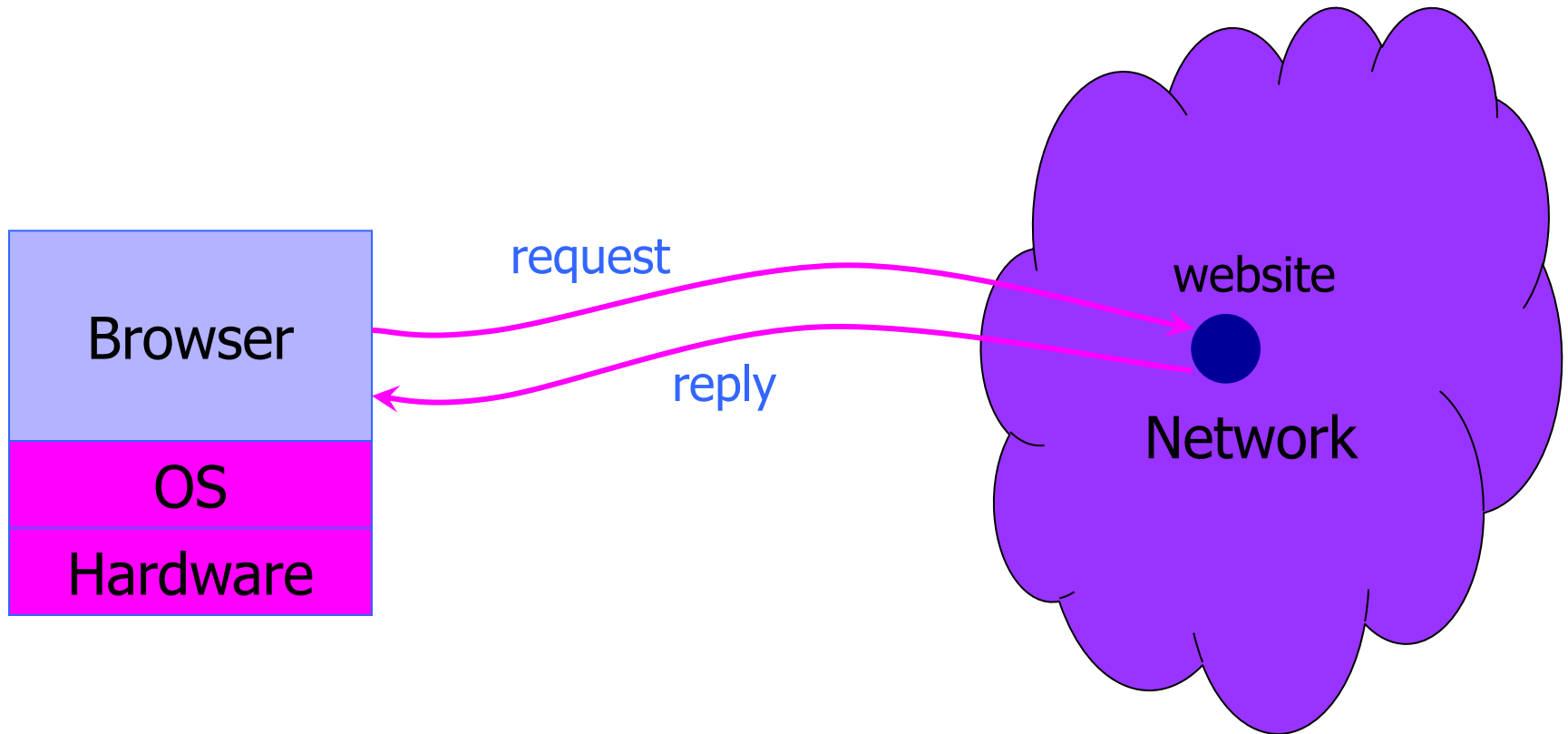


Same Origin Policy

Vitaly Shmatikov



Browser and Network



Two Sides of Web Security

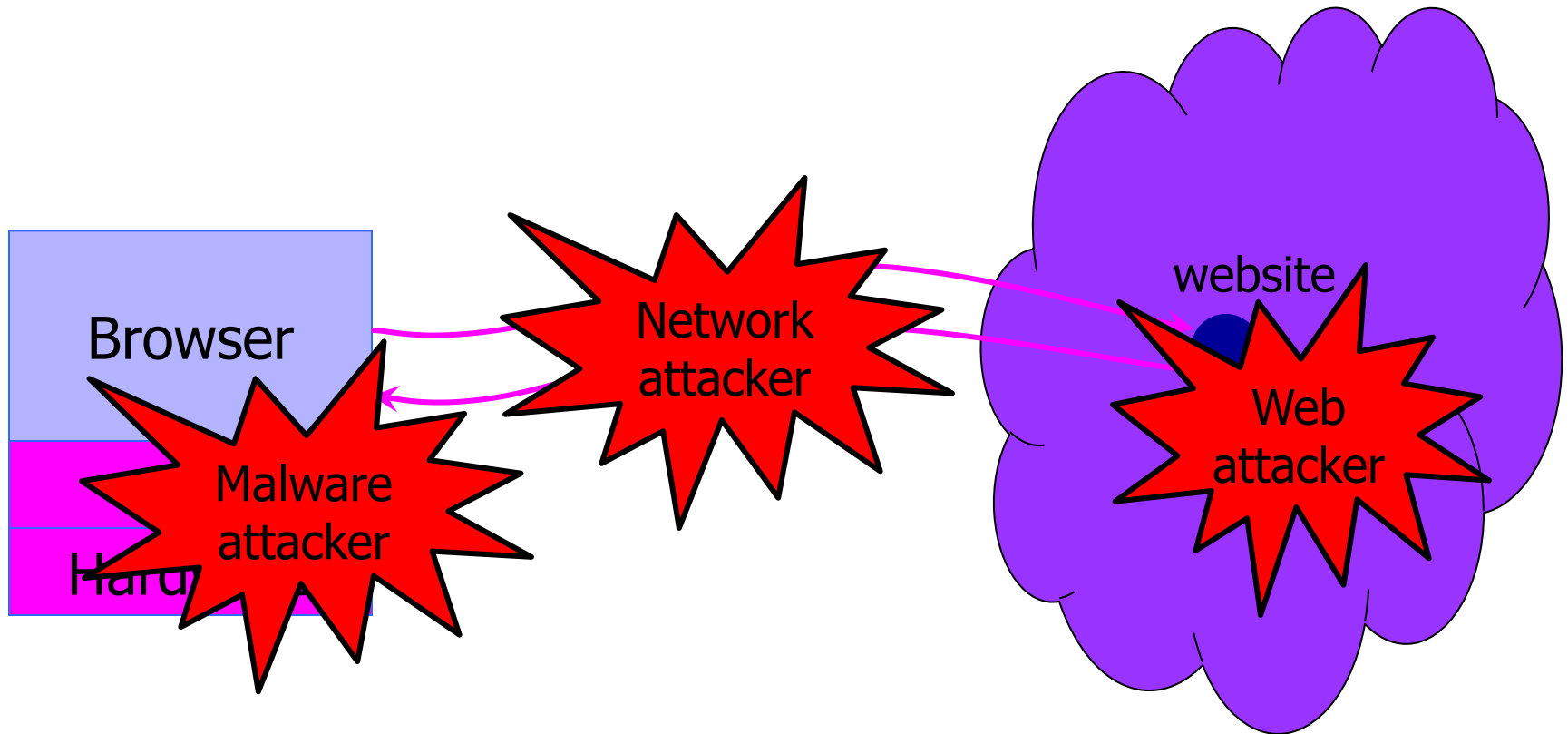
◆ Web browser

- Responsible for securely confining Web content presented by visited websites

◆ Web applications

- Online merchants, banks, blogs, Google Apps ...
- Mix of server-side and client-side code
 - Server-side code written in PHP, Ruby, ASP, JSP... runs on the Web server
 - Client-side code written in JavaScript... runs in the Web browser
- Many potential bugs: XSS, XSRF, SQL injection

Where Does the Attacker Live?



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.

Web Attacker

- ◆ Controls a malicious website (attacker.com)
 - Can even obtain an SSL/TLS certificate for his site (\$0)
- ◆ User visits attacker.com – why?
 - 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.

