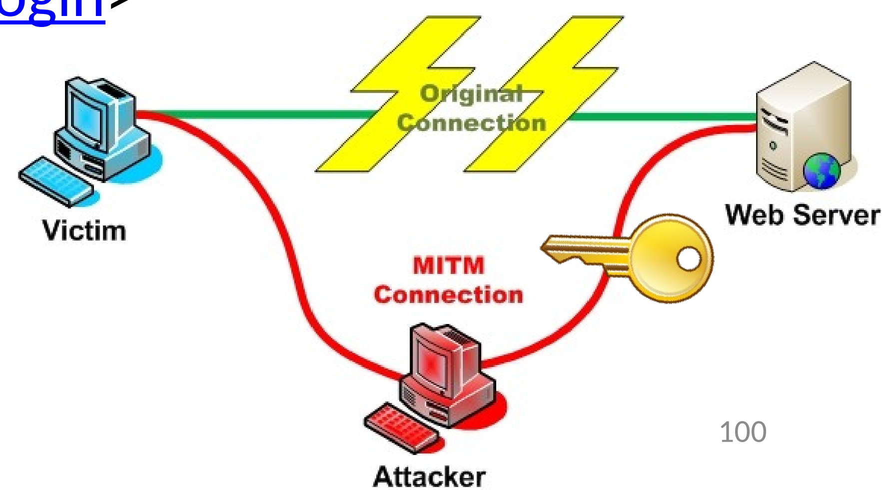
Page load when attacker is present



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 browser-vendor-updated 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: (URI-encodes newlines)

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

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
  - Step 2 can only be performed after Step 1
  - 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 by 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 in the browser or JS engine
- 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
  - Heapspray, JIT-spray and JIT-ROP 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