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 iframe on the same webpage

◆ Support secure Web applications

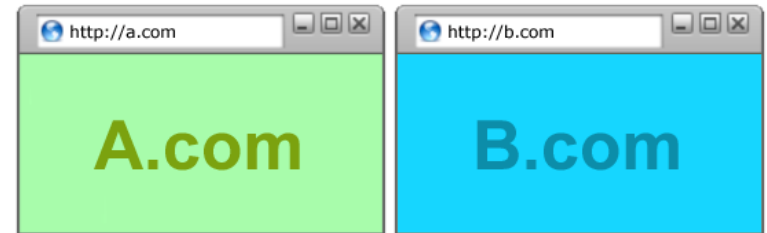
- Applications delivered over the Web should have the same security properties as required for standalone applications (what are these properties?)

All of These Should Be Safe

◆ Safe to visit an evil website



◆ Safe to visit two pages at the same time



◆ Safe delegation



OS vs. Browser Analogies

Operating system

◆ Primitives

- System calls
- Processes
- Disk

◆ Principals: Users

- Discretionary access control

◆ Vulnerabilities

- Buffer overflow
- Root exploit

Web browser

◆ Primitives

- Document object model
- Frames
- Cookies and localStorage

◆ Principals: “Origins”

- Mandatory access control

◆ Vulnerabilities

- Cross-site scripting
- Universal scripting

Browser: Basic Execution Model

◆ Each browser window or frame:

- Loads content
- Renders
 - Processes HTML and scripts to display the page
 - May involve images, subframes, etc.
- Responds to **events**

◆ Events

- User actions: OnClick, OnMouseover
- Rendering: OnLoad, OnUnload
- Timing: setTimeout(), clearTimeout()

JavaScript

- ◆ “The world’s most misunderstood programming language”
- ◆ Language executed by the browser
 - Scripts are embedded in Web pages
 - Can run before HTML is loaded, before page is viewed, while it is being viewed, or when leaving the page
- ◆ Used to implement “active” web pages
 - AJAX, huge number of Web-based applications
- ◆ Potentially malicious website 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
 - Mozilla’s SpiderMonkey and Rhino, several others

JavaScript in Web Pages

◆ Embedded in HTML page as `<script>` element

- JavaScript written directly inside `<script>` element
 - `<script> alert("Hello World!") </script>`
- Linked file as `src` attribute of the `<script>` element
`<script type="text/JavaScript" src="functions.js"></script>`

◆ Event handler attribute

``

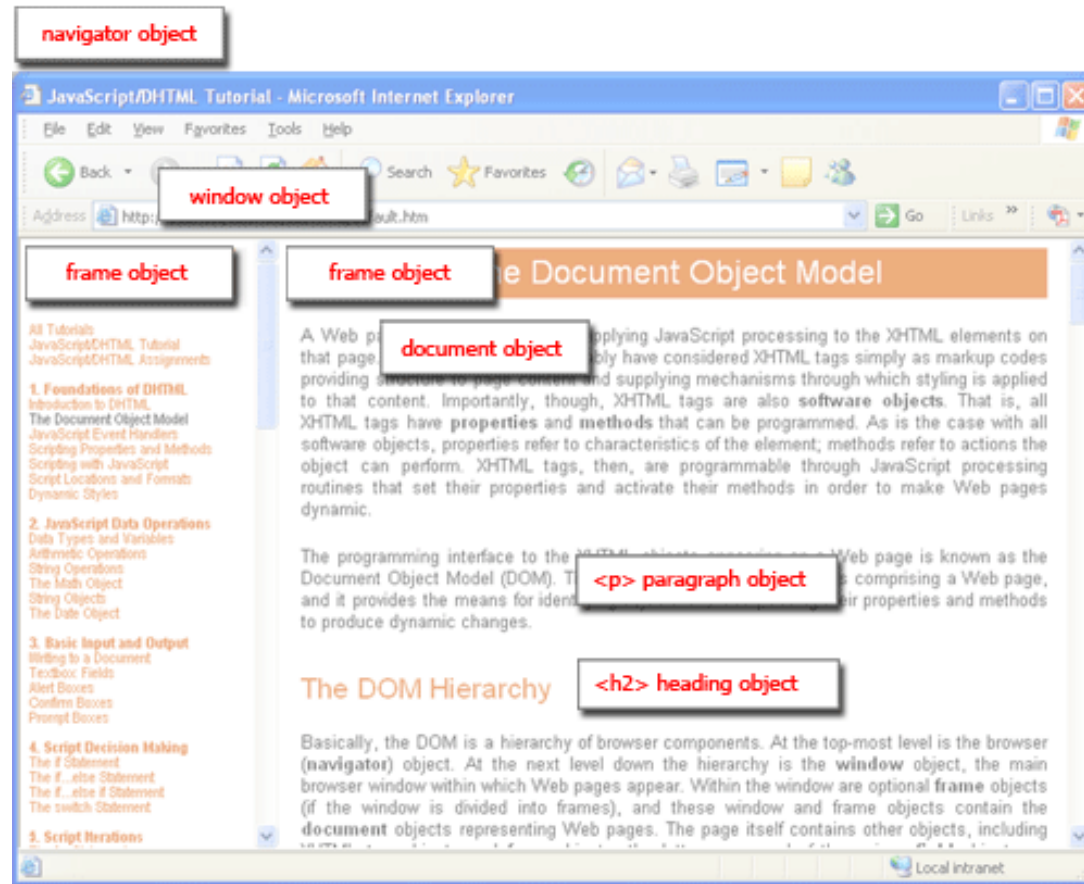
◆ Pseudo-URL referenced by a link

`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



W3C standard differs from models supported in existing browsers

Event-Driven Script Execution

```
<script type="text/javascript">  
  function whichButton(event) {  
    if (event.button==1) {  
      alert("You clicked the left mouse button!") }  
    else {  
      alert("You clicked the right mouse button!")  
    }  
  }  
</script>
```

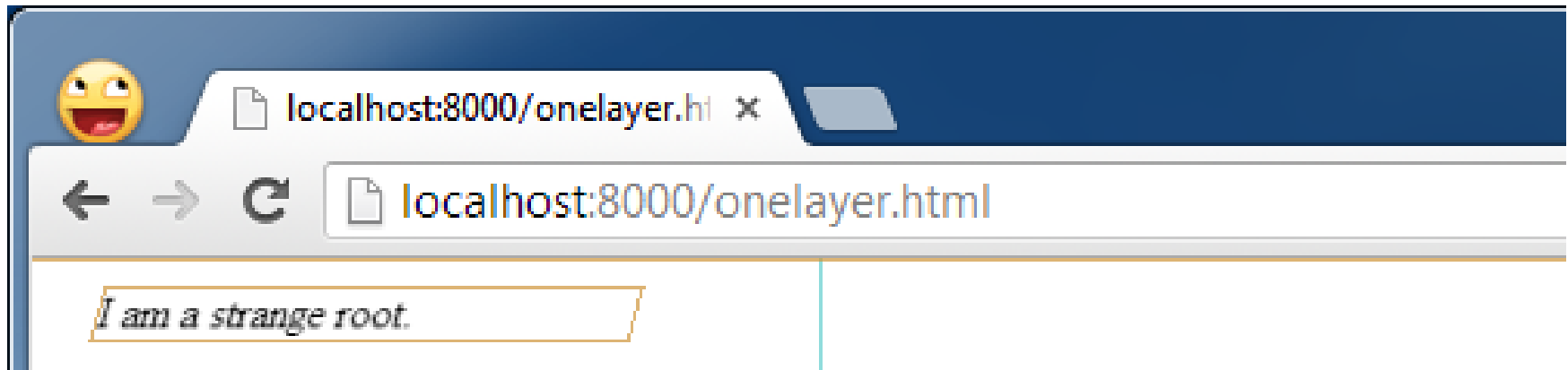
Script defines a page-specific function

Function gets executed when some event happens

```
...  
<body onmousedown="whichButton(event)">  
...  
</body>
```



```
<html>
  <body>
    <div style="-webkit-transform: rotateY(30deg)
      rotateX(-30deg); width: 200px;">
      I am a strange root.
    </div>
  </body>
</html>
```



Source: <http://www.html5rocks.com/en/tutorials/speed/layers/>

JavaScript Bookmarks (Favelets)

- ◆ Script stored by the browser as a bookmark
- ◆ Executed in the context of the current webpage
- ◆ Typical uses:
 - Submit the current page to a blogging or bookmarking service
 - Query a search engine with highlighted text
 - Password managers
 - One-click sign-on
 - Automatically generate a strong password
 - Synchronize passwords across sites

 Must execute only inside the "right" page

A JavaScript “Rootkit”

[Adida, Barth, Jackson.
“Rootkits for JavaScript
environments”. WOOT 2009]

```
if (window.location.host == "bank.com")  
  doLogin(password);
```

JavaScript bookmark

Malicious page defines a global variable named
“window” whose value is a fake “location” object
`var window = { location: { host: "bank.com" } };`



A malicious webpage

Let's Detect Fake Objects

["Rootkits for JavaScript environments"]

```
window.location = "#";
```

If window.location is a native object,
new value will be "https://bank.com/login#"

JavaScript bookmark

```
window.__defineGetter__("location",  
    function () { return "https://bank.com/login#"; });  
window.__defineSetter__("location", function (v) { });
```



A malicious webpage

Let's Detect Emulation

[“Rootkits for JavaScript environments”]

Use reflection API

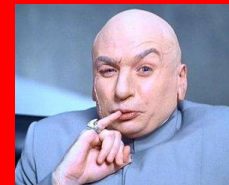
```
typeof obj.__lookupGetter__(propertyName)
!== "undefined"
```

↖️ `typeof` and `!==` avoid asking for the value of
“undefined” (could be redefined by attacker!)

JavaScript bookmark

Attacker emulates reflection API itself!

```
Object.prototype.__lookupGetter__ =  
function() { ... };
```



A malicious webpage

Content Comes from Many Sources

◆ Scripts

```
<script src="//site.com/script.js"> </script>
```

◆ Frames

```
<iframe src="//site.com/frame.html"> </iframe>
```

◆ Stylesheets (CSS)

```
<link rel="stylesheet" type="text/css" href="//site.com/theme.css" />
```

◆ Objects (Flash) - using swfobject.js script

```
<script> var so = new SWFObject('//site.com/flash.swf', ...);  
        so.addParam('allowscriptaccess', 'always');  
        so.write('flashdiv');  
</script>
```

Allows Flash object to communicate with external scripts, navigate frames, open windows

Browser Sandbox



- ◆ Goal: safely execute JavaScript code provided by a remote website
 - No direct file access, limited access to OS, network, browser data, content that came from other websites
- ◆ Same origin policy (SOP)
 - Can only read properties of documents and windows from the same protocol, domain, and port
- ◆ User can grant privileges to signed scripts
 - UniversalBrowserRead/Write, UniversalFileRead, UniversalSendMail

SOP Often Misunderstood

[Jackson and Barth.
"Beware of Finer-
Grained Origins".
W2SP 2008]

- ◆ Often simply stated as "same origin policy"
 - This usually just refers to "can script from origin A access content from origin B"?
- ◆ Full policy of current browsers is complex
 - Evolved via "penetrate-and-patch"
 - Different features evolved slightly different policies
- ◆ Common scripting and cookie policies
 - Script access to DOM considers protocol, domain, port
 - Cookie reading considers protocol, domain, path
 - Cookie writing considers domain

Same Origin Policy

protocol://domain:port/path?params

Same Origin Policy (SOP) for DOM:

Origin A can access origin B's DOM if A and B have same **(protocol, domain, port)**

Same Origin Policy (SOP) for cookies:

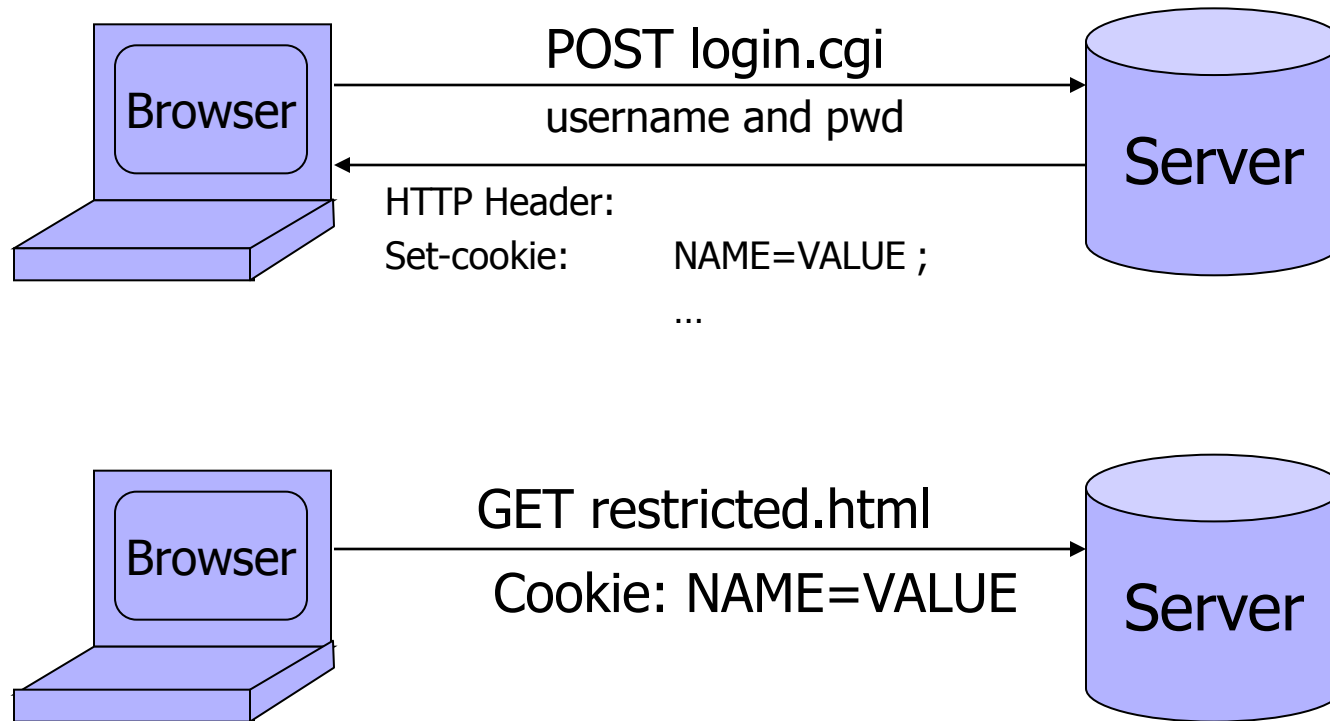
Generally, based on
([protocol], domain, path)

optional



Website Storing Info in Browser

A **cookie** is a file created by a website to store information in the browser



HTTP is a stateless protocol; cookies add state

What Are Cookies Used For?

◆ Authentication

- The cookie proves to the website that the client previously authenticated correctly

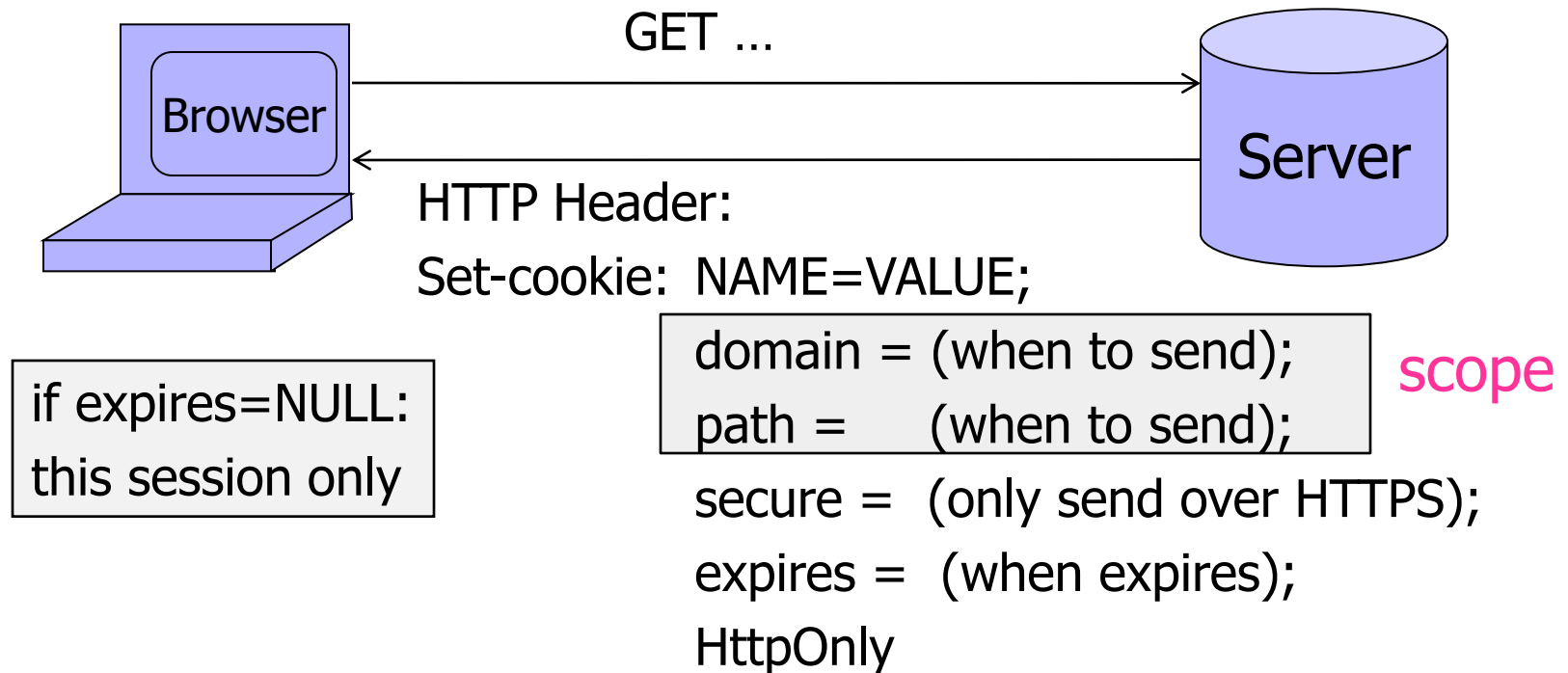
◆ Personalization

- 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

Setting Cookies by Server



- Delete cookie by setting “expires” to date in past
- Default scope is domain and path of setting URL

SOP for Writing Cookies

domain: any domain suffix of URL-hostname,
except top-level domain (TLD)

Which cookies can be set by **login.site.com**?

allowed domains

- ✓ **login.site.com**
- ✓ **.site.com**

disallowed domains

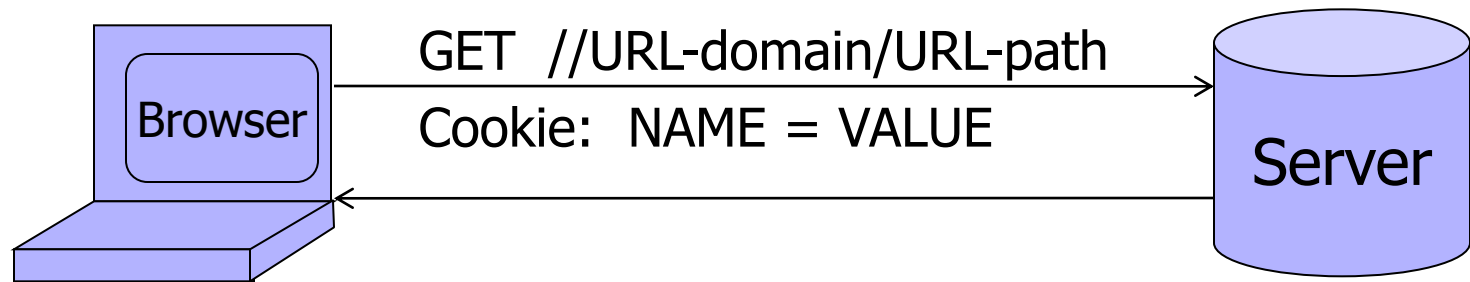
- ✗ **user.site.com**
- ✗ **othersite.com**
- ✗ **.com**

login.site.com can set cookies for all of **.site.com**
but not for another site or TLD

Problematic for sites like .cornell.edu

path: anything

SOP for Reading Cookies



Browser sends all cookies in URL scope:

- cookie-domain is domain-suffix of URL-domain
- cookie-path is prefix of URL-path
- protocol=HTTPS if cookie is "secure"

Examples of Cookie Reading SOP

cookie 1

name = **userid**

value = u1

domain = **login.site.com**

path = /

secure

cookie 2

name = **userid**

value = u2

domain = **.site.com**

path = /

non-secure

both set by **login.site.com**

http://checkout.site.com/

http://login.site.com/

https://login.site.com/

cookie: userid=u2

cookie: userid=u2

cookie: userid=u1; userid=u2

(arbitrary order; in FF3 most specific first)

Cookie Protocol Issues

- ◆ What does the server know about the cookie sent to it by the browser?
- ◆ Server only sees **Cookie: Name=Value**
 - ... does not see cookie attributes (e.g., "secure")
 - ... does not see which domain set the cookie
 - RFC 2109 (cookie RFC) has an option for including domain, path in Cookie header, but not supported by browsers

Overwriting "Secure" Cookies

◆ Alice logs in at <https://www.google.com>

Set-Cookie: LSID=EXPIRED;Domain=.google.com;Path=/;Expires=Mon, 01-Jan-1990 00:00:00 GMT

Set-Cookie: LSID=EXPIRED;Path=/;Expires=Mon, 01-Jan-1990 00:00:00 GMT

Set-Cookie: LSID=EXPIRED;Domain=www.google.com;Path=/accounts;Expires=Mon, 01-Jan-1990 00:00:00 GMT

Set-Cookie: LSID=cl:DQAAAHsAAACn3h7GCpKUNxckr79Ce3BUCJtluaI9a7e5oPvByTrOHUQiFjECYqr5r0q2cH1Cqk

Set-Cookie: GAUSR=dabo123@gmail.com;Path=/accounts;Secure

LSID, GAUSR are
"secure" cookies

◆ Alice visits <http://www.google.com>

- Automatically, due to the phishing filter

◆ Network attacker can inject into response

Set-Cookie: LSID=badguy; secure

- Browser thinks this cookie came from <http://google.com>, allows it to overwrite secure cookie

Surf Jacking

 <http://resources.enablesecurity.com/resources/Surf%20Jacking.pdf>

- ◆ Victim logs into **https://bank.com** using HTTPS
 - Non-secure cookie sent back, but protected by HTTPS
- ◆ Victim visits **http://foo.com** in another window
- ◆ Network attacker sends “301 Moved Permanently” in response to cleartext request to foo.com
 - Response contains header “Location **http://bank.com**”
 - Browser thinks foo.com is redirected to bank.com
- ◆ Browser starts a new HTTP connection to bank.com, sends cookie in the clear
- ◆ Network attacker gets the cookie!

SOP for JavaScript in Browser

- ◆ Same domain scoping rules as for sending cookies to the server
- ◆ **document.cookie** returns a string with all cookies available for the document
 - Often used in JavaScript to customize page
- ◆ Javascript can set and delete cookies via DOM
 - `document.cookie = "name=value; expires=...; "`
 - `document.cookie = "name=; expires= Thu, 01-Jan-70"`

Path Separation Is Not Secure

Cookie SOP: path separation

when the browser visits **x.com/A**,
it does not send the cookies of **x.com/B**
This is done for efficiency, not security!

DOM SOP: no path separation

A script from **x.com/A** can read DOM of **x.com/B**

```
<iframe src="x.com/B"></iframe>  
alert(frames[0].document.cookie);
```

Frames

◆ Window may contain frames from different sources

- frame: rigid division as part of frameset
- iframe: floating inline frame

```
<IFRAME SRC="hello.html" WIDTH=450 HEIGHT=100>
```

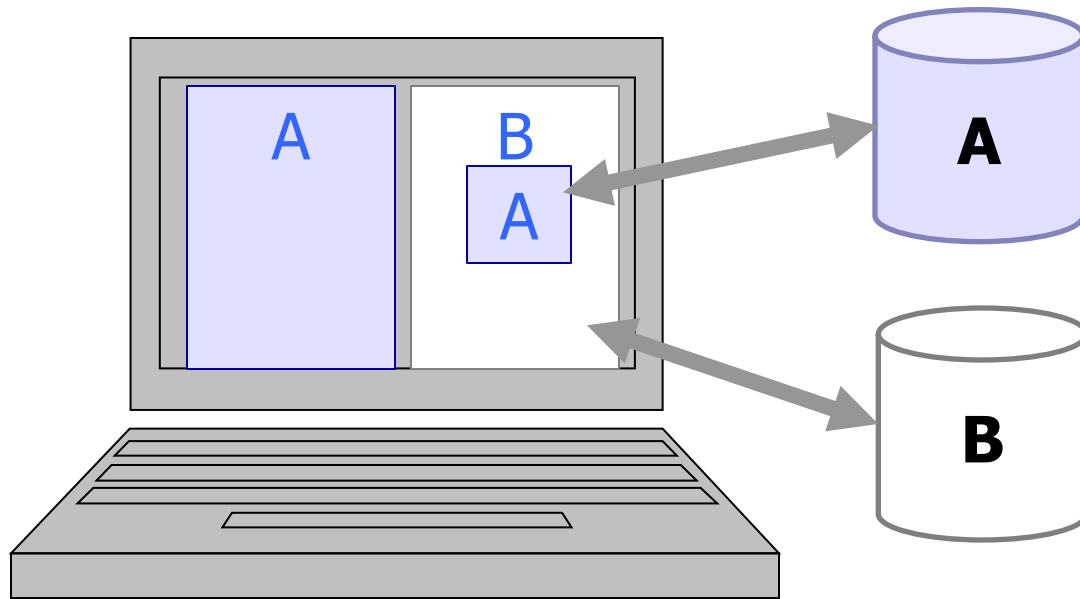
If you can see this, your browser doesn't understand IFRAME.

```
</IFRAME>
```

◆ Why use frames?

- Delegate screen area to content from another source
- Browser provides isolation based on frames
- Parent may work even if frame is broken

Browser Security Policy for Frames



- ◆ Each frame of a page has an origin
 - Origin = protocol://domain:port
- ◆ Frame can access objects from its own origin
 - Network access, read/write DOM, cookies and localStorage
- ◆ Frame cannot access objects associated with other origins

Cross-Frame Scripting

- ◆ Frame A can execute a script that manipulates arbitrary DOM elements of Frame B **only if**
 $\text{Origin}(A) = \text{Origin}(B)$
 - Basic same origin policy, where origin is the protocol, domain, and port from which the frame was loaded
- ◆ Some browsers used to allow any frame to navigate any other frame
 - Navigate = change where the content in the frame is loaded from
 - Navigation does not involve reading the frame's old content

Frame SOP Examples

Suppose the following HTML is hosted at site.com

◆ Disallowed access

```
<iframe src="http://othersite.com"></iframe>  
alert( frames[0].contentDocument.body.innerHTML )  
alert( frames[0].src )
```

◆ Allowed access

```

```

```
alert( images[0].height )
```

or

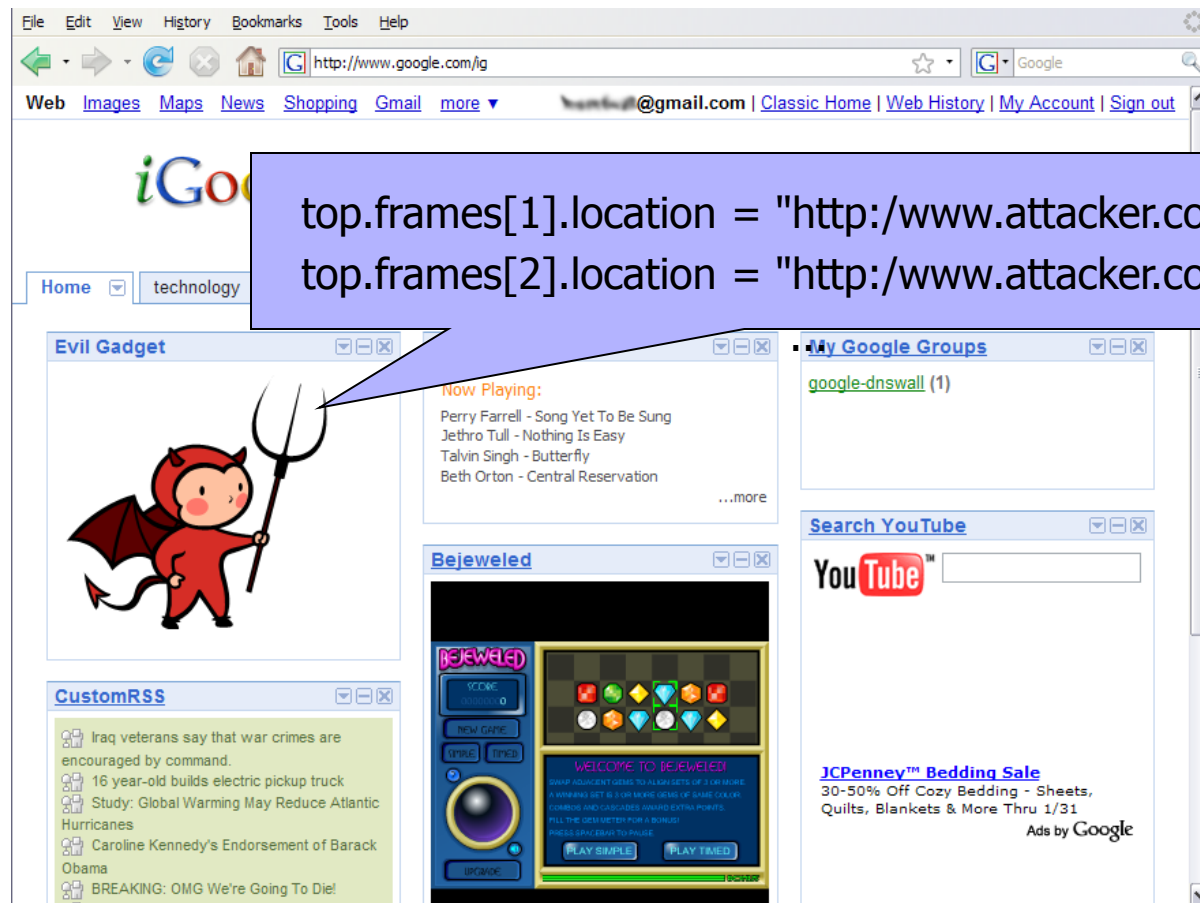
```
frames[0].location.href = "http://mysite.com/"
```

Navigating child frame is allowed,
but reading frame[0].src is not

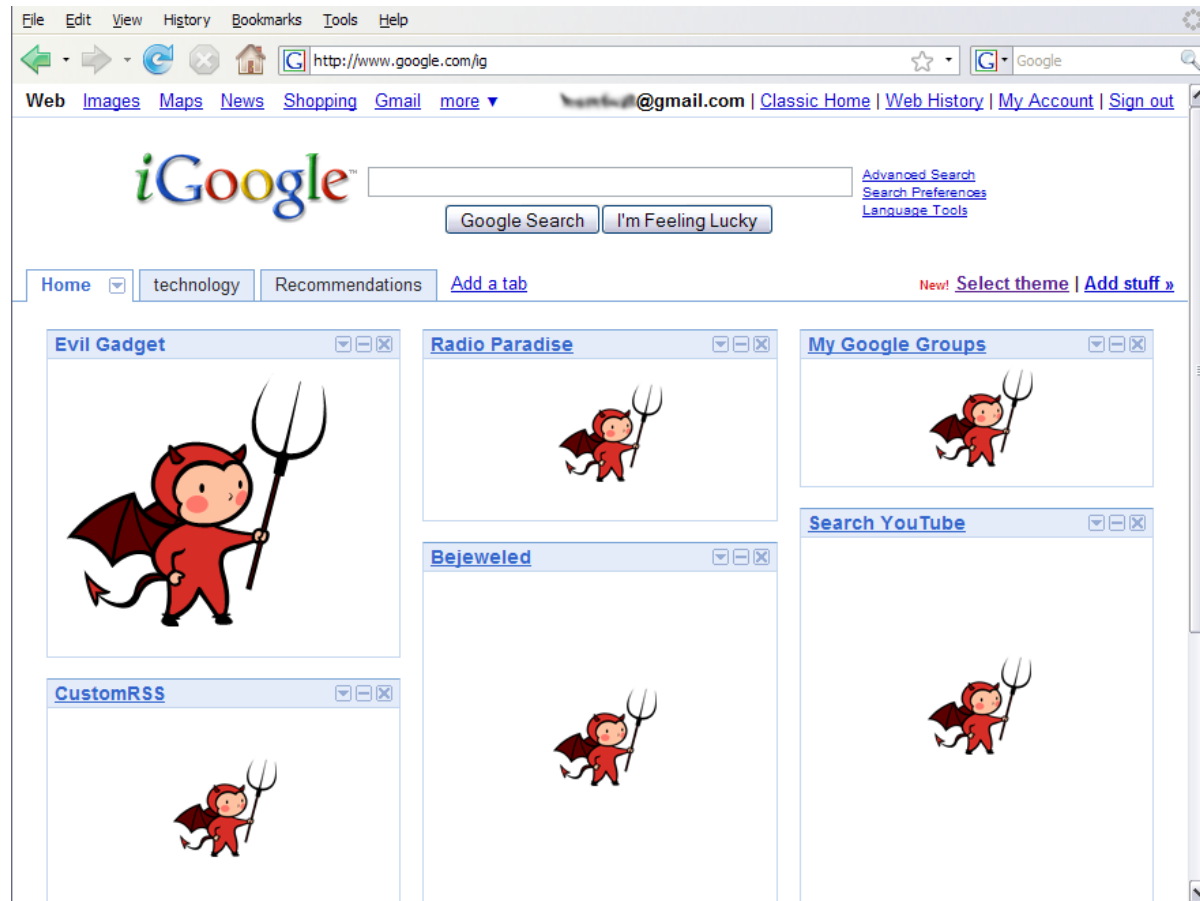


slide 41

Gadget Hijacking in Mashups



Gadget Hijacking



Modern browsers only allow a frame to navigate its “descendant” frames

Recent Developments

◆ Cross-origin network requests

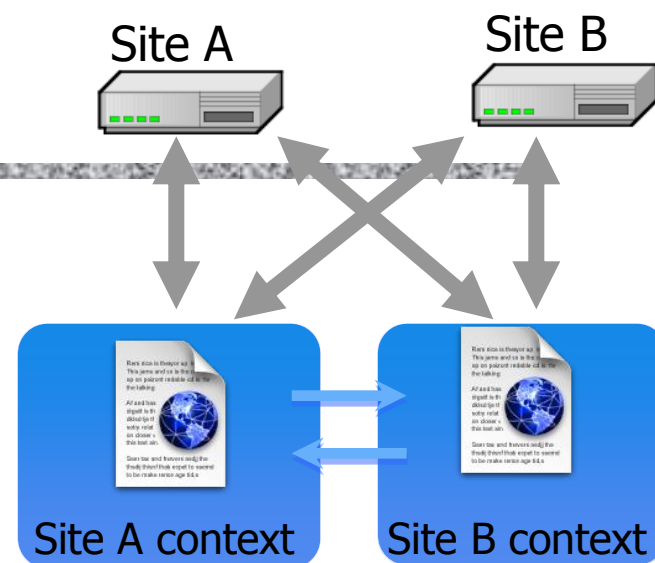
- Access-Control-Allow-Origin:
<list of domains>

– Typical usage:

Access-Control-Allow-Origin: *

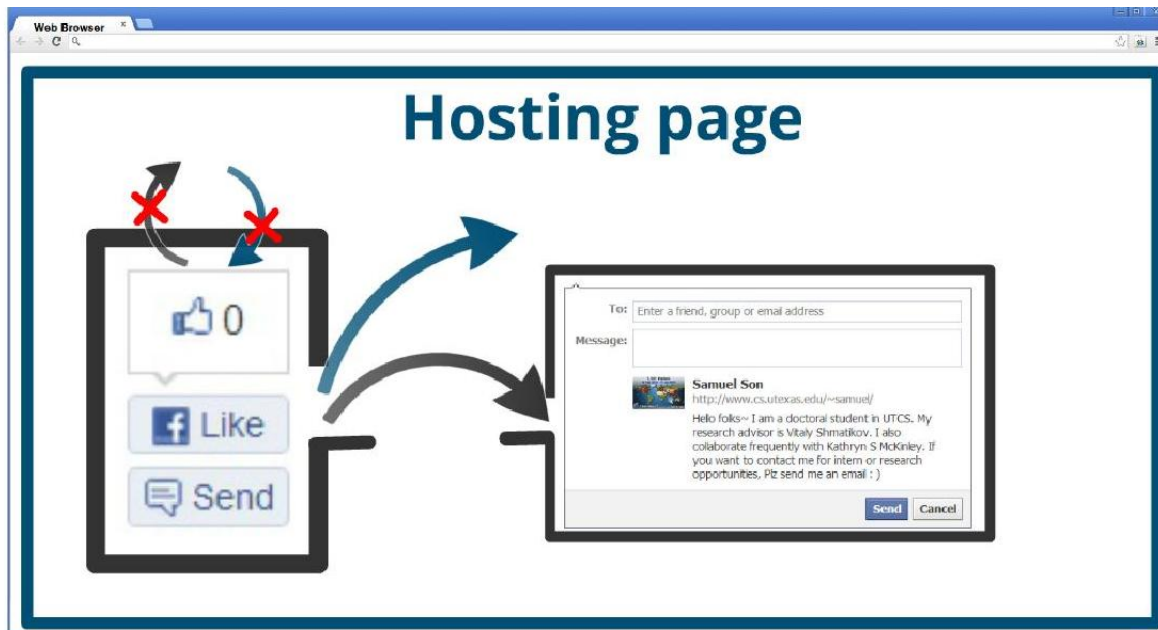
◆ Cross-origin client-side communication

- Client-side messaging via fragment navigation
- postMessage (newer browsers)

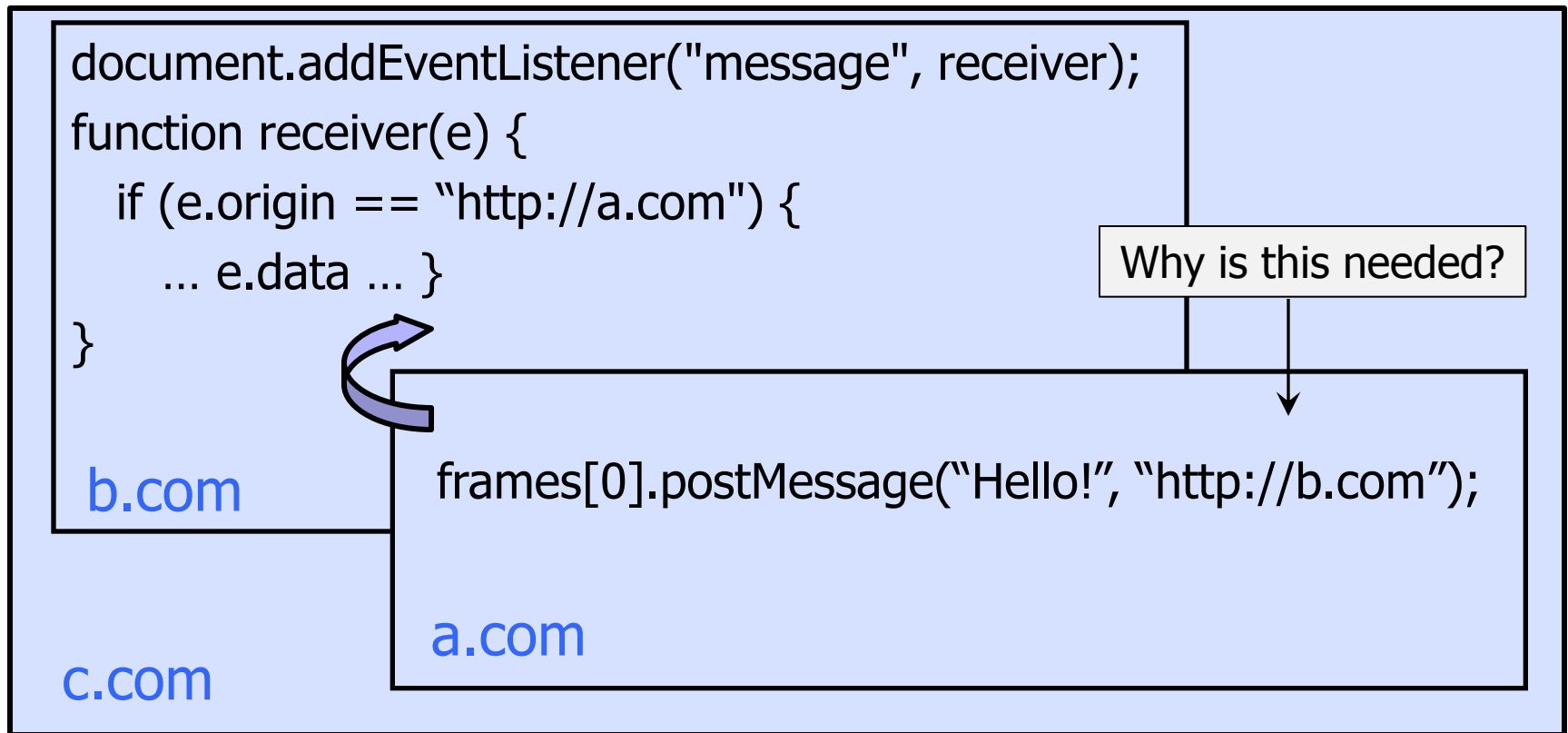


postMessage

- ◆ New API for inter-frame communication
- ◆ Supported in latest browsers



Example of postMessage Usage

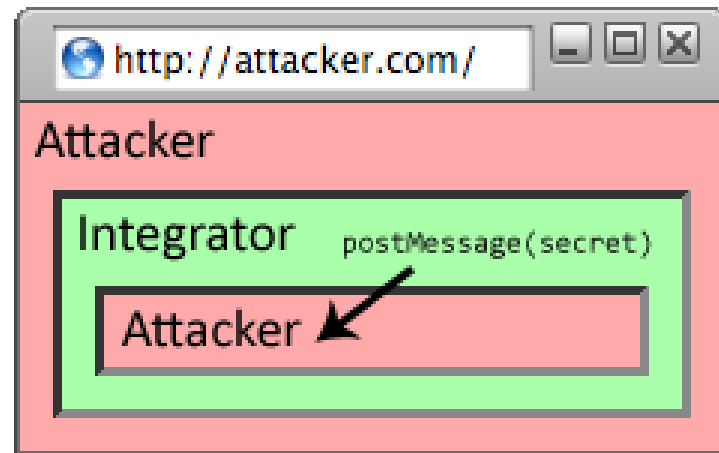
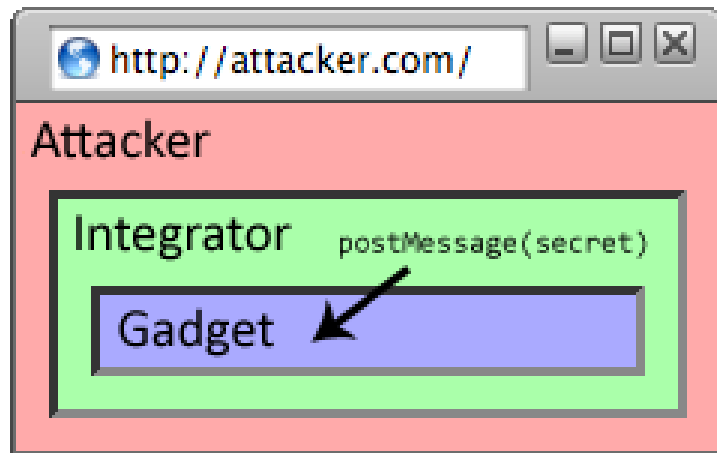


Messages are sent to frames, not origins

Message Eavesdropping (1)

`frames[0].postMessage("Hello!")`

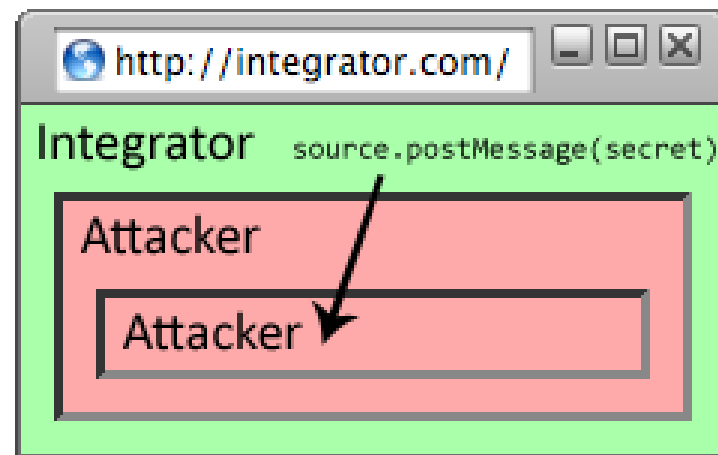
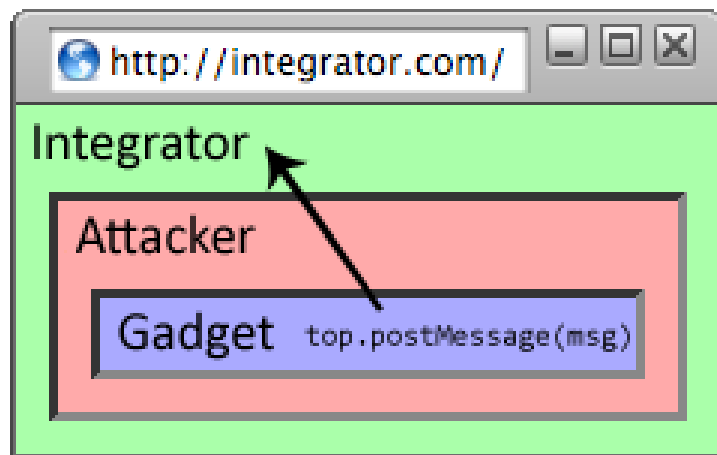
- ◆ With descendant frame navigation policy
- ◆ Attacker replaces inner frame with his own, gets message



Message Eavesdropping (2)

`frames[0].postMessage("Hello!")`

- ◆ With any frame navigation policy
- ◆ Attacker replaces child frame with his own, gets message



Who Sent the Message?



```
function msgReceiver(e) {  
  if(e.origin !== "http://hostA")
```

HTML Living Standard (whatwg.org)

Authors should check the origin attribute to ensure that messages are only accepted from domains that they expect to receive messages from

And If The Check Is Wrong?

www.bogusjumptime.com/exploit/

People

Like 1.6m

Search

HOME NEWS PHOTOS STYLE RED CARPET ROYALS TV WATCH BABIES PETS BEST OF 2012 CELEBS VIDEO MAGAZINE

NDSS 2013 call for papers

TOP STORY 09:45AM EST

SYMPOSIUM


THE LATEST MOST SHARED

The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites
10:00AM EST


The camera-ready due for NDSS 2013 is coming up
TV WATCH ONLY ON PEOPLE.COM 09:10AM EST

Internet Society 20 years
09:05AM EST

19th Annual Network & Distributed System Security Symposium



WHAT YOU ❤️ RIGHT NOW



READ IT ▶

Like 12k Tweet +1

The Postman Always Rings Twice

[Son and Shmatikov. "The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites". NDSS 2013]

A study of postMessage usage in top 10,000 sites

- ◆ 2,245 (22%) have a postMessage receiver
- ◆ 1,585 have a receiver without an origin check
- ◆ 262 have an incorrect origin check
- ◆ 84 have **exploitable vulnerabilities**
 - Received message is evaluated as a script, stored into localStorage, etc.

Incorrect Origin Checks

[Son and Shmatikov]

| Check | Hosts | Origin check |
|-------|-------|---|
| 1 | 107 | <code>if(/[\ \/\.]chartbeat.com\$/).test(a.origin))</code> |
| 2 | 71 | <code>if(m.origin.indexOf("sharethis.com") != -1)</code> |
| 3 | 35 | <code>if(a.origin && a.origin.match(/\.kissmetrics\.com/))</code> |
| 4 | 20 | <code>var w = /jumptime\.com(: [0 - 9])?\$/; if (!v.origin.match(w))</code> |
| 5 | 4 | <code>if(!a.origin.match(/readspeaker.com/gi))</code> |
| 6 | 1 | <code>a.origin.indexOf("widgets.ign.com") != 1</code> |
| 7 | 1 | <code>if(e.origin.match(/http(s?)\ : \ \/\ w+?\.?dastelefonbuch.de/)</code> |
| 8 | 1 | <code>if((/\api.weibo\.com\$/).test(l.origin))</code> |
| 9 | 1 | <code>if(/id.rambler.ru\$/i.test(a.origin))</code> |
| 10 | 1 | <code>if(e.origin.indexOf(location.hostname)==-1){return;}</code> |
| 11 | 7 | <code>if((/^(https? : \/[\ \/]+)\/. + (pss selector payment.portal matpay - remote).js/i) .exec(src)[1] == e.origin)</code> |
| 12 | 5 | <code>if(g.origin && g.origin !== l.origin) { return; } else { ... }</code> |
| 13 | 1 | <code>if((typeof d === "string" && (n.origin !== d && d !== ""))) j.isFunction(d) && d(n.origin) === !1))</code> |
| 14 | 24 | <code>if(event.origin != "http://cdn-static.liverail.com" && event.data)</code> |

Library Import

- ◆ Same origin policy does not apply to directly included scripts (not enclosed in an iframe)

```
<script type="text/javascript"  
src=http://WebAnalytics.com/analyticsScript.js>  
</script>
```



- This script has privileges of A.com, not WebAnalytics
 - Can change other pages from A.com origin, load more scripts

- ◆ Other forms of importing



SOP Does Not Control Sending

- ◆ Same origin policy (SOP) controls access to DOM
- ◆ **Active content (scripts) can send anywhere!**
 - No user involvement required
 - Can only read response from same origin

Sending a Cross-Domain GET

◆ Data must be URL encoded

``

Browser sends

`GET file.cgi?foo=1&bar=x%20y HTTP/1.1 to othersite.com`

◆ Can't send to some restricted ports

- For example, port 25 (SMTP)

◆ Can use GET for denial of service (DoS) attacks

- A popular site can DoS another site [Puppetnets]

Using Images to Send Data

◆ Communicate with other sites

```

```

◆ Hide resulting image

```

```

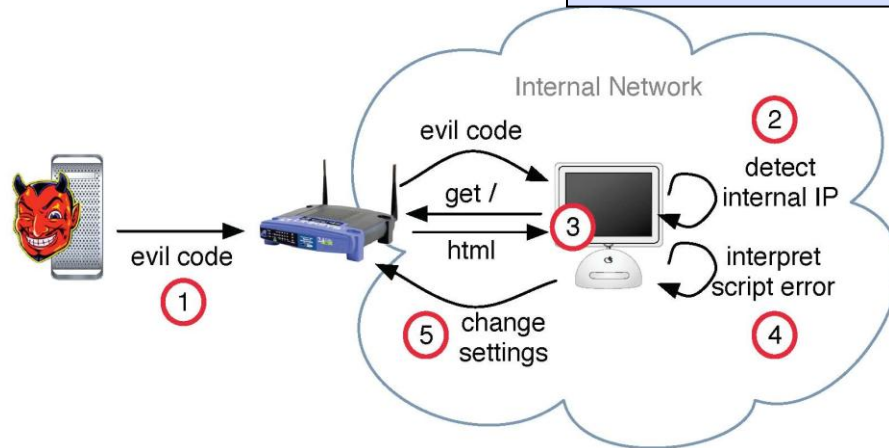


Very important point:

a web page can send information to any site!

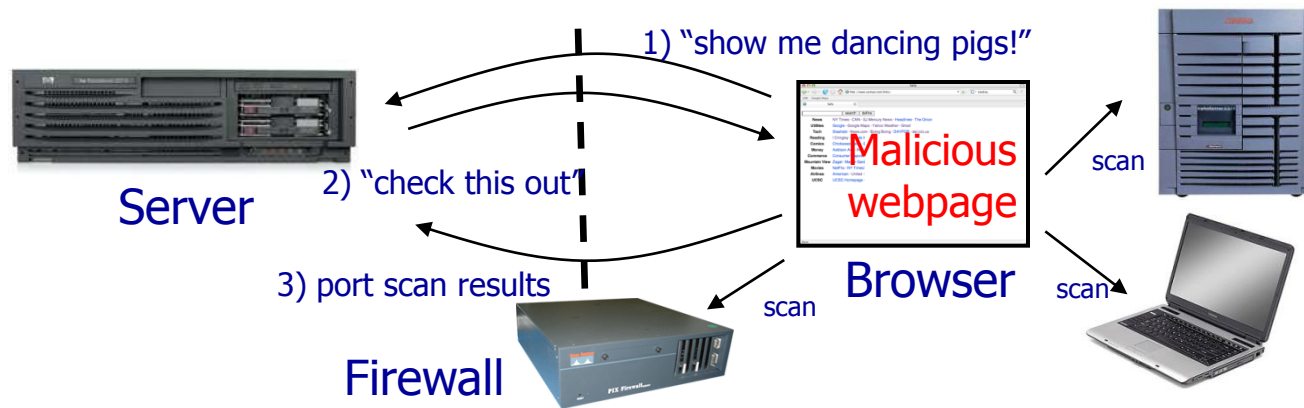
Drive-By Pharming

[Stamm et al. "Drive-By Pharming". 2006]



- ◆ User is tricked into visiting a malicious site
- ◆ Malicious script detects victim's address
 - Socket back to malicious host, read socket's address
- ◆ Next step: **reprogram the router**

Finding the Router



◆ Script from a malicious site can scan local network without violating the same origin policy!

- Pretend to fetch an image from an IP address
- Detect success using `onError`

``

Basic JavaScript function, triggered when error occurs loading a document or an image... can have a handler

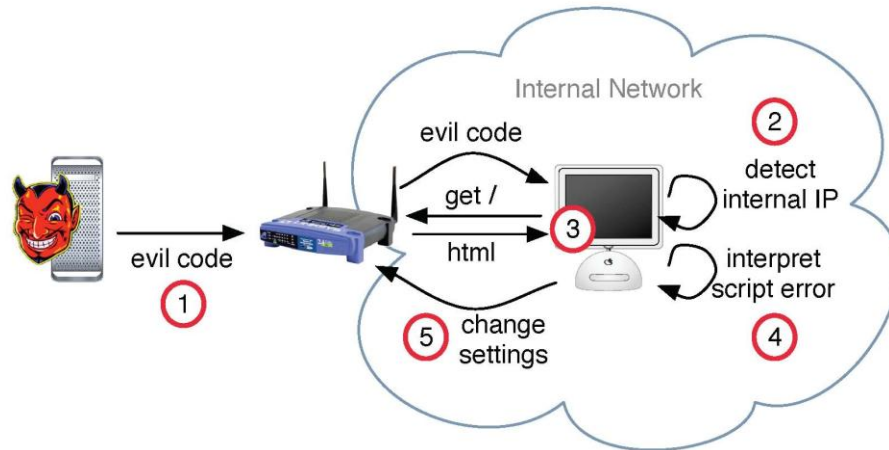
◆ Determine router type by the image it serves

JavaScript Timing Code (Sample)

```
<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>
```

When response header indicates that page is not an image, the browser stops and notifies JavaScript via the `onError` handle

Reprogramming the Router



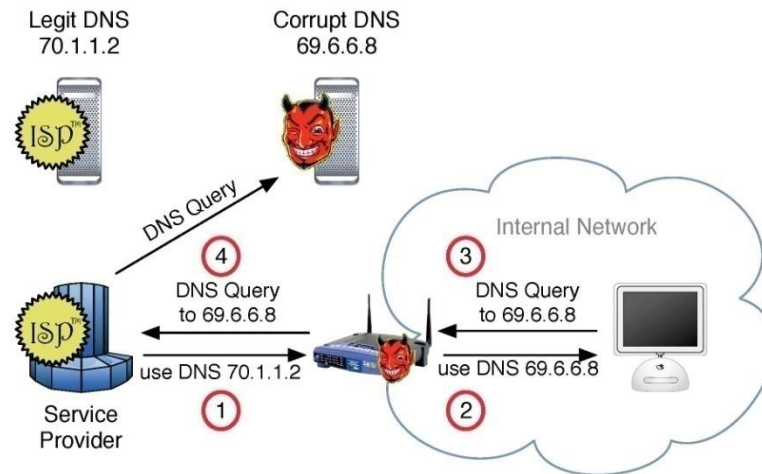
Fact: 50% of home users use a broadband router with a default or no password

◆ Log into router

```
<script src="http://admin:password@192.168.0.1"></script>
```

◆ Replace DNS server address with address of attacker-controlled DNS server

Risks of Drive-By Pharming



- ◆ **Completely Own the victim's Internet connection**
- ◆ Undetectable phishing: user goes to a financial site, attacker's DNS gives IP of attacker's site
- ◆ Subvert anti-virus updates, etc.